Alibaba Group 阿里巴巴集团

NEW RETAIL INTELLIGENCE ENGINE 新零售智能引擎事业群

GraphScope

A One-Stop Large-Scale Graph Computing System from Alibaba

White Paper

GraphScope: A One-Stop Large-Scale Graph Computing System from Alibaba

1.Background

1.1.What is graph computing

Graph models a set of objects (vertices) and their relationships (edges). As a sophisticated model, graphs can naturally express a large number of real-life datasets, such as social networks, Web graphs, transaction networks and knowledge graphs. Figure 1 shows an e-commerce graph in Alibaba, where there are various types of vertices (consumers, sellers, items and devices) and edges (purchase, view, comment and so on), and each vertex is associated with rich attribute information. Current graph data in real industrial scenarios usually contains billions of vertices and trillions of edges. In addition, continuous updates arrive at a tremendous speed. Given the ever-growing amount of graph data available, graph computing, which tries to explore underlying insights hidden in graph data, has attracted increasing attention in recent years.

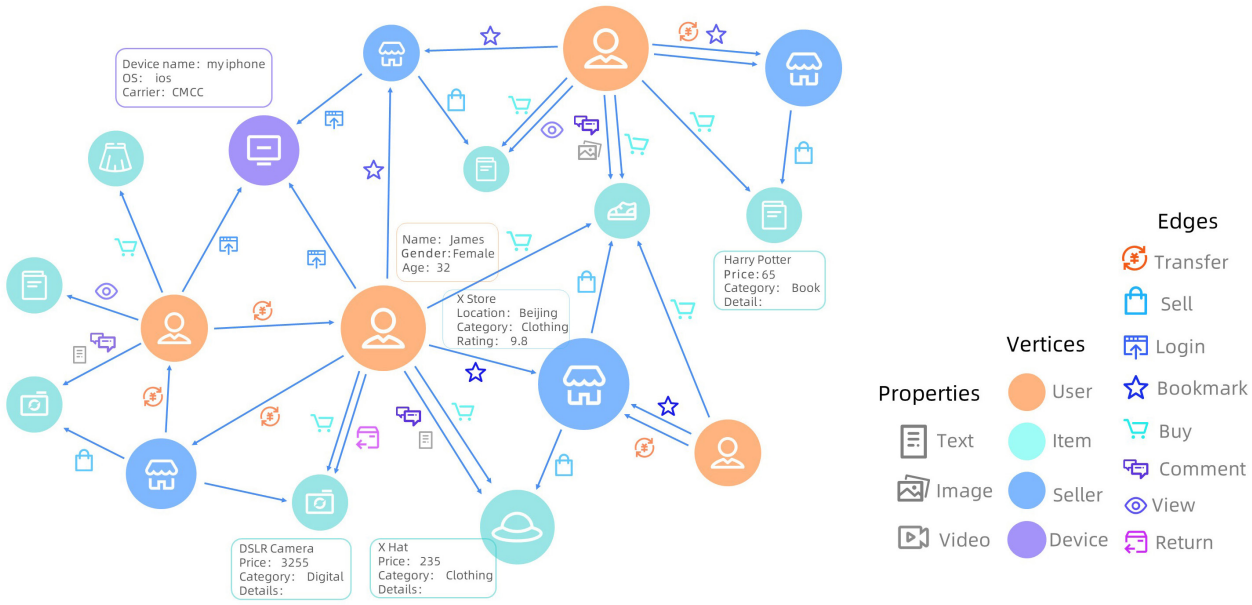

Figure 1: E-commerce graph at Alibaba

According to different aims and characteristics of graph-related tasks, current graph computing can be roughly divided into 3 categories, namely interactive queries on graphs, analytics on graphs and deep learning on graphs.

Interactive queries on graphs: Modern business often requires analyzing large-scale graphs in an exploratory manner in order to locate specific or in-depth information in time, as illustrated using the following example in Figure 2.

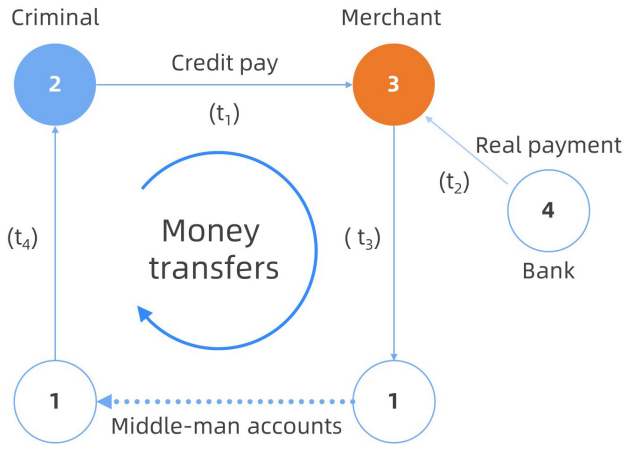

Figure 2: An example graph model for fraud detection using interactive query.

The graph depicted in Figure 2 is a simplified version of a real query employed at Alibaba for credit card fraud detection. By using a fake identifier, the "criminal" may obtain a short-term credit from a bank (vertex 4). He/she tries to illegally cash out money by forging a purchase (edge 2 --> 3) at time t_1 with the help of a merchant (vertex 3). Once receiving payment (edge 4 --> 3) from the bank (vertex 4), the merchant tries to send the money back (edges 3 --> 1 and 1 --> 2) to the "criminal" via multiple accounts of a middle man (vertex 1) at time t_3 and t_4, respectively. This pattern eventually forms a cycle (2 --> 3 --> 1 ... --> 2). Such fraudulent activities have become one of the major issues for online payments, where the graph could contain billions of vertices (e.g., users) and hundreds of billions to trillions of edges (e.g., payments). In reality, the entire fraudulent process can involve a complex chain of transactions, through many entities, with various constraints, which thus requires complex interactive analysis to identify.

