PSGraph: How Tencent trains extremely large-scale graphs with Spark?

Jiawei Jiang
Department of Computer Science
ETH Zürich
Zürich, Swizterland
jiawei.jiang@inf.ethz.ch

Pin Xiao
Data Platform, TEG
Tencent Inc.
Shenzhen, China
payniexiao@tencent.com

Lele Yu
Data Platform, TEG
Tencent Inc.
Shenzhen, China
leleyu@tencent.com

Xiaosen Li
Data Platform, TEG
Tencent Inc.
Shenzhen, China
hansenli@tencent.com

Jiefeng Cheng
Data Platform, TEG
Tencent Inc.
Shenzhen, China
geoffcheng@tencent.com

Xupeng Miao
School of EECS & MOE
Peking University
Beijing, China
xupeng.miao@pku.edu.cn

Zhipeng Zhang
School of EECS & MOE
Peking University
Beijing, China
zhangzhipeng@pku.edu.cn

Bin Cui
School of EECS & MOE
Peking University
Beijing, China
bin.cui@pku.edu.cn

***Abstract*—Spark has extensively used in many applications of Tencent, due to its easy deployment, pipeline capability, and close integration with the Hadoop ecosystem. As the graph computing engine of Spark, GraphX is also widely deployed to process large-scale graph data in Tencent. However, when the size of the graph data is up to billion-scale, GraphX encounters serious performance degradation. Worse, Graphx cannot support the rising advancement of graph embedding (GE) and graph neural network (GNN) algorithms. To address these challenges, we develop a new graph processing system, called PSGraph, which uses Spark executor and PyTorch to perform calculation, and develops a distributed parameter server to store frequently accessed models. PSGraph can train extremely large-scale graph data in Tencent with the parameter server architecture, and enable the training of GE and GNN algorithms. Moreover, PSGraph still benefits from the advantages of Spark via staying inside the Spark ecosystem, and can directly replace GraphX without modification to the existing application framework. Our experiments show that PSGraph outperforms GraphX significantly.**

***Index Terms*—graph algorithm, Spark, parameter server**

I. Introduction

Spark is a unified analytics engine for big data processing and is widely used in the industry, including Tencent, owing to its easy-to-use programming abstraction, deep integration with big data ecosystem, and useful pipeline infrastructure. GraphX, the graph processing module of Spark, is developed to process graph data, which are common in the real-word scenarios, e.g., social graph in social networks, citation graph in academia community, user interest graph in the e-commerce market, and knowledge graph. GraphX is certainly extensively used in many graph datasets of Tencent, for instance, a social network graph composed of more than one billion people and hundreds of billions of friendship connections. GraphX is chosen rather than other graph processing systems, such as Pregel [1], GraphLab [2] and Gemini [3], owing to several reasons. First, GraphX is compatible with the widely used MapReduce/Hadoop infrastructure since the underlying computation engine is Spark. As the de-facto infrastructure for big data processing, Hadoop has achieved great success in the past decade. GraphX is fundamentally compatible with the components in the Hadoop ecosystem, such as HDFS, Yarn, Hive, HBase, and Flume, because the underlying engine Spark is originally developed to replace the computation engine of Hadoop. Consequently, it is easy to deploy GraphX in real industrial Hadoop/MapReduce environments. Second, many applications in Tencent are developed with Spark and running for years. These applications usually use the pipeline capability of Spark to embrace different processing phases (including data ingest, data preprocessing, feature engineering, model training, and model evaluation) in a dataflow task, without moving data in and out of file systems. GraphX can be easily integrated into the existing application by putting it into the Spark pipeline, without interfering or reconstruction of the existing framework. In contrast, other graph processing systems, such as GraphLab and Giraph, either (1) entail users to exhaustively deploy them to the Hadoop ecosystem, leading to brittle interfaces and extra maintenance costs, or (2) incur expensive data movement through the file systems between different systems when embedding them into a dataflow pipeline.

Although GraphX has been successfully used in Tencent for years because it is suitable for the industrial environment, it reveals drawbacks when facing exponential growing of graph data and the rapid evolution of graph algorithms. One the one hand, when processing graph data with up to billions of vertices and hundreds of billions of edges, GraphX encounters severe performance degradation. The reason is that GraphX stores graph data in a table abstraction, in which every executor (worker) stores an edge table and a vertex table, as well as the features of edges and vertices. With a shared-nothing architecture, GraphX uses the table-join operation of Spark to implement message passing and calculation in graph algorithms. The join operation of Spark causes a large memory cost to store massive temporary data, and yields costly shuffle operation between the map task and the reduce task, which

2375-026X/20/$31.00 ©2020 IEEE
DOI 10.1109/ICDE48307.2020.00137

needs to write and read temporary data via the disk. Facing billion-scale graph data, GraphX is swamped by memory explosion and slow disk IO. One the other hand, GraphX cannot support many recently emerging graph algorithms since it is specialized designed for traditional graph algorithms that analyze graph structure and graph topology. Nevertheless, the recent several years have witnessed a booming progress of graph embedding (GE) and graph neural networks (GNN) algorithms. GraphX cannot handle these newly developed algorithms. Motivated by these challenges, we ask *"can we implement a graph system to replace GraphX that can support different kinds of graph algorithms for extremely large-scale graph data while staying inside the Spark ecosystem?"*

