# GraphScope: A Unified Engine For Big Graph Processing

Wenfei Fan[§,†], Tao He, Longbin Lai, Xue Li, Yong Li, Zhao Li, Zhengping Qian, Chao Tian, Lei Wang, Jingbo Xu, Youyang Yao, Qiang Yin, Wenyuan Yu, Jingren Zhou, Diwen Zhu, Rong Zhu
Alibaba Group [§] University of Edinburgh [†] Shenzhen Institute of Computing Sciences
graphscope@alibaba-inc.com

## ABSTRACT

GraphScope is a system and a set of language extensions that enable a new programming interface for large-scale distributed graph computing. It generalizes previous graph processing frameworks (*e.g.,* Pregel, GraphX) and distributed graph databases (*e.g.,* JanusGraph, Neptune) in two important ways: by exposing a unified programming interface to a wide variety of graph computations such as graph traversal, pattern matching, iterative algorithms and graph neural networks within a high-level programming language; and by supporting the seamless integration of a highly optimized graph engine in a general purpose data-parallel computing system.

A GraphScope program is a sequential program composed of declarative data-parallel operators, and can be written using standard Python development tools. The system automatically handles the parallelization and distributed execution of programs on a cluster of machines. It outperforms current state-of-the-art systems by enabling a separate optimization (or family of optimizations) for each graph operation in one carefully designed coherent framework. We describe the design and implementation of GraphScope and evaluate system performance using several real-world applications.

**PVLDB Reference Format:**
Wenfei Fan, Tao He, Longbin Lai, Xue Li, Yong Li, Zhao Li, Zhengping Qian, Chao Tian, Lei Wang, Jingbo Xu, Youyang Yao, Qiang Yin, Wenyuan Yu, Jingren Zhou, Diwen Zhu, Rong Zhu. GraphScope: A Unified Engine For Big Graph Processing. PVLDB, 14(12): 2879 - 2892, 2021.
doi:10.14778/3476311.3476369

**PVLDB Artifact Availability:**
The source code, data, and/or other artifacts have been made available at https://github.com/alibaba/GraphScope/tree/vldb.

## 1 INTRODUCTION

Distributed execution engines with high-level language support such as Koalas [45], Dask [15], and TensorFlow [5], have been widely adopted with great success in the development of modern data-intensive applications. Two factors largely account for the success of these systems. First, they provide developers with easy access to a core subset of domain-specific operators, such as relational join, matrix multiplication, and convolution, and allow further extensions via arbitrary user-defined functions. Second, they adopt the dataflow execution model, which is more scalable

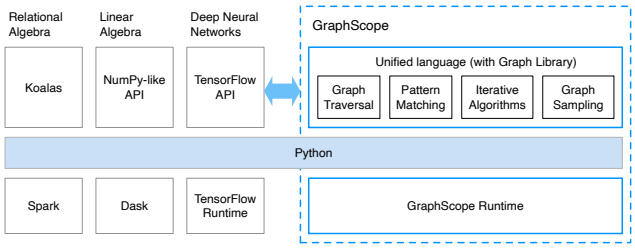



**Figure 1: The GraphScope system stack, and how it interacts with the PyData ecosystem.**

than alternative parallel-computing paradigms, such as Parallel Random Access Machines (PRAM) [20], and enables sophisticated optimizations to achieve high performance.

However, the operator semantics supported by these systems are ill-suited to efficiently solve a variety of important problems which requires a deeper analysis of heterogeneous data. In such cases, analysis tools involving graph computation are often called for instead. Social network mining, for example, routinely requires ranking and classification algorithms such as PageRank [44], connected components [29], and betweenness centrality [23]; similarly for identifying fraudulent activities in on-line payments, which involves matching of complex (subgraph) patterns [48]. And many algorithms [60] commonly used in product and advertisement recommendation boil down to deep learning over graph-structured data.

Given the importance of graph computation, it is rational to design a scalable engine for processing large graphs with high-level language support. Ideally, such an engine should allow developers to easily program graph algorithms and naturally exploit graph-specific optimizations, while at the same time, maintaining the scalability and efficiency of a dataflow execution model. Unfortunately, it is challenging to efficiently scale diverse graph computation types (Section 2.2) that require different design trade-offs and optimizations tightly coupled with specific programming abstractions. For example, while the popular vertex-centric model [34] works nicely for iterative algorithms, developers still have to derive specializations for random graph walking and pattern matching [24, 61]. This explains why existing graph processing systems [25, 32, 39] are designed for a particular type of graph computation.

In contrast, real-world graph applications are often far more complicated that intertwine many types of graph computation in one single workload. As a result, developers often have to comprise multiple systems with potentially very different programming models and runtime, which gives rise to a number of issues such as managing the complexities of data representation, resource scheduling, and performance tuning across multiple systems, etc. There


Proceedings of the VLDB Endowment, Vol. 14, No. 12 ISSN 2150-8097.
doi:10.14778/3476311.3476369

thus urgently needs a unified programming abstraction and runtime that allows developers to write applications in a high-level programming language for a wide range of graph computations.

Furthermore, in web-scale graph analytics, a graph pipeline also includes the construction of an input graph from various sources, as well as the preparation of final results for the downstream tasks to consume, which needs complex data extraction, cleaning, and transformations (such as joins), and often requires excessive data movements and non-trivial interplay among an array of systems (such as Hadoop [37] and Spark [63]). There have been attempts to implement diverse graph computations on top of the dataflow model (GraphX [26]) to ease such inter-operations. However, casting graph computations as a sequence of relational operators can incur significant overheads for problems that require low-latency such as graph traversal [4]. Moreover, as we will show in Section 5, it fails to take advantage of the well-defined semantics of graph computations to enable sophisticated optimizations such as pipelining. As a result, while distributed graph algorithms are already hard to implement efficiently in existing systems, implementing complex graph pipelines becomes more challenging.