Analytics on graphs: Analytics on graphs has been studied for decades, and tons of graph analytics algorithms have been proposed for different purposes. Typical graph analytics algorithms include general analytics algorithms (e.g., PageRank (see Figure 3), shortest path, and maximum flow), community detection algorithms (e.g., maximum clique/bi-clique, connected components, Louvain and label propagation), graph mining algorithms (e.g., frequent structure mining and graph pattern discovery). Due to the high diversity of graph analytics algorithms, programming models for graph analytics are needed. Current programming models can basically fall into the following categories:

"think like a vertex" , matrix algebra, "think like a graph/subgraph" and datalog. With these programming models in place, a lot of graph analytics systems have been developed, such as NetworkX, Pregel, PowerGraph, Apache Giraph and GRAPE.

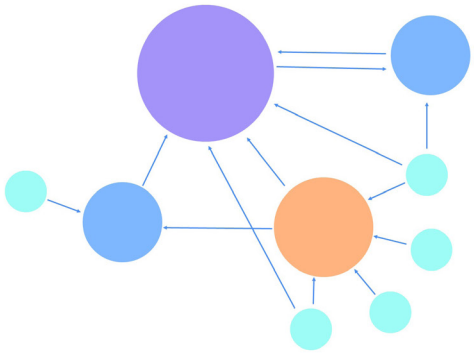

Figure 3: PageRank, an algorithm to assign an "importance" score for each vertex. In this figure, the diameter of each vertex indicates its PageRank score.

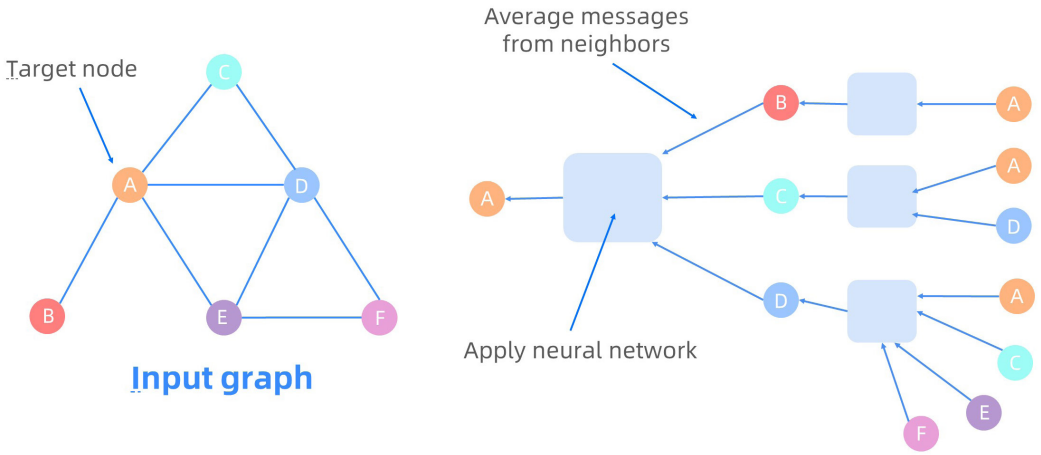

Figure 4: Graph neural network

Machine learning on graphs: Classic graph embedding techniques, such as Node2Vec and LINE, have been widely adopted in various machine learning scenarios. Recently Graph Neural Networks (GNNs) are proposed, which combine the structural and attribute information in the graph with the power of deep learning technologies. GNNs can learn a low-dimensional representation for any graph structure (e.g., a vertex, an edge, or an entire graph) in a graph, and the generated representations can be leveraged by many downstream graph-related machine learning tasks, such as vertex classification, link prediction, graph clustering. Graph learning technologies have demonstrated convincing performance on many graph-related tasks. Different from traditional machine learning tasks, graph learning tasks involve both graph-related and neural network operations (see Figure 4). Specifically, each vertex in the graph selects its neighbors using graph-related operations and aggregates its neighbors' features with neural network operations.

1.2.Graph computing: a foundation for the next generation of artificial intelligence

The study of graph computing algorithms and systems has been developing rapidly in recent years and become a hot topic in both academia and industry. In particular, the performance of graph computing systems has improved by 10 to 100 times over the past decade and the systems are still becoming increasingly efficient, making it possible to accelerate the AI and big data tasks via graph computing. In fact, graphs are able to naturally express data of various sophisticated types and can provide abstractions for common machine learning models. Compared to dense tensors, the graph representations also offer a much richer semantics and a more comprehensive capability for optimization. Moreover, graphs are a natural encoding of sparse high-dimensional data and the growing research literature in GCN (graph convolutional network) and GNN (graph neural network) has proven that graph computing is an effective complement to machine learning.