The first problem is how to resolve the poor performance of GraphX on large-scale graph data. As above analyzed, GraphX performs poorly owing to slow shuffle operation during table join. To address this problem, an alternative is to store the frequently accessed models in a shared in-memory storage system, and let workers read and update the models through the network. Motivated as such, we empower Spark with a parameter server (PS) [4] that partitions models over several machines and stores each partition in memory. The second problem is how to support various kinds of graph algorithms in a unified system. For traditional graph and graph embedding algorithms, the calculation patterns are generally simple so that we leverage Spark executor to perform the calculation. However, graph neural networks involve complex calculations such as matrix multiplication, gradient computation, and backward propagation. To ensure usability for the existing deep learning users, we embed PyTorch, a prevailing deep learning system, into Spark. In this way, the users can benefit from the easy-to-use programming interface of PyTorch, and obtain the distributed training capability meanwhile. The contributions of this paper are summarized as follows:

- We develop a unified graph processing system, called PSGraph, to replace GraphX in industrial applications of Tencent. [1] PSGraph can concurrently train traditional graph algorithms, graph embedding algorithms and graph neural networks. PSGraph can be easily integrated into the industrial environment and Spark pipeline.
- PSGraph empowers Spark with a parameter server to efficiently train billion-scale graph data and integrates PyTorch into Spark to implement graph neural networks.
- We describe the use cases of PSGraph in Tencent. Experiments show that PSGraph is significantly faster than the baselines.

II. Preliminaries

A. Graph Data

The input of a graph algorithm is denoted by (V, E) where V is the vertex set and E is the edge set. Each vertex in V represents an entity such as user; and each edge in E represents relations between entities, such as friendship.

[1] https://github.com/Angel-ML/sona, https://github.com/Angel-ML/PyTorch-On-Angel

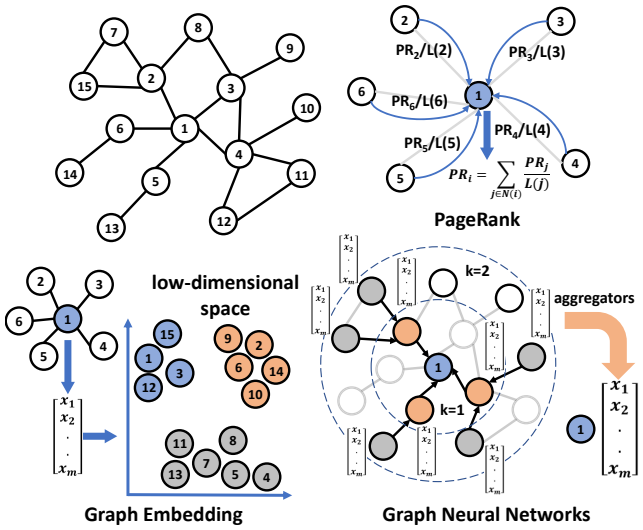

Fig. 1. Graph algorithms.

Each vertex can be given some latent properties, for instance, the PageRank value indicates the importance of vertex in the graph while the coreness value can help identify small interlinked core areas in a graph. Typically, graph algorithms, ranging from PageRank to Graph Neural Network, iteratively update the property of a vertex based on the properties of its adjacent vertexes and edges.

B. Graph Algorithms

According to different purposes and computation patterns, the existing graph algorithms can be roughly classified into three categories, as shown in Fig. 1.

- *Traditional Graph (TG) Algorithms* aim at analyzing the topological structure of the graph. For example, PageRank [5] measures the importance of graph vertices, K-core [6] [7] measures the connecting density of subgraphs, common neighbor measures the similarity of two vertices, and label propagation [8] detects densely connected community.
- *Graph Embedding (GE) Algorithms* provide an effective, yet efficient, way to deal with graph analytics problems, e.g., vertex classification and graph classification [9] [10]. GE generates low-dimensional vectors to represent the vertices or the entire graph to reduce computation and communication costs. With these representations, directly processing the original graph, bringing high computation, communication and space cost, can be avoided. Specifically, vertex embedding [11] [12] [13] encodes each vertex with a vector representation to perform predictions on the vertex level. e.g., prediction of new edges based on vertex similarities, while graph embedding [14] represents the whole graph

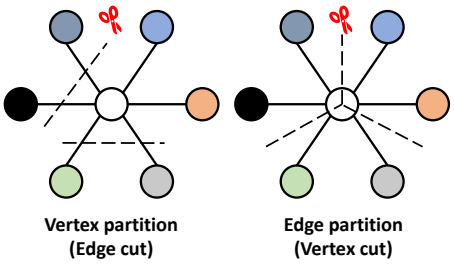

Fig. 2. Partitioning strategies.

with a single vector to make predictions on the graph level, e.g. comparison of chemical structures.

- *Graph Neural Network (GNN) Algorithms* learn representations using deep learning methods. GNN can be categorized into recurrent GNN [15], convolutional GNN [16], graph autoencoders [17], and spatial-temporal GNN [18]. Typically, GraphSage samples k-hop neighbors of the target vertex, collects their representation vectors, calculates the output with some aggregating function, and updates the current representation vector.

C. Programming Models

An important problem in graph algorithms is how to program calculation given the complexity of graph structure. There are three popular graph programming models according to different views of the graph.

- *Vertex-centric.* The *vertex program* running on each vertex gets the properties of adjacent vertices and edges, with which its own vertex property is updated. This process iterates until reaching some stopping criteria.
- *Partition-centric.* Partition-centric computation is to operate on a partition (subgraph). And the purpose is to reduce communication cost among partitions using effective graph partition approaches.
- *Edge-centric.* Using an edge-centric perspective, the scatter and gather phases process on edges rather than on vertices [19]. This is suitable for cases where sequentially reading edges from storage media is much faster than random accessing.

