GraphScope: A One-Stop Large Graph Processing System

Jingbo Xu, Zhanning Bai, Wenfei Fan, Longbin Lai, Xue Li, Zhao Li, Zhengping Qian, Lei Wang, Yanyan Wang, Wenyuan Yu, Jingren Zhou
Alibaba Group Ant Group University of Edinburgh Shenzhen Institute of Computing Sciences
graphscope@alibaba-inc.com

ABSTRACT

Due to diverse graph data and algorithms, programming and orchestration of complex computation pipelines have become the major challenges to making use of graph applications for Web-scale data analysis. GraphScope aims to provide a one-stop and efficient solution for a wide range of graph computations at scale. It extends previous systems by offering a unified and high-level programming interface and allowing the seamless integration of specialized graph engines in a general data-parallel computing environment.

As we will show in this demo, GraphScope enables developers to write sequential graph programs in Python and provides automatic parallel execution on a cluster. This further allows GraphScope to seamlessly integrate with existing data processing systems in PyData ecosystem. To validate GraphScope's efficiency, we will compare a complex, multi-staged processing pipeline for a real-life fraud detection task with a manually assembled implementation comprising multiple systems. GraphScope achieves a 2.86× speedup on a trillion-scale graph in real production at Alibaba.

PVLDB Reference Format:
Jingbo Xu, Zhanning Bai, Wenfei Fan, Longbin Lai, Xue Li, Zhao Li, Zhengping Qian, Lei Wang, Yanyan Wang, Wenyuan Yu, Jingren Zhou . GraphScope: A One-Stop Large Graph Processing System. PVLDB, 14(12): 2703 - 2706, 2021.
doi:10.14778/3476311.3476324

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://vldb.graphscope.app/.

1 INTRODUCTION

Distributed execution systems with high-level language support (*e.g.,* Koalas [7], Dask [2] and TensorFlow [9]) have been widely adopted for web-scale data analysis in a wide range of applications, such as e-commerce, on-line payments, and communication, largely attributed to ease of programming and scalability. However, the operator semantics provided by these systems is ill-suited to an important class of applications that require deeper analysis of complex interrelationships among data, where diverse analytics tools involving various graph computations are often called for instead.

For example, in e-commerce platforms, some sellers and buyers may conduct fraudulent transactions and reviews *collaboratively* in order to inflate their ratings and rankings. Recent studies [5, 10] show that by integrating various graph-based algorithms, the fraud detection task is able to better catch the collaborative nature of fraud behaviors, and thus achieve higher efficacy. However, the task is challenging to develop and deploy in real-life. Figure 1 illustrates a simplified fraud detection task on graph, where entities like sellers, buyers and items are represented as vertices, and relations like transactions are represented as edges. From other sources, such as customer complaints, some entities are marked with *known fraud* labels (red vertices in Figure 1). The aim for the task is to determine whether a given seller or buyer is involved in frauds based on structural and attribute information of the graph.

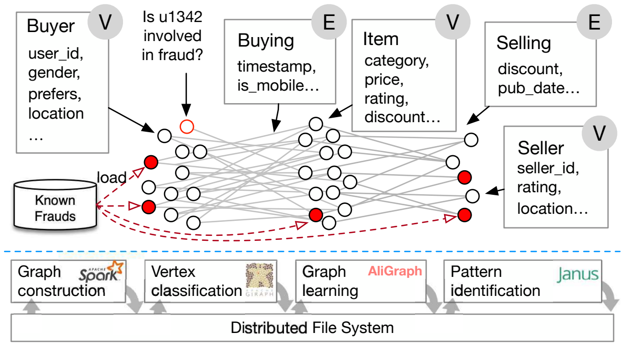


Figure 1: Fraud detection on a transaction graph.

The entire pipeline consists of the following four steps. First, the graph is constructed in a data-parallel system from the external storage (*e.g.,* HDFS). Next, a label propagation algorithm is performed in a graph analytics system (*e.g.,* Apache Giraph) for identifying a candidate set of vertices as possible fraudulent entities. Then, a graph learning system (*e.g.,* AliGraph [11]) is applied to conduct k-hop neighborhood sampling for each vertex, and the result is fed into a deep learning framework (*e.g.,* TensorFlow) to predict whether an entity is fraudulent, based on a graph neural network (GNN) model. Finally, experts may verify the results manually with interactive graph traversal queries, which typically depends on a graph database (*e.g.,* JanusGraph). We can see that this pipeline involves diverse types of graph computations (*i.e.,* graph iterative algorithms, graph traversal and graph sampling), as well as different types of computations in separate systems. Moreover, this graph-related task involves non-graph computations (*e.g.,* neural networks), and has to co-work with other data processing systems.