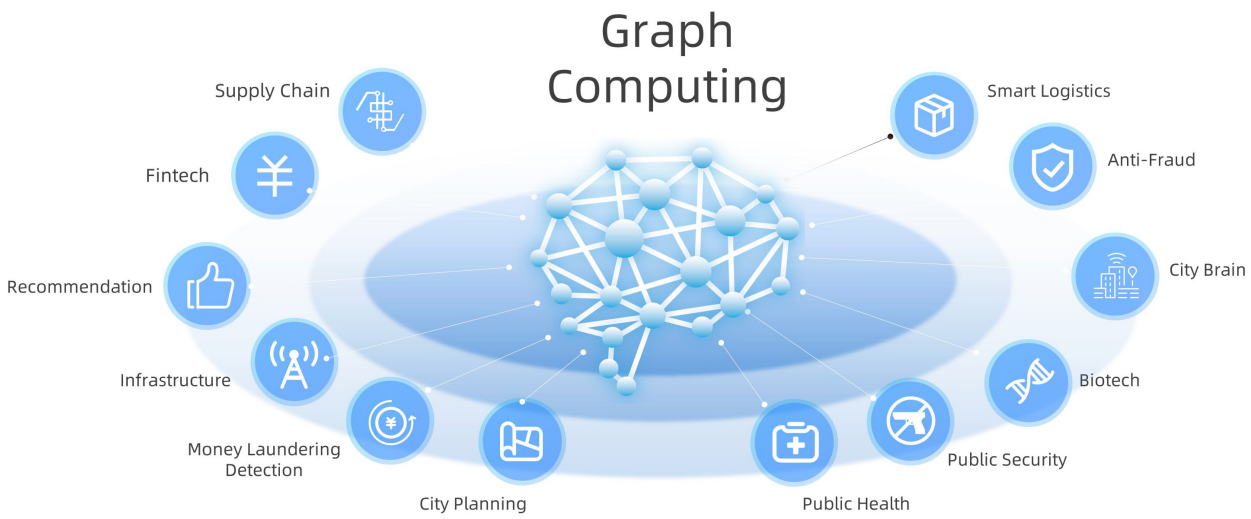

Figure 5: Applications of graph computing.

Putting these together, it is reasonable to expect that graph computing would play a big role in various applications within the next generation of artificial intelligence, including anti-fraud, intelligent logistics, city brain, bioinformatics, public security, public health, urban planning, anti-money laundering, infrastructures, recommender systems, financial technology and supply chains (see Figure 5).

1.3.Challenges of graph computing

Although graph computing has been considered as promising solutions for various applications, there is a huge gap between initial ideas and real productions. We summarize that the current large-scale graph computing faces the following challenges.

(1) Real-life graph applications are complex and diverse. In real-life scenarios, a graph-related task is typically very complex and involves multiple types of graph computing. Existing graph computing systems are mostly designed for a specific type of graph computation. Therefore, users have to disassemble a complex task into multiple jobs involving many systems. To bridge different systems, there could be significant overheads such as, extra costs on integration, I/O, format transformation, network and storage.

(2) It is difficult to develop applications for large graphs. To develop a graph computing application, users normally begin with small graphs on a single machine using easy- and ready-to-use tools (such as NetworkX in Python and TinkerPop). However, it is extremely difficult for average users to extend their single machine solution to handle large graphs in parallel. Existing distributed parallel systems for large graphs usually follow different programming models, and lack the rich ready-to-use libraries found in the single machine libraries (e.g., NetworkX). This makes distributed graph computing a privilege for experienced users only.

(3) The scale and efficiency of processing large graphs is still limited. Although current systems have largely benefited from years of work in optimizations for each individual system, they still suffer from efficiency and/or scale problems. For example, existing interactive graph query systems cannot parallelize the executions for Gremlin queries because of the high complexity of traversal patterns. For graph analytical systems, traditional vertex-centric model makes graph-level optimization unavailable. In addition, many existing systems lack optimizations at the compiler level.

2.GraphScope: a one-stop large-scale graph computing system

2.1.Introduction

To tackle the above three challenges, we propose GraphScope, a one-stop large-scale graph computing system. GraphScope aims to provide a single system that is able to support all three types of computation tasks, i.e., graph interactive query, graph analytics and graph deep learning. We carefully design GraphScope with user-friendly interface and extensible programming APIs, so that users can easily construct customized end-to-end graph processing pipelines involving different types of graph computation tasks. In specific, GraphScope fully embraces the Python and Gremlin ecosystem, and thus comes with a shallow learning curve for both data scientists and developers. Under the hood, GraphScope comprises core engines specifically optimized for each graph computation paradigm, and can smoothly orchestrate multiple engines to cooperate efficiently, avoiding the complexity of manually stitching multiple independent systems together. GraphScope can scale to ultra-large graphs, and run in industrial speed and robustness.

GraphScope has been battle-tested in production in Alibaba and over 30 external customers. GraphScope has supported graph processing tasks on extremely-large and complex graphs, which are of size more than 50TB, and consist of billions of vertices, hundreds of billions of edges, over one hundred labels, and more than one thousand attributes. GraphScope has demonstrated superior performance compared with state-of-the-art graph systems: on the industrial standard LDBC benchmark, GraphScope achieves up-to 5.22B EVPS on 4 nodes of Aliyun nodes for XL-size graphs on the analytical tasks, and nearly linear speed-ups on the SNB interactive queries tasks; GraphScope also speeds up the training time of graph learning models by 50%. In addition, GraphScope provides a rich set of built-in algorithm libraries, covering over 50 graph analysis and graph deep learning algorithms. These libraries, along with the user-friendly interface and APIs, significantly reduce the development cycle of end-to-end graph applications from weeks to days.