D. Distributed Graph Algorithms

When the size of graph data overwhelms the computation and storage capabilities of a single machine, it is inevitable to deploy graph algorithms in the distributed environments, bringing the arise of partition approaches and synchronization techniques.

- *Partitioning.* Previous studies on parallel graph algorithms have focused on a vertex partitioning (edge cut) or an edge partitioning (vertex cut), as Figure II-D shows. With vertex partitioning, the graph is cut by edges, and each worker is assigned a vertex subset with their adjacent edges, i.e., neighbor tables. While with edge partitioning, each worker is assigned an edge set [20].
- *Synchronization.* 1) Bulk synchronous parallel (BSP). At each iteration, each worker performs computation with its graph partition and pushes the results to other workers at a synchronization barrier [1]. 2) Asynchronous parallel (ASP). An alternative to BSP is asynchronous execution without any synchronization barrier [4]. 3) Gather-apply-scatter (GAS). GAS is a pull-based counterpart of BSP [21]. Each worker pulls states from other workers (gather), performs computation (apply), and updates states (scatter).

III. System Overview

Fig.3 shows the framework of PSGraph, including the parameter server, the computation engine, and the master.

A. Parameter Server

PSGraph leverages a parameter server (PS) to store high-dimensional data or models in graph algorithms. The following shows the basics of the parameter server.

- *Data structure.* PS supports different data structures, e.g., sparse/dense vector, sparse/dense matrix, CSR, vertex (with property), and neighbor table. PS also provides interfaces for users to implement a new data structure.
- *Data partitioning.* The graph data frequently accessed are partitioned over several machines. For vectors and matrices, PS partitions them by row index and column index. For graph vertex and neighbor table, PS partitions them by vertex index. We implement hash partition, range partition, and hash-range partition [22].
- *Synchronization.* PS has different synchronization protocols (BSP/ASP) to control the synchronization across workers.
- *Data Operators.* Many common operators are implemented to manipulate the data on PS, such as pull, push, addition, division, etc. Also, users can customize their operators via a user-defined function, called `psFunc`.
- *Checkpoint.* Each parameter server periodically stores the local data partition to HDFS. As we will explain later, we address failure by this checkpoint mechanism.

Specifically, for graph algorithms, we can store different kinds of data on PS, according to the patterns of data access and model update. If the algorithm is to calculate some latent properties for vertices, the vertex properties are stored on the PS as they are accessed and updated at every iteration. If the algorithm needs to get the adjacent vertices of a vertex frequently, the neighbor tables are stored on the PS.

B. Master

The master node of PS is responsible for resource allocation, task monitoring and failure recovery. When a task is submitted to the resource management platform such as

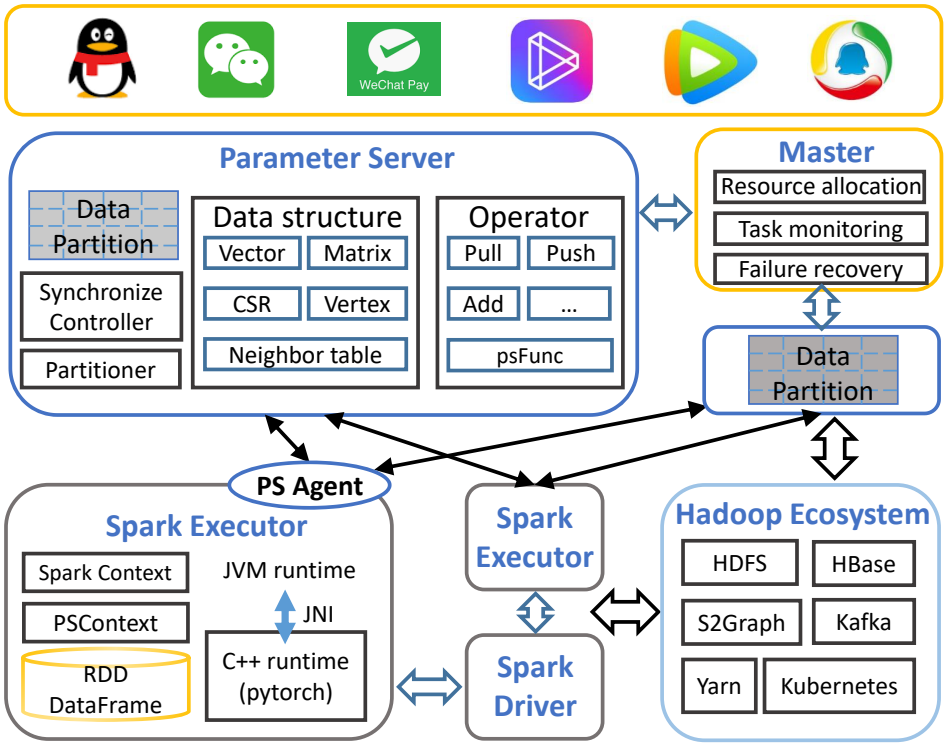

Fig. 3. System overview of PSGraph.

Yarn and Kubernetes, the master is first initialized. It then requests resources from the resource management platform to launch the parameter servers. During the execution, the master monitors the status of servers by periodical sending health checking signal. Once one server encounters failure, the master asks the resource management platform to restart the server. If the algorithm can bear inconsistency between model partitions, such as GE and GNN, the newly launched server pulls the checkpoint partition from HDFS (refer to Section III-A) and continues training. Otherwise, the master asks all the servers to restore the checkpoint partitions from HDFS, such that model consistency is ensured for algorithms such as PageRank.