To tackle the aforementioned problems, we propose a unified engine for big graph processing called GraphScope. Figure 1 gives the conceptual overview of the GraphScope system stack. At the bottom is a dataflow runtime that serves as the fabric to compose distributed execution of different graph computations, leveraging all the resources available in a cluster. This execution layer enables a separate optimization (or family of optimizations) for each graph computation in one carefully designed coherent framework, while at the same time it offers a simple and powerful programming interface. Further up the stack, we have developed a graph library with a large fraction of frequently used graph computations. Last but not least, by embedding the language (and hence the graph library) in Python, GraphScope can be integrated with other existing engines to deliver a holistic development experience.

In summary, we make the following contributions:

- A simple and unified programming interface for a wide variety of graph computations, which supports language constructs for graph traversal, pattern matching, iterative algorithms, and graph sampling (for GNNs).
- A distributed dataflow runtime that enables a separate optimization (or family of optimizations) for each graph operation in one carefully designed coherent framework.
- An in-memory data store that automatically manages the representation, transformation, and movement of intermediate results to facilitate efficient distributed execution for different computations.
- We adopt the language integration approach advocated by Python to integrate the graph operators into a general-purpose high-level programming interface. This approach allows us to seamlessly combine GraphScope with other data processing systems such as Koalas, Dask, and TensorFlow, and thus provide the functionality of relational algebra, linear algebra, graph algorithms and machine learning in one unified platform.

The rest of the paper is organized as follows. Section 2 reviews the graph data model and operations, and highlights limitations of existing graph processing systems. Section 3 describes GraphScope's programming interface. Section 4 and 5 detail our design and implementation of GraphScope. Section 6 describes example applications and Section 7 presents evaluation results. Section 8 discusses related work and we conclude in Section 9.

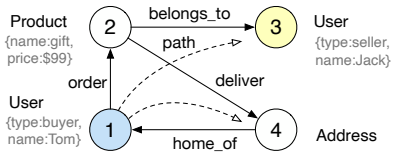



**Figure 2: An example "e-commerce" property graph.**

## 2 BACKGROUND AND PRELIMINARIES

As mentioned in the introduction, existing large-scale graph computing engines are typically tailored to solve one particular type of graph computation. In this section we provide a categorisation of graph computation and review the limitations of a state-of-the-art framework in unifying graph computation.

### 2.1 Property Graph Data Model

In graph computation, data is typically represented as a property graph [9], in which vertices and edges can have a set of properties. Every entity (vertex or edge) is identified by a unique identifier (`ID`), and has a (`label`) indicating its type or role. Each property is a key-value pair with combination of entity `ID` and property name as the key. Figure 2 shows an example property graph. It contains `user`, `product`, and `address` vertices connected by `order`, `deliver`, `belongs_to`, and `home_of` edges.

### 2.2 Categorisation of Graph Computation

While there is no textbook categorisation of graph computation, we propose one based on the long history of research and practice of graph computation. We first consider two major types, namely graph traversal and pattern matching, from the core of a variety of modern graph query languages [42, 51]. In addition, we bring into discussions the graph analytics and graph sampling as another two types for their usefulness in practice. Note that our categorisation compoensates a prior survey in [8] that only concerns about graph traversal (path) and pattern matching. We give details as follows.

**Graph traversal** is the most basic graph computation that visits vertices (and/or edges) of a graph in a certain order. Figure 2 shows an example traversal that starts from vertex 1, follows outgoing edges for 2 hops, and yields two paths $1 \longrightarrow 2 \longrightarrow 3$ and $1 \longrightarrow 2 \longrightarrow 4$.

**Pattern matching** is another important use case, in which a subgraph containing variables is created by the user and all graph elements that bind to those variables are returned as the result set. In essence, a complex pattern can be divided into a set of simple edges or paths and matched using graph traversal, followed by a multi-way join to compute the final results.

**Graph analytics** focuses on structural characteristics of the graph as a whole or of a pairwise relationship between two entities in the graph. Examples include finding shortest path (*e.g.,* the Dijkstra's

algorithm) and connected components, PageRank [44], clustering, and community detecting (*e.g.,* Louvain [11] and LPA [49]), etc.

**Graph sampling** is a special traversal operation used to generate samples for training graph neural networks. For each sample, the process of sampling usually starts from one single seed (vertex or edge). After a vertex has been sampled, the knowledge of the vertex's in-edges and out-edges can be used to choose the next vertex. The policy of choosing the next vertex depends on the design of the sampling algorithms.

## 2.3 TinkerPop

TinkerPop [56] is a powerful framework for developing graph applications, based on the Gremlin query language [51]. It introduces a rich set of operators for graph traversal, pattern matching and sampling, while at the same time, embeds iterative graph algorithms such as PageRank and shortest path as predefined operators.

TinkerPop exposes a set of interfaces that make it possible for system vendors to provide different implementations. To support large graph, ideally, an implementation has to support all kinds of the graph operations at scale, which is extremely challenging. Although there are a large number of TinkerPop-enabled systems, these systems retrofit Gremlin language and interfaces into existing frameworks such as Hadoop or Spark. Due to the fundamental limitation of each framework, these systems either offer a limited subset of the language constructs (such as the lack of nested loops in Grasper [14]), or come at the price of degradation of performance (*e.g.,* Hadoop-Gremlin [27] for graph traversal).