Our endeavor on GraphScope has also been recognized in both academia and industry. We have published tens of research papers on top tier conferences and journals. These research works have won SIGMOD 2017 Best Paper award and VLDB 2017 Best Demo award. Based on GraphScope, we developed a cognitive intelligence computing platform, which won the 'SAIL' prize of the 2019 World Conference on Artificial Intelligence.

2.2.Architecture overview

GraphScope is a full-fledged, in-production system for one-stop analysis on big graph data. Achieving this goal requires a wide variety of components to interact, including cluster management (deployment) software, graph store (as input, output and to hold intermediate results), distributed execution engines, language constructs, and development tools. Due to the space limit, we highlight the three major layers in GraphScope, namely algorithm, execution, and storage as shown in Figure 6, and give an overview to each of them below.

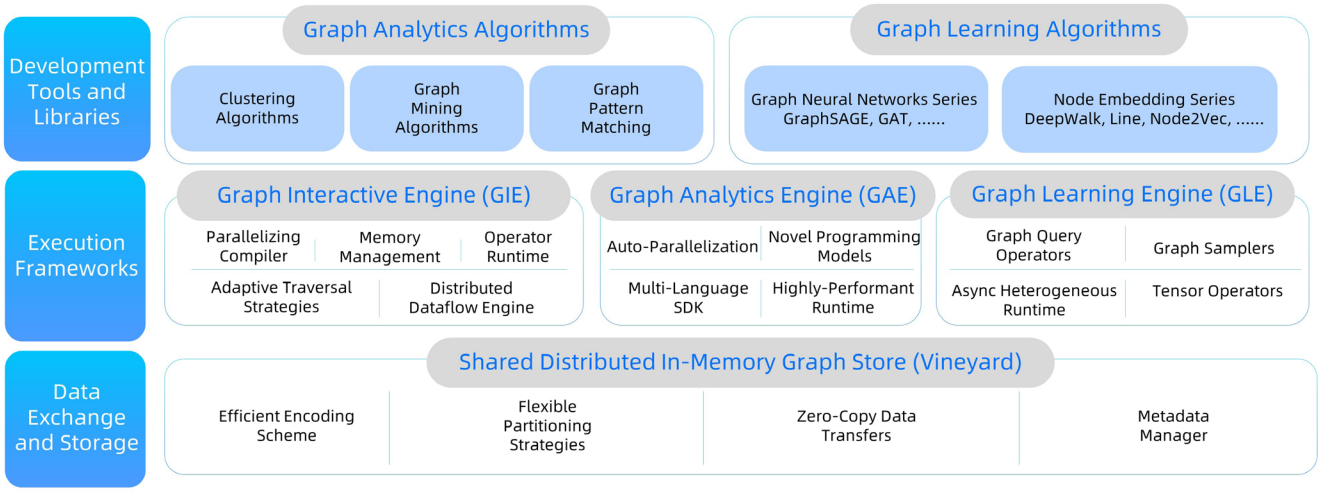

Figure 6: Architecture of GraphScope

Algorithm. Graph processing often requires specific algorithms for each particular task. While those algorithms can be directly written using the GraphScope's primitives, we have built libraries of common algorithms for various application domains (such as graph neural networks, clustering, and pattern matching) to ease the development of new graph applications. GraphScope gives the programmer the illusion of writing for a single machine using a general purpose high-level programming model (Python) and to have the system deal with the complexities that arise from distributed execution. This has tremendously simplified the defining and maintaining of such libraries. In addition, this approach allows GraphScope to seamlessly combine multiple graph processing engines in one unified platform as described below.

Execution. GraphScope execution runtime consists of three engines, namely GraphScope Interactive Engine (GIE), GraphScope Analytics Engine (GAE), GraphScope Learning Engine (GLE), and provides the functionality of interactive, analytical, and graph-based machine learning, respectively. A common feature all those execution engines provide is the automatic support for efficient distributed execution of queries and algorithms in their target domains.

Each query/algorithm is automatically and transparently compiled by GraphScope into a distributed execution plan that is partitioned across multiple compute nodes for parallel execution. Each partition runs on a separate compute node, managed by a local executor, that schedules and executes computation on a multi-core server.

Storage. Because the graph datasets are large, it can take a long time to load input into and save output from multiple processing stages (between different engines) within a complex pipeline. To mitigate such cost, GraphScope provides an in-memory storage layer called Vineyard that maintains an (intermediate) data object partitioned across a cluster. The storage is tightly coupled with the execution engines for efficiency, so that each local executor (of any engine) can access graph data completely avoiding unnecessary data copy. Furthermore, the storage provides high-level abstractions or data structures such as (sub)graphs, matrices and tensors as fundamental interfaces to its clients so as to minimize serialization and deserialization cost as well.

2.3. Components

2.3.1. GIE: a parallel interactive engine for graph traversal

| Challenges of parallelizing the interactive graph query

Different from an analytic query that may run minutes to hours without much human involvement, an interactive query allows human to interact with graph data in real time typically using a high-level, declarative query language. Because of such features, interactive query enables human, often non-technical users, to directly explore, examine and present data in order to locate specific or in-depth information at low latency, and is commonly recognized as an essential part of any data analytics project.