C. Computation Engine

The computation engine of PSGraph is implemented with Spark. Below, we show how the computation engine stores data, calculates and communicates with PS.

- *Context.* Spark has a context shared by all the executors, called *SparkContext*. PSGraph uses it to get Spark settings and runtime statistics. Besides, PSGraph creates a context called *PSContext* to store the configurations of PS, such as the locations of parameter servers and the partition layout (mapping of data partitions to servers).
- *PS agent.* PSGraph establishes a PS agent in every Spark executor to manage the data communication between Spark and PS. When the PS agent needs to get a data item from the PS, it first uses the data index to get the partition location from *PSContext*, including the partition index and the location of the corresponding server. Then, the PS agent gets the required data from PS via RPC (remote process call). Similarly, when the executor needs to update the data, the PS agent first locates the corresponding partition and server, and then sends the data update to PS.
- *Data abstraction.* The graph data are constructed as RDD, Dataframe or Dataset in executors [23]. Resilient distributed dataset (RDD), the core programming abstraction of Spark, is a fault-tolerant collection of elements that can be operated in parallel. Dataframe and Dataset extend RDD with relational schema, enabling SQL query and pipeline execution. PSGraph supports both edge partitioning and vertex partitioning, meaning that the element in RDD can be edge or neighbor table.
- *JVM runtime.* Spark runs in JVM (Java virtual machine). Since we implement TG and GE algorithms with Scala and Java, they naturally run in JVM runtime.
- *C++ runtime.* To run GNN algorithms, PSGraph embeds PyTorch inside Spark. However, there is a language gap between the JVM runtime of Spark and the C++ runtime of PyTorch. We transfer data between JVM runtime and C++ runtime using JNI (Java native interface) — 1) graph data is fed into PyTorch, 2) PyTorch performs forward calculation and backward propagation with Autograd mechanism, and

3) send gradients to JVM runtime.

- *Failure recovery.* Since Spark has a failure recovery mechanism, PSGraph utilizes it to address the failure of computation engine. Once an executor fails, it will be restarted by Spark. Afterward, it reloads graph data from HDFS and continues training. During the period, the other executors are blocked by the synchronization controller of PS.

D. Programming Interface

```
class GraphRunner {

    def main(args: Array[String]): Unit = {
        val params = ArgsUtil.parse(args)
        SparkContext.getOrCreate()
        PSContext.getOrCreate()
        val algo = new GraphAlgo(params)
        val graph = GraphIO.load(params)
        val output = algo.transform(graph)
        GraphIO.save(output)
    }

}

class GraphAlgo(args: Array[String]) {

    def transform(dataset: Dataset[_]): DataFrame = {
        val edges = GraphOps.loadEdges(dataset)
        val neighborTable = GraphOps.toNeighborTable(edges)
        val model = PSContext.matrix(row, col, DataType)
        val delta = ...     // do calculation
        model.update(delta)
        SparkContext.createDataFrame(model)
    }

}
```

Listing 1. Example Code

The above code showcases the programming of PSGraph. In *GraphRunner*, we create an instance of *GraphAlgo*, load data from the data source, run the algorithm and save the generated model. In *GraphAlgo*, we transform the original graph data to edges or neighbor tables, use *PSContext* to create a model on PS (given the number of rows, number of columns and data type), generate updates, and sends the updates to PS.

IV. Implementation of Graph Algorithms

In this section, we describe how to implement graph algorithms with PSGraph, e.g., PageRank, common neighbor, fast unfolding, Line, and GraphSage. These algorithms are widely used in many applications of Tencent, such as WeChat, WeChat Pay, and QQ. We assume the original dataset is stored on HDFS, and each data item is a pair (src, dst) where src is the index of source vertex and dst is the index of destination vertex. The vertex indices are encoded as *long int*.

A. PageRank

PageRank measures the importance of vertices. The update rule is $PR_i = \sum_{j \in N(i)} \frac{PR_j}{L(j)}$, where PR_i denotes the rank of vertex i, $N(i)$ are the neighbor vertices of vertex i, and $L(j)$ is the out-degree of vertex j. An optimization of this update rule is to use the increments of ranks instead of the ranks. Since the ranks of many vertices barely change after several iterations, we leverage this sparsity to reduce the communication cost by transferring the increments of ranks. The optimized update rule is $\Delta PR_i = \sum_{j \in N(i)} \frac{\Delta PR_j}{L(j)}$.

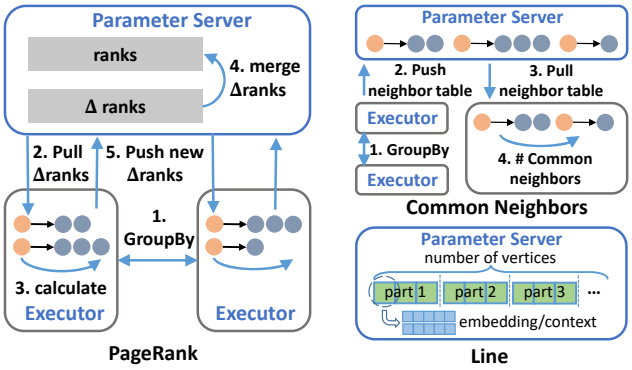

Fig. 4. Implementation of PageRank, Common Neighbor and Line.