Furthermore, these systems use the concept of a *vertex program* [39] to support user-defined iterative algorithms: this is necessary as the standard algorithms are rarely enough for real-world applications in which users have to implement their own specific algorithms for a particular task. However, such a model requires deep understanding of low level primitives such as graph partitioning and message passing to develop new algorithms, making efficient distributed graph computing a privilege to experienced users only [34]. As a result, even TinkerPop provides a high-level programming model to a variety of graph operations, it is rarely used in Web-scale data analysis.

# 3 PROGRAMMING INTERFACE

GraphScope extends Gremlin with a small set of data-parallel operators to cover complex iterative algorithms and provide a unified programming interface embedded in Python. This allows GraphScope to seamlessly integrate with other, existing data-parallel systems such as Koalas, Dask, and TensorFlow. This section provides a high-level overview of this programming interface.

## 3.1 Gremlin

Gremlin is a de facto standard language that allows high-level and declarative programming for various graph operations, including graph traversal, pattern matching, and sampling. For Gremlin applications, data is represented as streams of *traversers*. A traverser is the basic unit of data processed by a Gremlin engine. Each traverser consists of three parts: a reference to the current location (vertex, edge or property) being visited, the path history, and (optionally) an application state (also known as a *sack*). For example, the traversal

```
# Q1: Cycle detection using graph traversal.
g.V('account').has('id','2').as('s')
 .repeat(out('transfer').simplePath())
 .times(k-1)
 .where(out('transfer').as('s'))
 .path().limit(1)
# Q2: Graph sampling using a 5-hop random walk.
g.V().repeat(local(
  bothE().sample(1).by('weight').otherV()))
 .times(5).path()
# Q3: Pattern matching using match().
g.V().match(
  as('directors').hasLabel('person'),
  as('directors').in('director').as('movies'),
  as('movies').out('actor').as('directors'))
 .select('directors','movies')
```

**Figure 3: Example Gremlin queries for graph traversal, sampling and pattern matching.**

shown in Figure 2 can be executed as follows. Initially, there is only one traverser at vertex 1. A possible intermediate result is a collection of a single traverser located at vertex 2 with the corresponding path history ($1 \rightarrow 2$). The final result consists of two traversers, located at vertex 3 and 4, respectively, with different paths.

Gremlin operators perform transformations on traverser streams, and Gremlin queries are computations formed by composing these operators. The source operator (`V`) defines the starting vertices; and each of the graph-walking operators (`out`, `in`, `both`) walks a graph from the current locations by one hop, along out-edges, in-edges, or edges of both directions, respectively. The `sample` operator is useful for sampling some number of traversers previous in the traversal. The `match` operator provides supports for pattern matching. The user provides a collection of "patterns" carrying variables that must hold true throughout the duration of the `match`. Furthermore, Gremlin offers familiar relational operators, including projection (`select`), filters (`has`), grouping (`group`), aggregation (`count`), and top-K (`limit`), together with dynamic control-flow constructs such as conditionals (`where`) and loops (`repeat`). More details on the Gremlin operators can be found in [51].

**Example**. Figure 3 shows three example Gremlin queries.[1] The first query, `Q1`, finds cyclic paths of length $k$, starting from a given account. First, the source operator `V` (with the `has` filter) returns all the `account` vertices with an identifier of "2". The `as` operator is a *modulator* that does not change the input collection of traversers but introduces a name ($s$ in this case) for later references. Second, it traverses the outgoing `transfer` edges for exact $k-1$ times, skipping any repeated vertices (by the `simplePath` operator). Third, the `where` operator checks if the starting vertex $s$ can be reached by one more step, that is, whether a cycle of length $k$ is formed. Finally, for qualifying traversers, the `path` operator returns the full path information. The `limit` operator at the end indicates only one such result is needed. The second query, `Q2`, uses the `sample` operator to support the execution of random walks [58], which is a fundamental building block for graph sampling widely used in

[1] In situations where Python reserved words and global functions overlap with standard Gremlin steps and tokens (as(), in(), with(), and etc.), those bits of conflicting Gremlin get an underscore appended as a suffix. We omit this suffix for brevity in this paper.

```
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-$pr)>1e-10'))
        .property('$pr', expr('$new')))
    .until(count().is(0)))
 .withProperty('$pr', 'pr')
 .order().by('pr', desc).limit(10)
 .valueMap('name', 'pr')
```

**Figure 4: A PageRank implementation in Gremlin.**

GNN training. In the last query `Q3`, three "traversal fragments" are nested within the `match` operators used as subgraph patterns.

The rich set of operators provided by Gremlin make it easy to express a wide variety of graph computations by familiar notations. However, although each traverser can carry a state local to the traversal path, the execution is side-effect free with regard to the underlying graph itself. This makes it hard for Gremlin to represent iterative graph algorithms using the vertex-centric programs, which often require additional state or properties to be maintained on each vertex in the graph efficiently. Moreover, some graph-centric algorithms, such as Dijkstra's algorithm [22] for the single source shortest paths (SSSP) problem, are hard to implement in this model [19].

## 3.2 GraphScope Extension

To address the above challenges, this section describes a very small set of GraphScope-specific extensions that we believe are essential to model complex iterative algorithms using high-level operators. They are integrated in the Gremlin programming interface to provide a powerful hybrid paradigm of declarative and imperative programming for large-scale graph computing.