GIE exploits the Gremlin graph traversal language developed by Apache TinkerPop to provide a high-level language for interactive graph queries and provides automatic parallel execution. Gremlin is widely supported by popular graph system vendors such as Neo4j, OrientDB, JanusGraph, Azure Cosmos DB, and Amazon Neptune, which offers a flexible and expressive programming model to enable non-technical users to succinctly express complex traversal patterns in real-world applications. For example, one can write the above fraud-detection query (Figure 2) in just a couple of lines using Gremlin, as shown in Figure 7.

```
g.V('account').has('id','2').as('s')
 .repeat(out('transfer').simplePath())
 .times(k-1)
 .where(out('transfer').eq('s'))
 .path().limit(1)
```

Figure 7: An example Gremlin query

The flexibility of Gremlin mainly stems from nested traversal with dynamic control flow such as loops, which introduces fine-grained data dependencies at runtime that are complex and can incur significant overheads in distributed execution. Therefore, existing Gremlin-enabled, large-scale systems either adopt centralized query processing (such as JanusGraph and Neptune), or offer a subset of the language constructs that is often too limited for real-world applications (in production at Alibaba), or come as a huge performance sacrifice (e.g. Hadoop-Gremlin). In addition, such a system must cope with runtime dynamics related to variations in memory consumption in an interactive context. While several techniques exist for alleviating memory scarcity in distributed execution, such as backpressure and memory swapping, they cannot be directly applied due to potential deadlocks or big latency penalty.

| Efficient graph interactive engine

We tackle the challenges to scale the Gremlin queries by a novel distributed-system infrastructure designed specifically to make it easy for a variety of users to interactively analyze big graph data on large clusters at low latency. i) GIE compiles a Gremlin query into a dataflow graph that can be mapped to physical machines for distributed execution, ii) operators in Gremlin are precompiled and installed on each compute node to allow query plans to be dispatched at low latency, iii) each local executor employs dynamic scheduling and works together to optimize execution dynamically (to cope with runtime dynamics related to variations in memory usage and ensure bounded-memory execution), iv) finally, the same runtime can be dynamically reconfigured by user-defined graph traversal strategies (such as depth-first or breadth-first search) to achieve low latency by avoiding wasted computation. All of the above mechanisms are made possible by a powerful new abstraction we developed in GIE that caters to the specific needs in this new computation model to scale graph queries with complex dependencies and runtime dynamics, while at the same time maintaining the simple and concise programming model.

The interactive engine has been deployed in production clusters at Alibaba to support a variety of business-critical scenarios. Extensive evaluations using both benchmarks and real-world applications have validated the high-performance and scalability of GIE. Compared to the Gremlin-enabled graph database JanusGraph, GIE outperforms JanusGraph by over one order of magnitude on average using the industry-standard LDBC benchmark. Additionally, GIE can scale to much larger graphs. In the benchmark, we have adopted the largest graph that LDBC benchmark can generate, which contains over 2 billion vertices, 17 billion edges and occupies 2TB aggregated memory in the cluster; in production, GIE has been deployed in Alibaba cluster to process gigantic graphs with hundreds of billions of edges.

2.3.2. GAE: a high-performance graph analytics engine

| Challenges

In response to the need of analyzing big graphs, several parallel graph analytics engines have been developed, e.g., Pregel, GraphLab, Trinity, GRACE, Blogel, Giraph++, and GraphX. However, users often find it hard to write and debug parallel graph programs using these systems. The most popular programming model for parallel graph algorithms is the vertex-centric model, pioneered by Pregel and GraphLab. Although graph analytics computing has been studied for decades and a large number of sequential (single-machine) graph algorithms are already in place, to use the vertex-centric model, one has to recast the existing sequential algorithms into vertex-centric programs. The recasting is nontrivial for users who are not very familiar with the parallel models. Moreover, none of the systems provides guarantee on the correctness or even termination of parallel programs developed in their models. These make the existing systems a privilege for experienced users only.

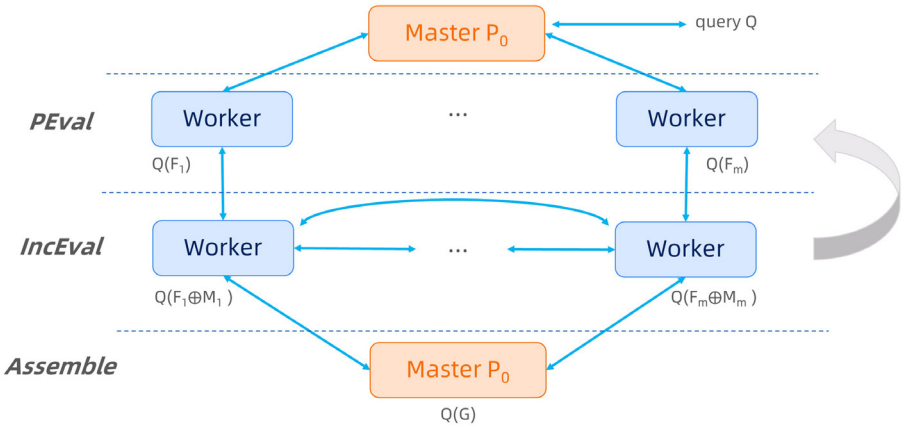

Figure 8: Programming model used in GAE

To reduce the programming burden from users while achieving high performance at the same time, we develop a large-scale parallel graph analytics engine, referred to as GAE.