With the combination of different systems come the following drawbacks. First, existing graph processing systems are often designed for a specific type of computation with potentially very different programming abstractions and runtime. This gives rise to a lot of programming challenges, *e.g.,* managing the heterogeneity of data representation and complexity of resource scheduling.

Proceedings of the VLDB Endowment, Vol. 14, No. 12 ISSN 2150-8097.
doi:10.14778/3476311.3476324

Second, many systems (*e.g.,* Apache Giraph) require a deep understanding of low-level programming abstractions, which makes graph computations accessible to only graph expertise. Last but not least, inter-operating with other systems (*e.g.,* Spark) usually involves excessive data transformation and movement, which may significantly deteriorate the performance of the overall execution.

In this paper, we show how GraphScope system tackles the above challenges. We will demonstrate the following unique features.

(1) A unified high-level programming interface designed for a wide variety of graph computations, including graph traversal, pattern matching, iterative algorithms, and graph sampling (for GNN). The interface is embedded in Python so as to allow GraphScope to seamlessly integrate with other systems.

(2) A full-fledged system, GraphScope, that can automatically compile sequential graph programs into distributed computations.

(3) A distributed in-memory store, Vineyard, that provides efficient data transfer across multiple systems.

GraphScope outperforms state-of-the-art systems that are designed for different types of graph queries. It runs 2.86× faster than a manually assembled pipeline with complex, multi-staged processing on large graphs in a real-life application at Alibaba.

2 PROGRAMMING WITH GRAPHSCOPE

In this section we highlight some particularly useful and distinctive aspects of the GraphScope programming interface. In Figure 2, we sketch the codes of the fraud-detection case discussed in Section 1 as a running example. On top is a unified Python interface to embrace the Python ecosystem. For example, in line 4 of Figure 2, we use PySpark for data loading, and the code in line 19-22 introduces a TensorFlow procedure of a graph neural network to do classification. The graph computation, as the core of GraphScope, is supported by extending Gremlin (with its Python library) as detailed below.

2.1 Gremlin

Gremlin [1] is a widely used graph traversal language that allows high-level and declarative programming for various graph operations, including graph traversal, pattern matching, and sampling. In Gremlin, data is represented as streams of *traversers*. At the minimum, a traverser consists of three parts: a reference to the current location (vertex, edge or property), the path history, and (optionally) an application state called *sack*. A traverser is the basic data processing unit in a Gremlin computation.

A Gremlin query consists of a series of operators, each of which takes traverser streams as input, conducts computation as instructed, and outputs traverser streams. For instance, Gremlin offers familiar relational operators (as line 27 in Figure 2 shows), including projection (`select`), filters (`has`), grouping (`group`), and top-K (`limit`), together with dynamic control-flow constructs such as conditional (`where`) and loop (`repeat`) statements.

2.2 GraphScope Extensions of Gremlin

It is easy to use Gremlin to express a variety of graph computations such as graph traversal, pattern matching and sampling. However, it is a pain to do so for another useful type of graph operation: the iterative computation. This is mostly because Gremlin does not provide a mechanism to maintain states on the vertices that are necessary for iterative computation. To address the issue, we extend Gremlin with a new `process` step and several other operators that allow the users to easily program both the vertex-centric [6] and graph-centric [3] algorithms for iterative computation.

```
1  # Build the graph
2  graph = sess.g().add_vertices(...).add_edges(...)
3  # Load the known labels
4  known_fraud = load_from('oss://.../fraud_score.tsv')
5  graph = graph.add_column('label', known_fraud)
6  # Call LPA algorithm
7  g = sess.gremlin(graph).traversal_source()
8  g.V().process(
9    V().property('$lpa_score', expr('label'))
10   .repeat(
11     V().scatter('$lpa_score').by(out())
12     .gather('$tmp', sum)
13     .property('$new_score', expr('$tmp/IN_DEGREE'))
14     .where(expr('abs($new_score-$lpa_score)>1e-10'))
15     .property('$lpa_score', expr('$new_score'))
16   ).until(count().is(0))
17 ).withProperty('$lpa_score', 'lpa_score')
18 # Load a GraphSage Model for prediction
19 from graphscope.learning.models import GraphSage
20 model = GraphSage('oss://../some_model')
21 sampler = g.sample(...).toTensorFlowDataset()
22 scores = model.predict(feed=sampler, config=...)
23 # Add "fraud" score back to graph
24 graph = graph.add_column('score', scores)
25 # Interactive explorations on the result graph
26 g = sess.gremlin(graph).traversal_source()
27 g.V().has('id', 'u1324').out('buy').where(__.in('buy').
       count().is(gt(10000))).order().by('score', desc)
```