GraphScope introduces several Gremlin extensions (which together form a `process` step) to support iterative graph algorithms, either vertex-centric or graph-centric. The first extension is to introduce the states of vertices. By extending the `property(Key, Value)` step, a special *runtime* property beginning with the "`$`" identifier can be created or updated on the current location (head of traverser). While Gremlin allows the head to be a vertex, an edge, or a property value, we constrain that only the traverser at a vertex can call the runtime properties. The runtime properties are accessible only during the course of the same `process` step, but they can be added to the graph data as ordinary properties through a `withProperty` step after that `process` step. For example, the code at line 2 in Figure 4 introduces `$pr` as such a property, and initializes it. This runtime property is updated later as an application state, and finally added to the graph as a new property `pr`.

The second extension is the "Scatter-Gather" step, which is implemented by `scatter(ValueSupplier).by(Traversal).gather(RuntimePropertyName, AggregateEval)`. This extension is mainly for supporting the logic of vertex-centric computation and its underlying message passing. The `scatter` operator packs application messages into the traversers' sacks, and then sends them to target vertices (which are selected by a sub-traversal). The `gather` operator aggregates messages (sacks)

```
@graphscope.step()
class SSSP(graphscope.PIE):
  def PEval(self, g, context):
    self.p = context.get_param("edgeProperty")
    self.d = context.get_param("distProperty")
    g[context.get_param("srcID")][self.d] = 0
    self.dijkstra(g, [context.get_param("srcID")])
  def IncEval(self, g, updates):
    self.dijkstra(g, updates)
  def dijkstra(self, g, updates):
    heap = VertexHeap(g, self.p)
    for i in updates:
      val = g[i][self.d]
      heap.push((i, val))
    while not heap.empty():
      u = heap.top().vid
      distu = heap.top().val
      heap.pop()
      for e in g.get_outgoing_edges(u):
        v = e.get_neighbor()
        distv = distu + e.data(self.p)
        if g[v][self.d] > distv:
          g[v][self.d] = distv
          if g.is_inner_vertex(v):
            heap.push((v, distv))

graphscope.registerUDF('SSSP', SSSP)
# SSSP from vertex with id '1234'
g.V().process('SSSP')
  # use the edge property "weight" to run SSSP
  .with('edgeProperty', 'weight')
  # record the distance result in the "$dist" column
  # in the context
  .with('distProperty', '$dist')
  .with('srcID', 1234)
  .withProperty("$dist", "dist")
```

**Figure 5: PIE of the single source shortest paths problem.**

from traversal paths using an aggregate operator, and saves the result in a specified runtime property on each current position of the path. For example, the code at lines 5 – 6 in Figure 4 sends the value of `$tmp` of a vertex to all of its outgoing neighbors, and then each vertex sums all received values and saves the result.

Based on the above extensions, the `process` step is added in GraphScope to implement graph processing algorithms. It processes the embedded graph and thus, all vertices or edges in it are analyzed (some times more than once for iterative, recursive algorithms). Typically in this step, `property` is used to create or update runtime vertex properties, `scatter-gather` is used for message passing between vertices, `where` can be used to filter the current vertices based on properties, and the source operator `V` is extended to reset the traversal with all vertices of the graph as sources. 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. Together with the `repeat`-loops, `process` can implement a "Scatter-Gather" iteration similar in Flink.[2] And this iterative computation will repeat until the convergence condition is met or a pre-defined number of iterations have taken place. With that, GraphScope allows easy expression of many vertex-centric graph algorithms using high-level language constructs only. Figure 4 shows a PageRank implementation in

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

```
# Use Mars to prepare the data for graphs
features = mars.read_csv('./features.csv')
persons = mars.read_csv('./person.csv')
persons = features[features.uid.isin(person.uid)]
# Construct the graph from various input
graph = sess.g()
  .add_vertices(person, label='person')
  .add_edges('hdfs://data/knows.csv', label='knows')
g = sess.gremlin(graph).traversal_source()
# Run a variety of graph algorithms.
sampler = g.V()
  .process('pageRank')
  .with('prProperty', '$pr')
  .withProperty('$pr', 'pr')
  .sample(___.V('person').batch(64).outV('knows').sample
       (10).by('random').values())
  .toTensorFlowDataset()
# Invoke TensorFlow to train a GNN.
embeddings = model.train(feed=sampler, ...)
```

**Figure 6: Inter-operation among Mars,** GraphScope**, and TensorFlow.**

extended Gremlin. A companion technical report [47] contains more sample graph programs using such a small set of language constructs, with a formal proof of its expressiveness.

### 3.3 PIE Model

In the case of that the graph algorithms are more complex and hard to be implemented using the "Scatter-Gather" model, GraphScope also supports to define an algorithm in Python. By using `registerUDF(name, UDF)`, a new user-defined algorithm is registered to GraphScope, which is callable as a UDF within the `process` step. For this step, the `with` modulating step could be used to provide configurations to the algorithm, and the `withProperty` step can add runtime properties to the graph data after executing the algorithm. For the sake of programming convenience, GraphScope implements the graph-centric PIE model as proposed by the GRAPE system [18]. Relying on this model, developers can access a global view of the graph and plug in existing conventional sequential (single-machine) graph algorithms, with minor changes. Specifically, developers only need to provide two functions, (1) PEval, a function that for a given query, computes the partial answer on a local fragment of the graph; different fragments exchange the status of their border nodes as messages; and (2) IncEval, an incremental function, computes changes to the old output by treating messages from other fragments as updates. Here GraphScope can automatically handle Assemble, which collects partial answers and combines them into a complete answer. Moreover, incrementalization methods have been developed [17] that are able to deduce incremental IncEval from PEval, using the same logic and data structures of PEval. Supporting the fixpoint computation of GRAPE, GraphScope parallelizes the computation of PEval and IncEval across multiple processors and machines. This makes parallel graph programming accessible to users who know conventional graph algorithms covered in undergraduate textbooks.