Fig. 4 shows the implementation of PageRank. The PS stores two vectors: 1) the rank $\vec{ranks}$ and 2) the increment of rank $\Delta\vec{ranks}$. The size of both vectors is equal to the maximal index of vertex. We use Spark to load the original dataset from HDFS, and store the dataset as an RDD in which each item is an edge. In a distributed cluster, edge partitioning (vertex cut) yields a high communication overhead as many executors need to get the ranks of one vertex concurrently. We then use the `groupBy` operator to transform the original edge-partitioned graph data to the format of vertex partitioning, that is, each item in RDD is a neighbor table — $(src, Array[dst])$. After this operation, the edge RDD is transformed into a RDD consisting of neighbor tables. The calculation procedure is as follows:

1) The executors run the `groupBy` operator to transform the edge-partitioned graph data to vertex partitioning.
2) The executor gets the rank increments, $\Delta ranks$, of all the local source vertices from PS.
3) The executor uses $\Delta ranks$ to calculate the updates of the destination vertices.
4) PS adds $\Delta ranks$ to $ranks$ and resets $\Delta ranks$ to zero.
5) The executor pushes the local updates of destination vertices to PS. PS adds them to $\Delta ranks$.

B. Common Neighbor

Common neighbor helps measure the closeness of two vertices and is used for link prediction. This algorithm requires frequent access to the adjacent vertices (neighbors) of a vertex. We hence store the neighbor tables on PS. This is achieved by first transforming the original graph data to neighbor tables by `groupBy` operator of Spark and then pushing the neighbor tables to PS. Afterward, the executor iteratively processes a batch of edges, gets the neighbor tables of the vertices from PS, and calculates the number of overlapping neighbors of each vertex pair.

C. Fast Unfolding

Fast unfolding can extract the community structure of large networks based on a metric called modularity [8]. Note that the input is a weighted graph, that is, each edge is a triplet $(src, dst, weight)$. Each pass of fast unfolding is made of two phases: 1) modularity optimization: assign a community for each vertex to maximize the modularity; 2) community aggregation: build a new network whose vertices are the communities found during the first phase. The passes are iterated until no increase of modularity.

The following formula is the change of modularity when adding vertex i to community C:

$$\Delta Q = [\frac{\sum_{in}+k_{i,in}}{2m} - (\frac{\sum_{tot}+k_i}{2m})^2] - [\frac{\sum_{in}}{2m} - (\frac{\sum_{tot}}{2m})^2 - (\frac{k_i}{2m})^2\ ]$$

where $\sum_{in}$ is the sum of weights of edges inside the community, $\sum_{tot}$ is the sum of weights of edges ected to vertices in the community, k_i is the sum of the weights of the links associated with node i, $k_{i,in}$ is the sum of the weights of the links from i to nodes in C, and m is the sum of the weights of all the links in the network.

In fast unfolding, two models are frequently accessed, i.e., the community of each vertex and the sum of edge weights in each community. To implement fast unfolding in PSGraph, we store these two models as *vertex2com* and *com2weight* on the PS. The calculation consists of the following steps: 1) Spark loads the dataset from HDFS; 2) Spark transforms the original edge-partitioned RDD to vertex partitioning. Then, each item in the transformed RDD is a triplet $(src, arr[dst], arr[weight])$; 3) Each vertex is given an initial community index, identical to the vertex index. Afterward, each executor calculates the local *vertex2com* and *com2weight*, and pushes them to PS; 4) At each iteration, the executor pulls the current *vertex2com* and *com2weight* from PS, performs the modularity optimization and community aggregation, and pushes the new communities of vertices and the new sum of edge weights of communities to PS.

D. Line

Line is a GE algorithm that uses both first-order proximity and second-order proximity to measure the similarity of two vertices. The first-order proximity refers to the local proximity between the vertices in the network. The second-order proximity assumes that vertices sharing many connections to other vertices are similar to each other. Each vertex has two latent vectors — an embedding vector itself and a context vector when the vertex is a "context" of other vertices. These vectors can be extremely huge for a large-scale graph. For example, when the number of vertices is up to one billion and the embedding size is ten, the size of latent vectors is larger than 80GB. Using one machine to store the latent vectors could cause serious network congestion because all the executors need to get and update these vectors. Therefore, we store these vectors on PS to address this problem. These vectors are formatted as a PS vector, where the size is the number of vertices and the data type is a user-defined type containing the embedding vector and the context vector of one vertex.

During one training iteration, the executor gets a batch of edges, pulls the necessary embedding vectors and context vectors from PS, uses stochastic gradient descent to update these vectors and pushes back the updates to PS.

However, letting every executor directly pull embedding vectors from PS is still communication-intensive. We try to leverage the `psFunc` of PS to perform some computation on PS. The training of Line involves a lot of dot product operations between vectors — the first-order proximity calculates the dot product of two embedding vectors: $p_1(v_i, v_j) = \frac{1}{1+exp(-\vec{u_i}^T \cdot \vec{u_j})}$, the second-order proximity calculates the dot product of embedding vector and context vector: $p_2(v_j|v_i) = exp(\vec{c_j}^T \vec{u_i}) / \sum_{k=1}^{|V|} exp(\vec{c_k}^T \vec{u_i})$. To enable the doc product operation on PS, we partition the embedding vectors and context vectors by column, as shown in Figure 4. In this way, the same dimensions of $\vec{u}$ and $\vec{c}$ are co-located on the same server, so that we can calculate partial dot products on PS and merge them on the executor.