| Auto parallelization

GAE follows the GRAPE for auto-prallelization, which proposed in the paper "*Wenfei Fan, Wenyuan Yu, Jingbo Xu, Jingren Zhou, Xiaojian Luo, Qiang Yin, Ping Lu, Yang Cao, Ruiqi Xu: Parallelizing Sequential Graph Computations. ACM Trans. Database Syst. 43(4): 18:1-18:39 (2018)*". GAE supports a simple paradigm such that to implement a graph analytics algorithm, users only need to provide three sequential (incremental) functions, (1) PEval, a sequential function for given a query, computes the answer on a local graph; (2) IncEval, a sequential incremental function, computes changes to the old output by treating messages among different workers as updates; and (3) Assemble, which collects partial answers that are computed locally at each worker by PEval and IncEval, and combines them into a complete answer. In this model, users do not need to know the details of the distributed setting while processing big graphs in a cluster, and GAE auto-parallelizes the graph analytics tasks across a cluster of workers, based on a fixpoint computation. Under a monotonic condition, it guarantees to converge with correct answers as long as the three sequential algorithms provided are correct. That

| Flexible programming models

is, GAE parallelizes sequential algorithms as a whole. This makes parallel computations accessible to users who know conventional graph algorithms covered in college textbooks, and there is no need to recast existing graph algorithms into a new model.

Existing parallel programming models can be easily adapted and executed on GAE. GAE works on a graph G fragmented via a partition strategy picked by the user and each worker maintains a fragment of G. Given a query, GAE posts the same query to all the workers. As shown in the Figure 8, it computes Q(G) in three phases following BSP (Bulk Synchronous Parallel). More specifically, each worker first executes PEval against its local fragment, to compute partial answers in parallel. This facilities data-partitioned parallelism via partial evaluation. Then each worker may exchange partial results with other processors via synchronous message passing. Upon receiving messages, each worker incrementally computes IncEval. The incremental step iterates until no further messages can be generated. At this point, Assemble pulls partial answers and assembles the final result.

| Multi-language SDKs

Multi-language SDKs are provided by GAE. Users choose to write their own algorithms in either C++ or Python. With Python, users can still expect a high performance. GAE integrated a compiler built with Cython. It can generate efficient native code from Python algorithms behind the scenes, and dispatch the code to the GraphScope cluster for execution. The SDKs further lower the total cost of ownership of graph analytics.

| High-performance runtime

GAE achieves high performance through a highly optimized analytical runtime based on libgrape-lite. Many optimization techniques, such as pull/push dynamic switching, cache-efficient memory layout,

2.3.3.GLE: an end-to-end graph learning framework

| Challenges

Graph deep learning algorithms, including graph embedding (GE) and graph neural networks (GNN), have increasingly attracted a lot of interests in both academia and industry since the last decade. Recently, the rising of big data and complex systems brings a quick proliferation of graph data and reveals new insights. We have observed four properties in the vast majority of real-world graph data, namely large-scale, heterogeneous, attributed and dynamic. For example, nowadays e-commerce graphs often contain billions of vertices and tens of billions of edges, with various types and rich attributes, and quickly evolve over time. These properties pose great challenges for embedding and representing graph data:

• Graph data, very different from other forms of data, usually exhibits structural irregularity in Euclidean space. It is challenging to scale graph learning algorithms well on real-world graphs with extremely large sizes. Thus, it is a top priority for graph deep learning engines to ensure the time and space efficiencies on large-scale graphs.

• Attributed heterogeneous graphs usually contain various types of vertex and edges with different attributes. This rich information is critical for leveraging both inductive and transductive settings and enhancing the representation power of a graph deep learning algorithm. However, it is non-trivial to integrate both the topological structure information and the unstructured attribute information in a unified embedding space.

| An industrial-scale graph deep learning engine

GLE is designed for industrial scenarios at the very beginning, and thus is able to efficiently handle large-scale, heterogeneous graph data. We carefully design GLE to be light-weight, portable, extensible, and easily customizable for various kinds of tasks: GLE provides a set of user-friendly programming APIs essential for developing an end-to-end graph learning application, and can smoothly co-work with popular deep learning engines such as TensorFlow and PyTorch to implement task oriented neural network layers.

| User-friendly interface

Graph sampling is an essential step in large-scale graph learning. To ease the development of graph sampling, we propose a high-level language named as GSL (Graph Sampling Language). GSL features a Gremlin-like syntax, which is a widely adopted graph query language. With GSL, a sampling query can be implemented as a traversal following a user-defined pattern, allowing users to apply customized sampling logics at corresponding vertices and edges. Figure 9 shows an example GSL query, which samples a batch of 512 vertices of type "vertex_type", and in turn for each vertex samples 10 neighbors following the "edge_type" edges.

```
graph.V("vertex_type").shuffle().batch(512)
     .outV("edge_type").sample(10).by("random");
```

Figure 9: An example graph sampling query

| Modularized and extensible design

As shown in Figure 10, we design GLE in a modularized approach, where each module can be extended independently. This design enables GLE to keep up with the pace of the vibrant research and industrial advances in this field. Specifically, GLE abstracts out four layers: graph data layer, sampler layer, tensor operator layer, and algorithm layer. The graph data layer provides basic graph query functionalities, allowing users to query the attribute/label data, vertices and edges of a graph. Users can extend this layer to adapt to their customized graph stores. The sampler layer provides three types of sampling operators (traverse sampler, neighbor sampler and negative sampler) and a rich set of built-in sampling operators implementations. The tensor operator layer comprises tensor operations used in graph neural networks. Users can easily plugin their own operators into these two

layers when developing new algorithms. The algorithm layer is built on top of the above three layers. We have also incorporated in GLE a library of popular graph learning algorithms, including both graph neural network algorithms and node embedding algorithms.