Figure 5 shows an implementation of SSSP using the PIE model. The example shows that programming a graph algorithm in GraphScope is straightforward, literally a direct translation of the algorithm (*e.g.,* Dijkstra's algorithm [22]) into a sequential program.

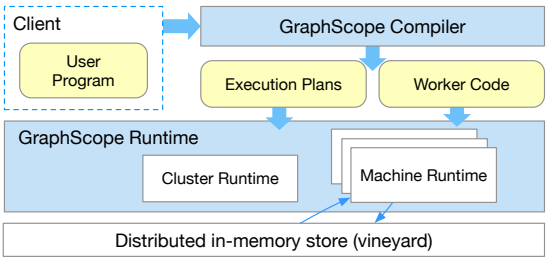



**Figure 7: The** GraphScope **system architecture.**

This allows even complex algorithms to be expressed naturally. For example, we have ported more than 30 algorithms from the well-known single-machine graph library of NetworkX [28].

### 3.4 Integration with PyData Ecosystem

We have shown in the previous section how to implement user-defined graph computations in GraphScope. One strength of our approach is that by embedding the language (and hence the graph library) in Python, it is natural and easy to inter-operate with other Python data processing systems such as TensorFlow and Mars [46]. This seamless integration of GraphScope with other data-parallel systems provides a unified and elegant solution to many real-world problems in which certain parts of the computation are naturally handled using relational operators; whereas other parts of the computation require graph operations, and machine learning.

To illustrate such mixed styles of data analysis, consider the example in Figure 6, where we use raw logs from e-commerce transactions to produce a recommendation for a product (item) to each user. The code shows that three systems (*i.e.,* Mars, GraphScope, and TensorFlow) inter-operate to finish this recommendation task with 3 steps, and each system conducts the part of the computation for which it is suited. The initial graph data is represented as text files stored in HDFS. In step 1, Mars loads data from the filesystem, and performs a join to prepare data for GraphScope. It boils down to creating a graph `g` to represent user-item transaction graph. In step 2, GraphScope is called to perform graph algorithms, which computes a PageRank for each user, followed by a sampling algorithm to generate samples using 2-hop neighbors. In step 3, running in parallel (pipelined) with the GraphScope computation, TensorFlow is used to further train an embedding for each item.

## 4 SYSTEM ARCHITECTURE

Figure 7 shows an architectural overview of the GraphScope system. There are two main components: the GraphScope compiler generates the execution plans and the worker code to be run on the cluster, and the GraphScope runtime uses the execution plans to distribute worker processes across multiple machines, schedule computation on each multicore server, and manage communications and intermediate results. It returns control to the client when the execution terminates. Because the flow of execution is similar to existing systems [55, 62, 63], this section highlights some of the unique features of the GraphScope system.

### 4.1 GraphScope Compiler

GraphScope automatically and transparently distributes a graph computation to a cluster for parallel execution. We describe a graph

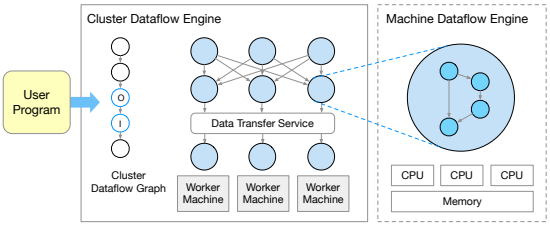



**Figure 8: An example dataflow execution in** GraphScope**.**

computation as a dataflow graph. Each vertex in the dataflow graph performs a local computation on input streams from its in-edges and produces output streams as its out-edges. To enable optimized execution for each type of graph computation supported by GraphScope, we allow a subgraph of dataflow to be treated as a *composite* vertex that can have its own execution strategy. This allows the GraphScope compiler to group together a segment of graph operations of the same type in a multi-staged processing pipeline and configure the strategy that best fits its computation in each stage.

Furthermore, to separate concerns of cluster scheduling and local (optimized) execution, the compiler structures the dataflow graph into two levels. The first is a dataflow graph at the stage/cluster level in which each vertex represents a single computation stage (or type). The compiler applies query rewrite rules to the Gremlin program to transform it into such a dataflow graph that is optimized for parallel execution on the cluster. Consider the `sampler` computation shown in Figure 6. It reads input graph `g` from an in-memory store and performs two stages of graph computation. Accordingly, two composite vertices will be created for the implementation of `pageRank` and `sample` operation, respectively. Finally, the result vertex writes to the same store for downstream computation (`train`) to consume. This simplifies cluster scheduling for which the partitioning of input/output graphs and each computation as well as the scheduling are all performed at a coarser granularity, and remain the same during the execution within the same stage.

At the second level, as composite vertices from the cluster-level dataflow graph are assigned to machines, each of them expands into a subgraph of the dataflow, consisting of graph operators as vertices running on each local machine. The GraphScope compiler generates worker code with the help of a *primitive library* installed on each machine. The primitive library contains pre-compiled Gremlin operators as generic templates which GraphScope instantiates dynamically at runtime, along with the other supporting functions. In our example, the primitive for `sample` is parameterized by both the strategy (`random`) and the projection function, and an `add_column` operator is inserted by the compiler to save the computation result of `pr`.