During one iteration of training, the executor gets a batch of edges, pulls the necessary dot products from PS, uses stochastic gradient descent to update the embedding vectors and context vectors, and pushes the updates to the PS.

E. GraphSage

GraphSage is a GNN algorithm that learns the representations for vertices. Instead of training individual embeddings for each vertex, GraphSage learns a function that generates embeddings by sampling and aggregating features from a vertex's neighborhoods. The following describes the generation of embeddings where the entire graph $G = (V, E)$ and features for all nodes $X = \{x_v, \forall v \in V\}$ are provided as inputs.

1) Sample a fixed-size of K-hop neighbors of a vertex $v \in V$.
2) The representations of the vertices when k = 0 (base case) are defined as the input features. The following k-th step uses representations generated at the previous $k-1$-th step.
3) At iteration k, each vertex v aggregates the representations of its neighborhood vertices $\{h_u^{k-1}, \forall u \in N(v)\}$ into a single vector $h_{N(v)}^{k-1} \leftarrow Aggregate(\{h_u^{k-1}, \forall u \in N(v)\})$. The aggregation function can be done by a variety of aggregator architectures, such as mean aggregator, LSTM aggregator, and pooling aggregator.
4) After aggregating the neighboring representations, GraphSage concatenates the vertex's current representation h_v^{k-1} with the aggregated neighborhood vector $h_{N(v)}^{k-1}$. This concatenated vector is fed to a fully connected layer with nonlinear activation function σ, which outputs the new representation $h_v^k \leftarrow \sigma(W^k \cdot Concat(h_v^{k-1}, h_{N(v)}^k))$.
5) The final output representations at iteration K as $z_u = h_v^K$ are the embedding of vertices.

As the training procedure shows, the executors need to frequently get three models — the vertex features X, the

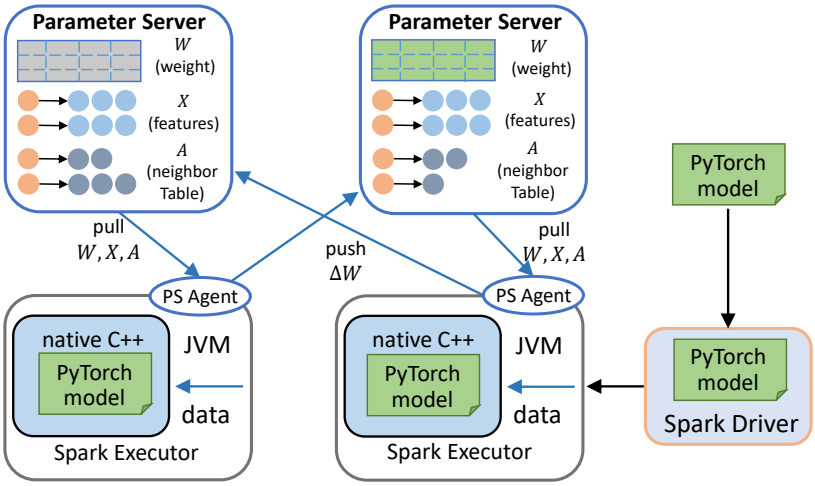

Fig. 5. Implementation of GraphSage.

neighbor table A, and the weight matrix W^k. We store these models on PS — X and A are partitioned by the index of vertices, and W^k is partitioned by column. Fig. 5 shows the training steps of GraphSage: 1) The user writes PyTorch script and generates PyTorch model. 2) Spark driver loads PyTorch model and pushes the initialized model to PS. 3) Every executor loads PyTorch model, reads the dataset from HDFS, uses `groupBy` to generate neighbor tables, and pushes graph features and neighbor tables to PS. 4) At each iteration, the executor pulls the current weight matrix W^k from PS, reads a batch of edges, samples 2-hop neighboring vertices, gets the features of these vertices from PS, performs back-propagation using PyTorch, and pushes the gradients to PS. 5) After training, the system outputs the embedding of all the vertices. Note that, PSGraph uses JNI to let Spark executor feed graph data into the native C++ runtime of PyTorch and let PyTorch send back gradients to JVM runtime. Further, we implement more advanced gradient descent optimizers on PS, such as AdaGrad and Adam, using the user-defined function `psFunc` provided by PS. With our implementation, the user can directly write python scripts of PyTorch and do not need to care about the details of distributed training.

V. Evaluation

To assess the performance of PSGraph, we conduct extensive experiments on large-scale graph datasets.

A. Experimental Setup

a) Datasets: Three real datasets in Tencent are used for the experiments. The first dataset *DS1* contains 0.8 billion vertices and 11 billion edges. The second dataset *DS2* contains 2 billion vertices and 140 billion edges. The third dataset *DS3* contains 30 million vertices and 100 million edges.

b) Algorithms: We evaluate seven algorithms — PageRank, common neighbor, fast unfolding, K-core [24], triangle count[2], Line, and GraphSage.

c) Cluster: The experiments are run on a productive cluster in Tencent. The cluster contains more than 1000 machines, connected by 10GB Ethernet.

d) BaseLine: We compare PSGraph with GraphX on TG algorithms, and Euler[3] on GNN algorithms.