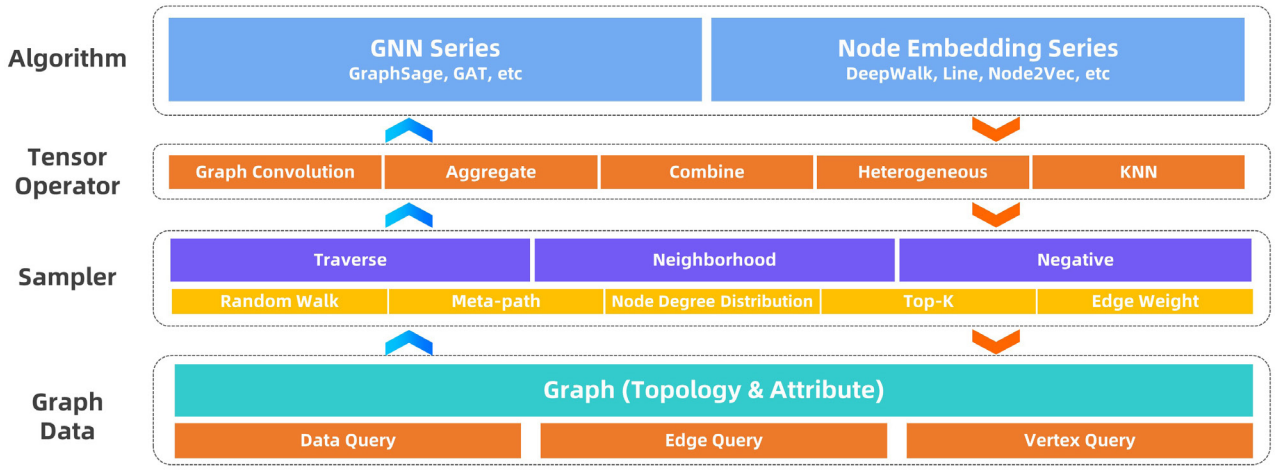

Figure 10: Modularized design of GLE

| Effective distributed runtime

To further optimize the sampling performance, GLE caches the remote neighbors for vertices that are visited frequently. In addition, the attribute indexes are also cached to speed up attribute lookups for vertices in each partition. GLE adopts an asynchronous execution engine with the support for heterogeneous hardware, which enables GLE to efficiently overlap a huge number of concurrent operations such as I/Os, sampling and tensor computation. GLE abstracts heterogeneous computation hardware as resource pools, e.g., CPU thread pool and GPU stream pool, and cooperatively schedules fine-grained concurrent tasks.

2.3.4.Vineyard: an in-memory immutable data manager

| Challenges

In a common graph computing practice, several different computing systems are involved to tackle different kinds of workloads. Existing solutions usually adopt distributed databases or file systems as the intermedia storage to share distributed data between heterogeneous computing systems that are involved in the task. This brings two significant overheads: 1) the structural data are transformed from/to the external data storage format (e.g., tables in relational databases, files in HDFS) back and forth in the beginning/end of each computation step. Meanwhile, the structure and operations of the data are dismissed. 2) saving/loading the data to/from the external storage requires lots of memory-copies and disk-IO costs, which becomes the bottleneck of the entire process in more and more cases as the efficiency of the computing systems are growing rapidly. In addition, the lack of managing the data uniformly through the big data task obstructs the application of modern techniques such as data monitoring, data-aware optimization, and fault-tolerance, thus, further decreases the productive efficiency.

| A distributed in-memory data manager

To bridge different graph computation systems, a distributed in-memory data manager Vineyard is developed. It provides 1) in-memory distributed immutable data sharing in a zero-copy fashion to avoid introducing extra I/O costs, 2) some built-in out-of-box high-level data abstractions with efficient underlying memory layout to share the distributed data with complex structures (e.g., distributed graphs). 3) extensible mechanism to allow to transplant user-defined data structures or functionalities (e.g., graph partitioning algorithms) into Vineyard. 4) metadata management. With Vineyard in place, users can handle large-scale distributed data across different graph computing systems as simple and efficient as local variables, and finish a graph-related task in an end-to-end way.

| Data sharing with zero-cost

Vineyard supports various flexible and efficient memory layout for partitioned immutable graph data. Both simple graphs and property graphs are supported. And it utilized a columnar layout of properties of graphs, which can speed-up the computing tasks and make it easier to interact with other data tasks with Apache Arrow efficiently.

| Effective graph partitions with extensible design

Vineyard employs the extensible design concept of registry mechanism to facilitate users transplanting their defined data structures into Vineyard. In particular, the extensible design involves three components builders, resolvers and drivers, and allows users to build, resolve and share their data structures easily among different systems and programming languages respectively. In general, the registry mechanism decouples the functionality methods from the definition of Vineyard data structures. For builders and resolvers, users can flexibly register different implementations in different programming languages to build and resolve the same Vineyard data structures, which makes the data structures available to share among different systems and programming languages, and makes it possible to exploit native language optimizations. For drivers, the registry mechanism allows users to flexibly plug-in functionality methods in different programming languages for Vineyard data structures, which assigns required capability to the data structures along with the data analytics process. For graph computing, graph partitioning is critical to both analytics or interactive processing. With the registry mechanism, users can easily extend Vineyard by plugin their own graph partitioning drivers, which can be implemented and optimized in accordance with specific graph computation tasks for further efficiency augmentation.

| Metadata management

In addition, Vineyard provides management for the metadata of the data stored in Vineyard. It keeps the structures, layouts and properties of the data to construct high-level abstractions (e.g., graphs, tensors, dataframes). The metadata managers in a Vineyard cluster communicate with each other through the backend key-value store, e.g., etcd server, to keep the consistency of the distributed data stored in Vineyard.