## 4.2 GraphScope Runtime

Figure 8 shows an example dataflow execution, which includes two levels of dataflow executions. Each level of dataflow graphs are handled by a different execution engine in the runtime, namely cluster dataflow engine, and machine dataflow engine, respectively. The cluster dataflow engine assigns input graph or intermediate data partitions to available machines, and the composite vertices in the *logical* plan that follow is replicated across the same set of machines. This forms the *physical* execution plan in which all the (composite) vertices at the same stage perform the same computation but demands different partitions of the input. The machine dataflow engine executes its own dataflow graphs on each machine, managing input/output and memory on a multi-core server.

During execution, the vertices read from and write to an in-memory store called Vineyard (Section 5.2). Vineyard is designed as a distributed in-memory store that provides zero-copy data sharing and management. It provides high-level abstractions or commonly used Python data structures, such as `Array`, `Tensor`, and `DataFrame`, as the storage interface, and enables easy and seamless integration with other data-parallel systems. Specifically, at the border between GraphScope and these systems, Vineyard can convert the Vineyard objects to objects of the target systems. Unless necessary, this process will avoid copying of the blobs (payloads). This allows GraphScope to manage transfer of intermediate results automatically and achieve high performance for the overall execution across multiple systems. Last but not least, data statistics are collected by Vineyard at runtime that enables dynamic optimizations such as dynamic graph partitioning and hybrid join for pattern matching (Section 5.3).

# 5 IMPLEMENTATION

This section provides implementation details of the components of the GraphScope system. GraphScope leverages a number of existing technologies and we therefore focus our attention mainly on the novel aspects of the system.

## 5.1 Two-Level Dataflow Execution

To support optimized graph computation with specialization, we built a new distributed in-memory dataflow engine called Gaia. In Gaia, the cluster execution is orchestrated by a coordinator that schedules the (composite) vertices to run on separate machines (similar to Dryad [31] and Spark [63]). Each vertex encodes a subgraph of dataflow operators that all run within a same process on each machine, managed by a local executor (like in Dandelion [52] and TensorFlow [5]). Gaia sets itself apart from existing dataflow engines in two important ways:

**Integrated Design with the** Vineyard **Store**. Inspired by Dryad [31], Gaia can modify the dataflow execution plan by enabling rewritings at runtime. Unlike Dryad, however, the rewriting can be directly based on the statistics provided by Vineyard without inserting additional statistics-collecting vertices.

**Optimized Local Execution with Specialization**. Gaia leverages the two-level dataflow scheduling to incorporate a variety of optimizations as execution strategies in local engines, which can be applied to each individual subgraph separately. This allows Gaia to support optimized and specialized runtime for each type of graph computation in one coherent framework, such as graph traversal [4], iterative computation [19], pattern matching [36], and graph sampling [65], as briefly described below.

- Iterative graph computations typically base the execution of iteration on the bulk synchronous parallel (BSP) model. However, as workers converge asymmetrically, the synchronization barriers result in that the speed of each iteration is

limited to that of the slowest worker. To solve this problem, Gaia uses a flow-controlled message queue for this case that allows better overlapping of computations and communications across iterations.

- Graph traversal can produce paths of arbitrary length, leading to memory usage growing exponentially with the number of hops. Fortunately, it is very common for Gremlin queries to terminate with a top-k constraint and/or aggregate operation, and therefore the memory crisis mainly stems from the intermediate paths. In this case, the scheduling policy can greatly impact the memory usage. Gaia adopts a hybrid BFS/DFS strategy, that is, it uses BFS-prioritized scheduling as it has better opportunities for parallelization, and automatically switches to DFS-prioritized in case that the current operator arrives at the memory bound.
- Graph sampling (for GNN training) is a special traversal that often starts from all the vertices in an input graph [65]. Graph pattern matching has long been treated as a multi-way join task [36]. In both cases, Gaia attempts to optimize for throughput by batching data.

## 5.2 Distributed In-Memory Store

We build Vineyard, a distributed in-memory store that provides zero-copy data sharing and management. Vineyard enables GraphScope to achieve high performance and the capability to exchange (intermediate) data efficiently across multiple systems.

Vineyard keeps data structures, such as graphs, dataframes/tables and tensors as objects, and each is assigned with an id. An object consists of a metadata (*i.e.,* the data layout) map and/or data payloads. The metadata map supports basic data types such as integer, boolean and string as values. Values can also be ids to refer other objects, making vineyard objects composable, *e.g.,* a set of array objects (each as a column) can form a columnar table object. Data payloads are a special type of objects in Vineyard called *blobs*. Each blob is a piece of continuous memory of requested size on a specific node for large payloads. Metadata maps are synced across the cluster using the key-value store etcd([2]). And blobs are kept in shared-memory arenas managed by Vineyard. They can be memory mapped (with zero-copy) to any worker processes of GraphScope and other systems such as Spark and TensorFlow as required.

For data-parallel execution, objects can be partitioned and stored distributedly over a cluster in Vineyard. A distributed object contains a metadata map only, which additionally contains a list of objects called *fragments*, where each fragment is a *local* object (*i.e.,* payloads live entirely on a local node). As an example, Figure 9 shows how Vineyard keeps a fragment of a property graph.

Moreover, Vineyard enables easy and seamless integration of GraphScope with other data-parallel systems. Specifically, at the border with these systems, Vineyard can convert the Vineyard objects to/from objects of the target systems. Unless necessary, this process will avoid copying of the blobs (payloads).