B. Performance Comparison

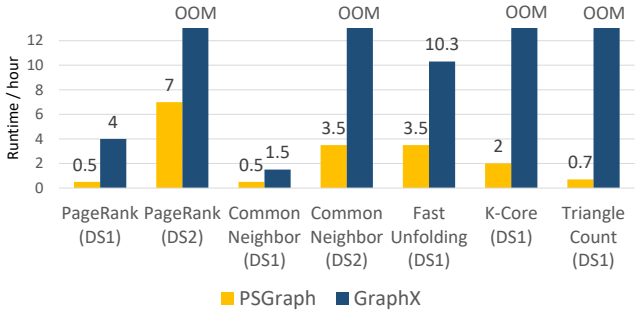

Fig. 6. Performance comparison on traditional graph algorithms.

1) Traditional Graph Algorithms: PageRank, common neighbor, fast unfolding, K-core and triangle count are evaluated on the DS1 dataset. For PSGraph, we allocate 100 executors (20GB) and 20 parameter servers (15GB) for all the algorithms. For GraphX, we allocate 100 executors (55GB). GraphX needs 4 hours to converge on PageRank, while PSGraph only needs 0.5 hours, yielding an 8× improvement. PSGraph benefits from the PS which accelerates the access

[2]The implementation of K-core is similar to PageRank, and that of triangle count is similar to common neighbor.

[3]A graph system developed by Alibaba. (https://github.com/alibaba/euler)

TABLE I
PERFORMANCE ON GRAPHSAGE.

System	Preprocessing time	Training time	Accuracy
Euler	8 hours	200 seconds/epoch	91.5%
PSGraph	12 minutes	7 seconds/epoch	91.6%

and update of models, while GraphX suffers serious memory explosion and slow data shuffling. Moreover, PSGraph only needs half of the resources consumed by GraphX. For common neighbor and fast unfolding, PSGraph is 3× and 2.9× faster than GraphX. For K-core and triangle count, GraphX fails due to an OOM (out of memory) error even giving 55 GB for each executor, while PSGraph needs 2 and 0.7 hours respectively.

PageRank and common neighbor are evaluated on the DS2 dataset, which is much larger than DS1. We allocate 300 executors (30GB) and 200 parameter servers (30GB) for PSGraph, and allocate 500 executors (55GB) for GraphX. GraphX encounters an OOM error even if we give 55 GB for each executor, while PSGraph runs the algorithms in 7 and 3.5 hours with only half of the resources. This result verifies that PSGraph can scale to an extremely large-scale graph and is more resource-efficient.

2) Graph Embedding Algorithms: Although there exist systems that can run Line[4] on a single node, there is rare open-source distributed graph embedding system than can run Line in a productive environment. Therefore, we report the performance of PSGraph for reference. On the DS1 dataset using an embedding size of 128 and the same resources as TG, PSGraph takes 40 minutes per epoch and 4 hours in total.

3) Graph Neural Network Algorithms: We compare PSGraph and Euler on the DS3 dataset and GraphSage. Euler is allocated with 90 executors (16 cores, 50GB), and PSGraph is allocated with 30 executors and 30 parameter servers, each of which is equipped with 10 cores and 10GB.

Note that, Euler has a strict constraint on the graph data so that the original graph data needs complex preprocessing. These operations are executed sequentially and individually, meaning that every operation needs to read data from disk and write output to disk. Unsurprisingly, as Table I illustrates, Euler takes about 8 hours to transform the graph data to the required data format — 4 hours for index mapping, 4 hours for data-to-JSON transformation, and several minutes for JSON partitioning. During the training, Euler takes 200 seconds every epoch if using two-hop neighbors ($k = 2$). In contrast, PSGraph only takes 12 minutes to preprocess the graph data with the help of the efficient Spark pipeline mechanism and consumes 7 seconds every epoch when $k = 2$ — almost 30× faster than Euler. For this real application in WeChat Pay, Euler and PSGraph achieve comparable accuracy.

4) Failure recovery: To assess the effectiveness of failure recovery, we conduct an experiment on common neighbor and DS1 using the same setting in Section V-B1. We manually kill an executor and a parameter server. The killed server will restart and pull the checkpoint of model, i.e., neighbor tables, from HDFS; and the killed executor will restart and pull the checkpoint of edges from HDFS. As shown in Table II, PSGraph can recover quickly (5 minutes for executor failure and 6 minutes for PS failure), and ensure the correctness of the algorithm output meanwhile.

[4]https://github.com/tangjianpku/LINE, https://github.com/snowkylin/line

TABLE II
EVALUATION ON FAILURE RECOVERY.

Algorithm	Without failure	Executor failure	PS failure
Common neighbor	30 minutes	35 minutes	36 minutes

VI. RELATED WORK

In Section II, we have discussed the preliminaries of graph systems. Here we present a summary of prior works.

Single-node graph processing system. Some works tried to process graph algorithms in a multi-core machine, such as GraphChi [25], X-Stream [19] and GridGraph [26]. They designed efficient data access and data partitioning methods on disk to accelerate the processing of graph data. However, when the graph data grow to billion-scale, using only one machine becomes inefficient. Besides, these systems are not suitable for industrial applications that generate and store distributed graph data.

Distributed graph processing system. The exponential increase of graph data aroused the emerging of a series of distributed graph systems. Pregel [1] proposed a push-based vertex-centric approach in which a vertex can receive messages from other vertices and modify its own state and that of its outgoing edges. GraphLab [2] abstracted asynchronous and dynamic computation, and developed pipelined locking and data versioning to reduce network congestion. Gemini [3] extended the hybrid push-pull computation model from shared-memory to distributed scenarios. GraphX [27], built on top of Apache Spark, represented graphs as horizontally-partitioned collections and graph computation as dataflow operators on those collections. PowerGraph [21] introduced a new approach to distributed graph placement and representation that exploits the structure of power-law graphs.