3.Case study: anti-fraud and risk control

With the rapid growth of Taobao users, there has been an increasing volume of fraudulent user behaviors on the Taobao platform. These fraudulent behaviors, such as spam transactions, fake comments, risky traffic, bring various threats to our ecosystem. Generally, all the users and transactions can be organized as a large-scale heterogeneous graph which contains both different types of vertices such as buyers, sellers and items, and different types of edges with edge attributes extracted from various retailing scenarios. Facing the heterogeneous graph, GraphScope integrates three types of tools, i.e., graph analytics, graph interactive query, and graph deep learning tools, to enable automatically to recognize spam transactions, fake comments, and risky traffic from huge volumes of Taobao transactions. Moreover, because fraudulent users grow rapidly towards highly organized groups, GraphScope can also identify fraudulent communities by using the graph interactive query tools. To fulfill the need of discovering fraudulent behaviors in real-time, GraphScope implements graph deep learning models especially graph neural networks to recognize and process fake behaviors within a short period of time.

One-stop graph computing for fraudulent community detection. GraphScope can identify fraudulent communities by combining the components of graph analytics, graph interactive query and graph deep learning. Figure 11 displays how GraphScope helps fraudulent community detection in Alibaba. First of all, graph analytics can extract discriminative patterns of each single fraudulent user. We first build a bipartite graph with vertices and edges representing buyers and their purchases of goods. Then, a Biclique subgraph pattern matching algorithm in graph analytics can be used to detect purchase fraud behavior.

Then, after processing by graph analytics engine, GraphScope calls the component of graph learning to further discover deep complicated patterns of fraudulent users. Graph learning builds a semi-supervised learning model on the large-scale user purchase graph. Specifically, graph neural networks (GNNs) are used to integrate the attribute information of vertices and edges with local network structure information to learn deep network representations.

At the last step, GraphScope uses the component of graph interactive query to discover fraudulent communities. Specifically, label propagation algorithms (LPA) in the component of graph traversal can discover fraudulent communities by spreading the class labels of fraudulent users discovered by graph analytics and graph learning. LPA runs in a large-scale decentralized graph, and discovers overlapping fraud communities.

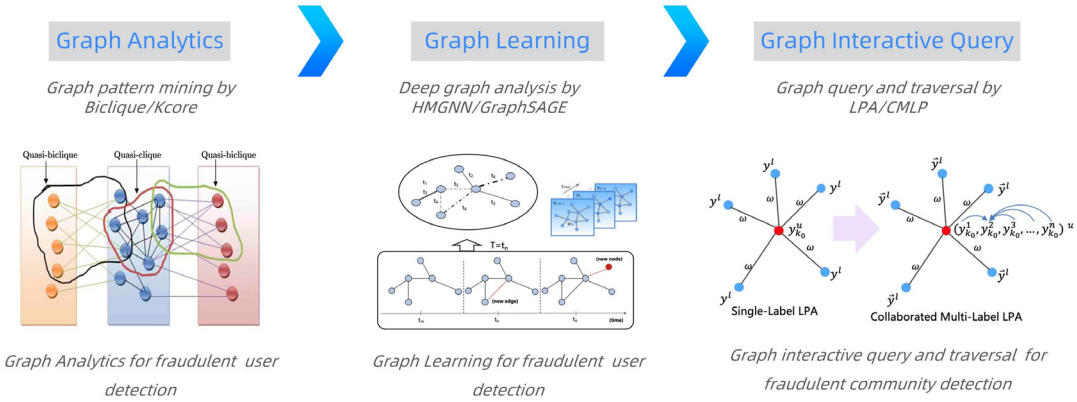

Figure 11: One-stop graph computing for fraudulent community detection

4.The future of GraphScope

According to our plan, the GraphScope is to be open-sourced by *December 2020** under Apache License 2.0. Here at Alibaba, we are fully committed to the long-term active development and maintenance of the project. We aim for a major release every 6 months with new features and improvements. Following the initial release, the following new capabilities are planned for the next major release around *June 2021**, please stay tuned :

1. NetworkX-API support
2. A persistent dynamic graph storage
3. Performance improvement over runtime and compiler modules
4. Java SDK for developing analytics algorithms

We also pledge to foster an open and welcoming graph computing community around the project. We encourage everyone to try and contribute to GraphScope to improve this project.

5.Conclusion

Because of its rapid evolution and rich expressivity, graph computing demonstrates a huge potential to unlock many of the next generation AI applications. In this white paper, we have introduced GraphScope, a one-stop large-scale graph computing system to be open-sourced by Alibaba. By providing a unified system for analytics, interactive queries and deep learning over graphs, GraphScope addresses many of the challenges encountered in existing systems. It is designed as a one-stop, user-friendly and highly performant system for industrial-scale graphs and applications. We believe in the future of graph computing, and we are fully committed to the future development of GraphScope. Welcome to join the force with us!

* These dates are subject to change

GraphScope

A One-Stop Large-Scale Graph Computing System from Alibaba