## 5.3 Optimizations

GraphScope performs both static and dynamic optimizations to optimize user code and improve performance, while being transparent to users. The static optimizations are greedy heuristics implemented

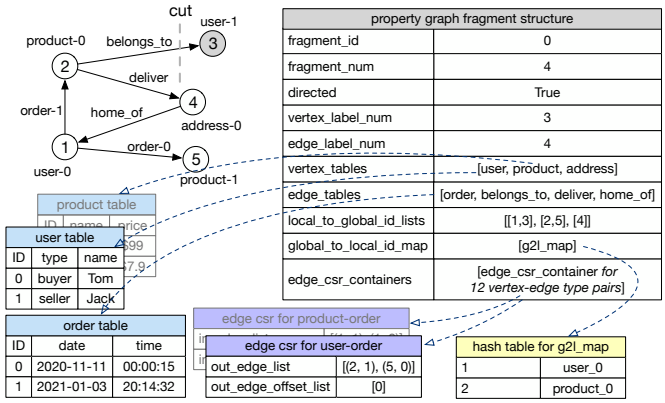



**Figure 9: A property graph fragment in** Vineyard**.**

as extensions to the GraphScope compiler. The dynamic optimizations are applied during job execution, and consist in rewriting the dataflow graph depending on runtime data statistics. The most important optimizations are:

**Pipelining**. Multiple vertices (in a logical plan) may be scheduled to execute all at once to enable pipelined execution. Its performance gain is operator-dependent, but it is one optimization that simultaneously exploits more parallelism, reduces burstiness of network traffic, and removes the dependency of (relatively) expensive serialization of the intermediate results (to/from Vineyard). In addition, pipelining can be applied across multiple frameworks. For example, through a special output adapter (`toTensorFlowDataset`) attached to a `sample` traversal, we can overlap graph sampling with the downstream GNN training (using TensorFlow).

**Adaptive Pulling/Pushing**. This technique is used to optimize computing efficiency in various iterative graph algorithms. GraphScope adopts a dual-mode processing strategy that enables the adaptive switch between *pull and push modes* according to the density of the active edge set. In a graph iterative algorithm, the *active edge set*, defined as the outgoing edges from vertices whose states are updated in a step, may vary in different computation steps. For example, in connected components, the active edge sets are relatively dense in the first few steps, and gradually become sparse since more vertices converge. In the *push* mode, updates are passed to neighboring vertices through outgoing edges. This is more efficient for sparse active edge sets, since we only need to traverse outgoing edges of active vertices. In contrast, the *pull* mode updates each vertex by collecting states of its neighbors along incoming edges. This is more beneficial for dense active edge sets, as it significantly reduces the contention in updating vertex states via locks or atomic operations. GraphScope takes advantages of the both modes via the adaptive switching.

**Dynamic Graph Partitioning**. Data partitioning is crucial for the efficient execution of a data-parallel computation. However, it is nontrivial to decide what partitioning strategies work the best and hence should be picked when data statistics are not available, and sub-optimal partitioning may lead to data and computation skew where the data or computations are not balanced among machines.

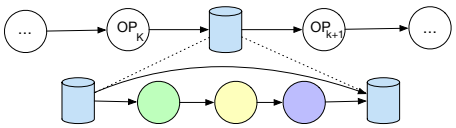



**Figure 10: An example of dynamic graph partitioning.**

GraphScope supports the determination of partitioning strategies at runtime. GraphScope first partitions an input graph with well-studied partitioning methods, *e.g.,* streaming partitioning algorithms [39, 57, 66] or offline partitioning heuristics [33, 54]. Due to the diversity of graph data and graph operations, an initially configured graph partition can become very skewed in multi-stage graph computation tasks. To alleviate this situation and improve the overall performance, GraphScope collects runtime statistics and learns a cost model to characterize the workload pattern for each graph computation; based on the cost model, GraphScope rewrites the dataflow execution to trigger a lightweight *re-partitioning* process to balance the workload for an intended graph operation.

As shown in Figure 10, let $\text{OP}_k$ and $\text{OP}_{k+1}$ be two consecutive graph operations in a multi-stage graph computation task. The output of $\text{OP}_k$ is written to Vineyard and then read by $\text{OP}_{k+1}$. GraphScope implements dynamic graph re-partitioning by rewriting the execution workflow as follows.

(1) GraphScope first inserts a statics-collection vertex in the execution dataflow graph (a green node in Figure 10). This is to sample small graphs from the output graph of $\text{OP}_k$.

(2) GraphScope next learns a cost model $C_{k+1}$ from the samples to characterize the workload for the subsequent operation $\text{OP}_{k+1}$. This is represented by a yellow node in the dataflow of Figure 10.

(3) Based on the cost model $C_{k+1}$, GraphScope re-balances the graph partition for operation $\text{OP}_{k+1}$ if necessary. Suppose the workload imbalance indicator estimated by the cost mode $C_{k+1}$ exceeds a predefined threshold. In that case, GraphScope triggers a lightweight heuristic to re-partition the graph [16] (represented by a purple node in Figure 10). This heuristic identifies vertices and edges and shuffles them from overload partitions to underloaded ones. If the partition is already balanced for $\text{OP}_k$, GraphScope skips the re-partitioning process and reads directly from the output of $\text{OP}_k$.