VII. CONCLUSION

We develop a graph processing system for Tencent's dataflow applications. PSGraph can scale to billion-scale graph data by implementing a parameter server to manage graph models. Further, PSGraph supports different kinds of graph algorithms together by integrating Spark and PyTorch. We describe the implementations of graph algorithms and evaluate the performance of PSGraph via extensive experiments.

VIII. ACKNOWLEDGEMENTS

This work is supported by NSFC (No. 61832001, 61702016, 61702015), the National Key Research and Development Program of China (No. 2018YFB1004403), Beijing Academy of Artificial Intelligence (BAAI), and PKU-Tencent joint research Lab.

REFERENCES

[1] G. Malewicz and et al., "Pregel: a system for large-scale graph processing," in *ACM SIGMOD*, 2010, pp. 135–146.

[2] Y. Low and et al., "Distributed graphlab: a framework for machine learning and data mining in the cloud," in *PVLDB*, 2012, pp. 716–727.

[3] X. Zhu, W. Chen, W. Zheng, and X. Ma, "Gemini: A computation-centric distributed graph processing system," in *USENIX OSDI*, 2016, pp. 301–316.

[4] J. Jiang, B. Cui, C. Zhang, and L. Yu, "Heterogeneity-aware distributed parameter servers," in *ACM SIGMOD*, 2017, pp. 463–478.

[5] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web." Stanford InfoLab, Tech. Rep., 1999.

[6] V. Batagelj and M. Zaversnik, "An o(m) algorithm for cores decomposition of networks," *arXiv preprint cs/0310049*, 2003.

[7] S. B. Seidman, "Network structure and minimum degree," *Social networks*, vol. 5, no. 3, pp. 269–287, 1983.

[8] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre, "Fast unfolding of communities in large networks," *Journal of statistical mechanics*, p. P10008, 2008.

[9] S. Bonner, I. Kureshi, J. Brennan, G. Theodoropoulos, A. S. McGough, and B. Obara, "Exploring the semantic content of unsupervised graph embeddings: An empirical study," *Data Science and Engineering*, vol. 4, no. 3, pp. 269–289, 2019.

[10] P. Cui, X. Wang, J. Pei, and W. Zhu, "A survey on network embedding," *TKDE*, vol. 31, no. 5, pp. 833–852, 2018.

[11] B. Perozzi, R. Al-Rfou, and S. Skiena, "Deepwalk: Online learning of social representations," in *ACM SIGKDD*, 2014, pp. 701–710.

[12] A. Grover and J. Leskovec, "node2vec: Scalable feature learning for networks," in *ACM SIGKDD*, 2016, pp. 855–864.

[13] J. Tang, M. Qu, M. Wang, M. Zhang, J. Yan, and Q. Mei, "Line: Large-scale information network embedding," in *WWW*, 2015, pp. 1067–1077.

[14] A. Narayanan, M. Chandramohan, R. Venkatesan, L. Chen, Y. Liu, and S. Jaiswal, "Graph2vec: Learning distributed representations of graphs," *arXiv preprint arXiv:1707.05005*, 2017.

[15] H. Dai, Z. Kozareva, B. Dai, A. Smola, and L. Song, "Learning steady-states of iterative algorithms over graphs," in *ICML*, 2018, pp. 1114–1122.

[16] W. Hamilton, Z. Ying, and J. Leskovec, "Inductive representation learning on large graphs," in *NIPS*, 2017, pp. 1024–1034.

[17] S. Cao, W. Lu, and Q. Xu, "Deep neural networks for learning graph representations," in *AAAI*, 2016, pp. 1145–1152.

[18] A. Jain, A. R. Zamir, S. Savarese, and A. Saxena, "Structural-rnn: Deep learning on spatio-temporal graphs," in *CVPR*, 2016, pp. 5308–5317.

[19] A. Roy, I. Mihailovic, and W. Zwaenepoel, "X-stream: Edge-centric graph processing using streaming partitions," in *ACM SOSP*, 2013, pp. 472–488.

[20] A. N. Sørnes, "A comparison of vertex and edge partitioning approaches for parallel maximal matching," Master's thesis, The University of Bergen, 2013.

[21] J. E. Gonzalez and et al., "Powergraph: Distributed graph-parallel computation on natural graphs," in *USENIX OSDI*, 2012, pp. 17–30.

[22] S. Ghandeharizadeh and D. J. DeWitt, "Hybrid-range partitioning strategy: A new declustering strategy for multiprocessor database machines," in *PVLDB*, 1990, pp. 481–492.

[23] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," *HotCloud*, p. 95, 2010.

[24] L. Cui, L. Yue, D. Wen, and L. Qin, "K-connected cores computation in large dual networks," *Data Science and Engineering*, vol. 3, no. 4, pp. 293–306, 2018.

[25] A. Kyrola, G. Blelloch, and C. Guestrin, "Graphchi: Large-scale graph computation on just a pc," in *USENIX OSDI*, 2012, pp. 31–46.

[26] X. Zhu, W. Han, and W. Chen, "Gridgraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning," in *USENIX ATC*, 2015, pp. 375–386.

[27] J. E. Gonzalez and et al., "Graphx: Graph processing in a distributed dataflow framework," in *USENIX OSDI*, 2014, pp. 599–613.