Figure 2: The GraphScope **extension on Gremlin.**

In line 8-17 of Figure 2, we illustrate a vertex-centric algorithm (as a `process` step) for label propagation (LPA) composed via the extended operators. In this algorithm, a `$lpa_score` initialized by known labels will be assigned to each vertex. The extended step `property(K,V)` (line 9,13,15) creates or updates a specified *runtime* property beginning with the "`$`" identifier. Inspired by Flink[1], we introduce `scatter()` and `gather()` steps to simulate the message passing and aggregation in a vertex-centric algorithm. Specifically, `scatter()` is used to send the specified value to all vertices given by the following `by()` modulator, and `gather()` is for aggregating the received values via an aggregate function and saving the result in a runtime property. In line 12, a built-in `sum` operator is used to sum `$lpa_score` of its incoming neighbors and save the result in `$tmp`; and then `$new_score` can be further calculated. Besides, we also introduce `expr(String)` as a syntactic sugar that users could rely on to make their arithmetic expression or logic expression more succinct. When an iteration finishes, the iterative algorithms require "redoing" the computation from a certain set of vertices. While Gremlin does not natively provide such operators, we extend the source operator `V()` within `process` (line 11) to give the traversal machine a hint to *reset the traversal* with all vertices as sources. The computation terminates if no vertex changes its `$lpa_score` (line 16). Finally, selected properties can be added to the graph via a `withProperty` step.

In addition to the vertex-centric model, we also support the graph-centric PIE model as proposed by GRAPE [3] by calling

[1]See https://ci.apache.org/projects/flink/flink-docs-stable/dev/libs/gelly/iterative_graph_processing.html.

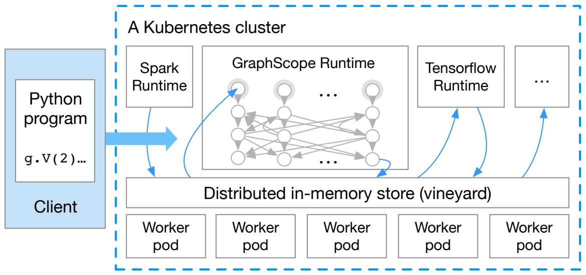


Figure 3: The GraphScope **system architecture.**

a user-defined algorithm in the `process` step. More specifically, developers only need to provide two functions: (1) PEval, a function for partial evaluation: for a given query, it computes its partial answer on a local fragment of the graph, and exchanges the status of those border nodes with other workers via messages. and (2) IncEval, an incremental function that computes changes to the old output by treating messages from other fragments as updates. With this, developers can access a global view of the graph and parallelize exiting single-machine graph algorithms such that it runs across multiple processors and machines, with minor changes.

3 SYSTEM ARCHITECTURE

The system architecture of GraphScope is shown in Figure 3. Its applications may contain multiple types of workloads, each handled by a different distributed framework, *e.g.,* Spark (for relational operations), the GraphScope runtime (for graph algorithms), and TensorFlow (for neural networks computations). All the computations (regardless of what framework) read inputs from and write outputs to a shared distributed in-memory store provided by GraphScope.

GraphScope Execution Overview. The execution flow of a computation in each framework typically contains the following steps.

Step 1. A Python application runs on a client machine, connected to a Kubernetes cluster. When a certain computation is triggered (possibly by a function call that relies on the outputs), the corresponding expression object is handed over to the framework.

Step 2. The framework compiles the expression into a distributed execution plan and generates the codes for running. Because this aspect is similar to [3, 8, 9, 11], we do not elaborate on it further.

Step 3. The plan is submitted to a centralized, framework-specific "job manager" (JM), which orchestrates the execution of a distributed computation on the cluster, and is responsible for jobs such as scheduling resources and distributing the computation across a cluster of containerized workers.

Step 4. Each worker executes a fragment of the distributed computation, and reports progress to the JM via heartbeat signal.

Step 5. When all workers complete, the JM will terminate, and return control back to the user application.

Step 6. The triggered function call returns a local (Python) object encapsulating the outputs of the execution, which can be used as inputs to subsequent expressions in the user program, to be executed following the above Steps 2-5.

GraphScope is a complex system that requires the interactions of a wide variety of systems. We highlight two novel components.

Distributed In-Memory Store (Vineyard**).** GraphScope allows the entire computation (pipeline) to stay in memory. Disk I/O is only needed at the very beginning (to load the input) and at the very end (to write the result). All the intermediate results (possibly across multiple frameworks) are maintained in a shared distributed in-memory store, called Vineyard. Vineyard provides high-level data (PyData) abstractions, such as `Graph`, `Array`, `Tensor`, and `DataFrame`, as the storage interface, which not only simplifies programming, but also allows efficient data exchange between two computations on the same machine. This is achieved by mapping or transforming the metadata of the data objects, which avoids costly copying of their payloads. Given that multiple stages in a data-parallel pipeline typically share a common data partitioning scheme, such a "zero-copy" abstraction significantly improves performance for the overall execution. Vineyard makes it possible to orchestrate a workflow across many systems, while existing works like Spark, require more integration effort for other frameworks.

Dataflow Engine (Gaia**).** GraphScope compiles a wide range of graph computations expressed by (extended) Gremlin into distributed execution plans, and runs them on a new dataflow engine, called Gaia. Unlike existing graph systems built on a dataflow model such as GraphX [4], Gaia [8] employs a number of technologies to enable and combine a set of graph-specific optimizations in one carefully designed coherent framework. Moreover, by integrating with Vineyard (which provides streams between steps), the execution of Gaia can be overlapped with downstream computations (even in a different framework). For example, a pipeline execution from GNN sampling (in Gaia) to training (in TensorFlow).

4 DEMONSTRATION

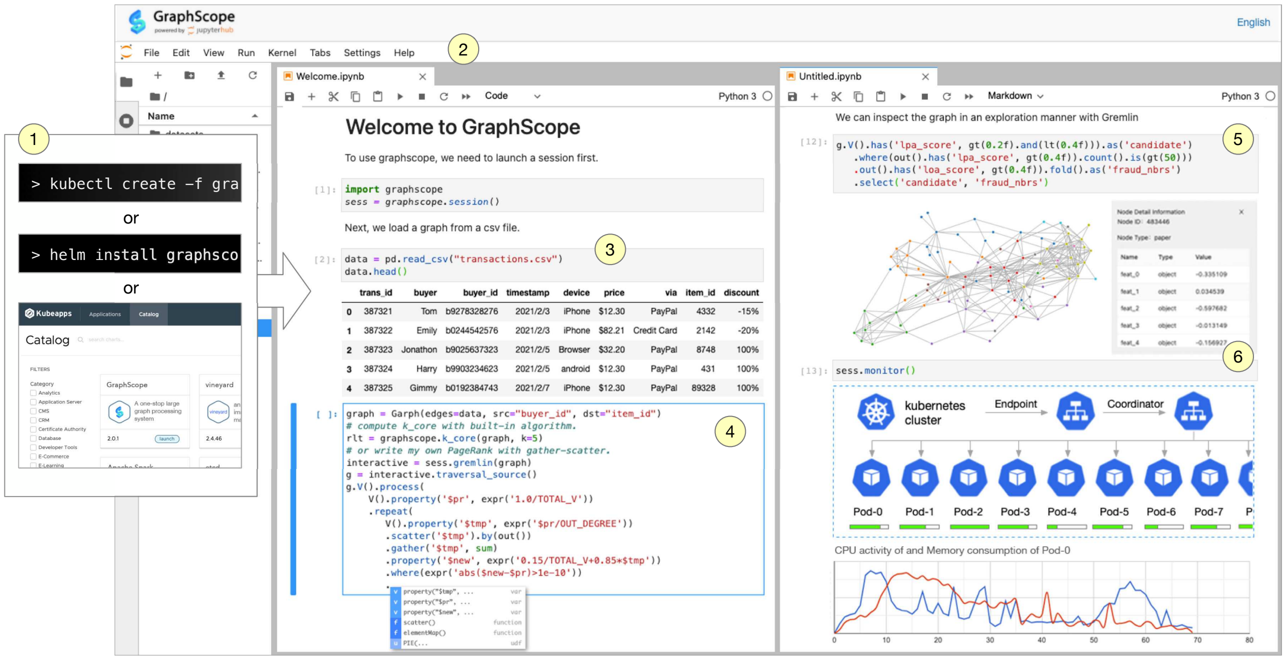
The demonstration consists of two parts. (1) We walk through GraphScope to demonstrate its main features; and (2) we also demonstrate how GraphScope is applied in various scenarios.

A walk through. We visualize and demonstrate how GraphScope processes large-scale graph computation as shown in Figure 4.

(1) Deployment. GraphScope is natively built and deployed on Kubernetes. As displayed in Figure 4(1), GraphScope follows the standard Kubernetes procedure to deploy and launch. Users can use the CLIs provided by `kubectl` or `helm`. Alternatively, GraphScope can be launched with a few clicks in a web-based app manager like `kubeapps`. Users are able to customize the configurations during the deployment, including how many worker pods are used, whether to mount a volume for the data input and output, etc.

(2) Jupyter Notebook interface. Figure 4(2) shows the main interface of GraphScope. It is a managed Jupyter Notebook container that serves as the IDE, with a cluster of containerized workers serving in the background to process queries. Jupyter Notebook is the de-facto standard Python development environment for data scientists. Thus writing code to process large graphs is easy in GraphScope, since all the graph computing abilities provided by GraphScope can be easily accessed directly within the Notebook.

(3) Loading a graph. GraphScope supports property graphs in which the edges/vertices are labelled and have many properties. With Vineyard, GraphScope can load graph from various sources,



```
> kubectl create -f gra
```

```
> helm install graphsco
```

Welcome to GraphScope

To use graphscope, we need to launch a session first.

```
[1]: import graphscope
     sess = graphscope.session()
```

Next, we load a graph from a csv file.

```
[2]: data = pd.read_csv("transactions.csv")
     data.head()
```

	trans_id	buyer	buyer_id	timestamp	device	price	via	item_id	discount
0	387321	Tom	b9278328276	2021/2/3	iPhone	$12.30	PayPal	4332	-15%
1	387322	Emily	b0244542576	2021/2/3	iPhone	$82.21	Credit Card	2142	-20%
2	387323	Jonathon	b9025637323	2021/2/5	Browser	$32.20	PayPal	8748	100%
3	387324	Harry	b9903234623	2021/2/5	android	$12.30	PayPal	431	100%
4	387325	Gimmy	b0192384743	2021/2/7	iPhone	$12.30	PayPal	89328	100%

```
[ ]: graph = Garph(edges=data, src="buyer_id", dst="item_id")
     # compute k_core with built-in algorithm.
     rlt = graphscope.k_core(graph, k=5)
     # or write my own PageRank with gather-scatter.
     interactive = sess.gremlin(graph)
     g = interactive.traversal_source()
     g.V().process(
         V().property('$pr', expr('1.0/TOTAL_V'))
         .repeat(
            V().property('$tmp', expr('$pr/OUT_DEGREE'))
            .scatter('$tmp').by(out())
            .gather('$tmp', sum)
            .property('$new', expr('0.15/TOTAL_V+0.85*$tmp'))
            .where(expr('abs($new-$pr)>1e-10'))
            .
            v property("$tmp", ...     var
            v property("$pr", ...      var
            v property("$new", ...     var
            f scatter()                function
            f elementMap()             function
            u PIE(...                  udf
```

We can inspect the graph in an exploration manner with Gremlin

```
[12]: g.V().has('lpa_score', gt(0.2f).and(lt(0.4f))).as('candidate')
        .where(out().has('lpa_score', gt(0.4f)).count().is(gt(50)))
        .out().has('loa_score', gt(0.4f)).fold().as('fraud_nbrs')
        .select('candidate', 'fraud_nbrs')
```

Node Detail Information
Node ID: 483446
Node Type: paper

Name	Type	Value
feat_0	object	-0.335109
feat_1	object	0.034539
feat_2	object	-0.597682
feat_3	object	-0.013149
feat_4	object	-0.156927

```
[13]: sess.monitor()
```

Figure 4: Demonstration overview

e.g., local file systems, NFS, Amazon S3 and Aliyun OSS, etc. Figure 4(3) shows that graph data in a dataframe can be generated from other PyData libraries and loaded in parallel via Vineyard. Then the graph data is managed in Vineyard across the cluster.

(4) Programming in Gremlin and Python. GraphScope extends Gremlin and provides a unified programming interface in Python. In addition, GraphScope embodies a set of built-in algorithms for graph analytics and graph learning, enabling users to easily process their graph tasks. Better still, GraphScope supports "plug-and-play" sequential algorithms following PIE model or vertex-centric algorithms in GAS model. Figure 4(4) demonstrates how to use a built-in algorithm and compose algorithm with code completion.

(5) Interactive queries with visualization. For interactive queries, GraphScope supports analyzing large-scale graphs in an exploratory manner. As shown in Figure 4(5), interactive queries expand a panel in the notebook with visualization of the result graph. Users can further inspect the properties of the vertices/edges.

(6) System monitor and performance report. The audience is invited to customize the configurations of GraphScope, and observe its scalability by varying the number of worker pods, datasets and computation tasks. As shown in Figure 4(6), users can easily monitor the status of the cluster, including the real-time progress, the resources usage, the performance and communication of each worker, etc.

Scenarios. In addition to the fraud-detection task in Section 1, GraphScope can be applied to various graph-related tasks. Next, we briefly demonstrate 3 scenarios based on real-life applications.

(1) Cybersecurity monitoring. Trojans are often controlled by their holders with malicious domains and IPs. We build a bipartite graph where domains and IPs are treated as vertices, and there is an edge (u, v) if domain u is resolved from IP v. Given a known malicious domain, the resolved IPs are potentially malicious, as they are controlled with high possibility by the Trojan holders, and vice versa. This can be easily expressed in Gremlin supported by GraphScope.

(2) Recommendation in e-commerce. Recommendation services are developed to predict a user's interest for certain items. Given a transaction graph like in Figure 1, we first conduct graph algorithms like common neighbors to compute the potential links, and then run a GNN model to learn the representation for each user and item. Finally some items are recommended to a target user by assessing the representation similarities of user-item pairs.

(3) Node classification on citation graph. This task aims to predict the research topics of some unlabeled papers in a citation graph. We first write in Gremlin to extract a sub-graph with entities satisfying certain conditions. We then conduct iterative algorithms (*e.g.,* k-core and triangle counting) to generate the structural feature for each node. Combining both structural and semantic features for paper node, we finally run a GNN model to classify the paper topics.

REFERENCES

[1] Gremlin Apache Tinkerpop. 2015. https://tinkerpop.apache.org/gremlin.html

[2] Dask Development Team. 2016. https://dask.org

[3] Wenfei Fan, Jingbo Xu, Yinghui Wu, Wenyuan Yu, Jiaxin Jiang, Zeyu Zheng, Bohan Zhang, Yang Cao, and Chao Tian. 2017. Parallelizing Sequential Graph Computations. In *Proceedings of the 2017 ACM SIGMOD*. 495–510.

[4] Apache Spark GraphX. 2014. https://spark.apache.org/graphx/

[5] Chen Liang, Ziqi Liu, Bin Liu, Jun Zhou, Xiaolong Li, Shuang Yang, and Yuan Qi. 2019. Uncovering insurance fraud conspiracy with network learning. In *Proceedings of the 42nd International ACM SIGIR*. 1181–1184.

[6] 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*. 135–146.

[7] Koalas Project. 2020. https://github.com/databricks/koalas

[8] 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 *NSDI 21*. USENIX Association.

[9] Google Tensorflow. 2016. http://tensorflow.org

[10] Haobo Wang, Zhao Li, Jiaming Huang, Pengrui Hui, Weiwei Liu, Tianlei Hu, and Gang Chen. 2020. Collaboration based multi-label propagation for fraud detection. In *IJCAI*.

[11] Rong Zhu, Kun Zhao, Hongxia Yang, Wei Lin, Chang Zhou, Baole Ai, Yong Li, and Jingren Zhou. 2019. AliGraph: a comprehensive graph neural network platform. *Proceedings of the VLDB Endowment* 12, 12 (2019), 2094–2105.