**Hybrid Join for Pattern Matching**. The performance of graph pattern matching is greatly impacted by the join algorithm [36]. Let's consider the pattern matching example (Q3) in Figure 3. For simplicity, we view the edges of a graph as a table of 3-tuple $(src\_id, id, dst\_id)$, denoting the identifiers of the source vertex, the edge, and the target vertex, respectively. There are hence two edge tables here: `director_edge(movie_id, id, director_id)` and `actor_edge(movie_id, id, actor_id)`. As a result, the pattern matching can be queried via following join:

```
SELECT DE.movie_id, DE.director_id
FROM director_edge as DE, actor_edge as AE,
WHERE DE.movie_id = AE.movie_id AND
     DE.director_id = AE.actor_id
```

Join algorithm itself is a widely studied subject, while we focus on the algorithms that have been adopted for distributed graph pattern matching, namely binary join [35] and worst-case optimal join [7]. Binary join processes pattern matching using multi-round of two-way joins, where each join simply shuffles the edge tables based on the join key and then conducts a conventional hash-join. Worst-case optimal join algorithm follows Ngo's algorithm [43] that gradually expands the pattern, each time by adding one more vertex. While neither algorithm can guarantee the best performance by all means [36], researchers have proposed hybrid solutions that mix both types of joins [6, 41]. The main idea of these works is to minimize the intermediate result size along the course of join processing, where the result size is estimated from a sampled graph.

In GraphScope, we follow [41] to process graph pattern matching using hybrid join algorithms. While [41] is developed in single machine, the techniques can be easily adapted to a distributed context. Issues remain on the sampling process for result estimation. While the graph is handy as for [41], it may be an arbitrary potion of the graph for GraphScope from a previous traversal query. There is an naive option to sample on the entire input graph, but the statistics can be too inaccurate to produce a good plan. In GraphScope, we rather leverage the dynamic graph rewriting of Gaia. Specifically, before the join operation, we insert an operator to process sampling on the runtime graph that will be the input for the task of pattern matching. The sampling is parallelized for each partition of the input graph, and then aggregated to one machine, on which the optimizer of [41] will be applied to produce the hybrid join plan to be injected into the dataflow, according to the result estimation of the sampled input graph. In order to do so, we carefully design the join plan produced by [41] to incorporate only the general-purposed dataflow operators such as `flatMap` (for vertex extension in worst-case optimal join) and `join` (for binary join).

## 6 APPLICATIONS

Large-scale graph computing has a wide range of applications, *e.g.,* social network analytics, fraud detection in financial trading, and cybersecurity monitoring. To check the usefulness of the GraphScope programming interface, we have developed a graph library with frequently used graph operations. Using the library, we have implemented several real-world applications, three of which are described in detail here as they form the basis of our evaluation in Section 7.

**Cybersecurity Monitoring**. We introduce an application that uses Gremlin for preventing Trojans. Trojans are often commanded and controlled by their holders through a number of malicious domains. The security researchers can prevent the transformation of Trojans by blocking the communication across these malicious domains. However, the holders can easily purchase new malicious domains to keep connection with their Trojans. As a result, the security researchers should be able to continuously find *unknown* malicious domains which are hosted by the Trojan holders. This task could be easily fulfilled using graph model.

Specifically, we take each domain $u$ and each IP $v$ as a vertex in the graph. Each edge $(u, v)$ is labeled "resolved_to" if the domain $u$ is resolved from the IP $v$. Each edge $(v, u)$ is labeled "query" if the IP $v$ has ever accessed the domain $u$. If the domain $u$ is known to be malicious, the resolved IP $v$ is very likely controlled by the Trojan holders, so other domains also resolved to $v$ are likely malicious. These domains could be detected by executing Gremlin query `Q1`.

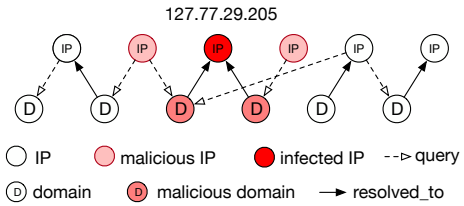



**Figure 11: Detecting injected IPs from known malicious IP.**

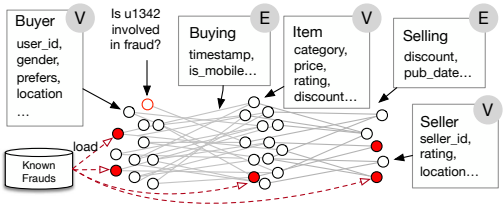



**Figure 12: A transaction graph.**

```
# Q1: A Gremlin query for detecting malicious domains.
g.V().hasLabel('domain').has('name','trojan.xx')
  .out('resolved_to').in('resolved_to').values('name')
# Q2: A Gremlin query for detecting infected machines.
g.V().hasLabel('ip').has('ip','125.77.29.205').in('
     resolved_to').in('query').values('id').dedup()
```

Further, from each known malicious IP $v$, any IP vertex that has ever accessed a domain resolved from $v$ is potentially to be infected by the Trojans. This process is illustrated in Figure 11. We can also detect these machines by executing the Gremlin query `Q2`.

Notice that, the above Gremlin queries could be also implemented by SQL queries on databases. We show the equivalent SQL query `Q3` of `Q2` as follows. Clearly, Gremlin queries are much simpler and easier for users. Moreover, as we reported in Section 7.2, their execution efficiency is much higher than SQL queries.

```
-- Q3: The equivalent SQL query of Gremlin query Q2.
SELECT DISTINCT a.id FROM ip_query_domain a JOIN
  (SELECT FROM domain_resolved_ip WHERE
   ip = '125.77.29.205') b ON a.query_name = b.name
```

**Fraud Detection**. Fraudulent transactions deceptively inflate ratings and rankings of sellers and items in the online marketplace. As an example, Figure 12 illustrates a simplified graph computing job for catching these transactions.

In the graph, 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 can be associated with frauds. By leveraging graph computing, can we determine whether a given user is involved in fraud?

One solution is to use label propagation algorithms to propagate known labels, and then employ a GNN model to do the classification [59]. With GraphScope, we can perform this job in a unified environment and achieves high performance with just a few lines.

```
# Build the graph
graph = sess.g().add_vertices(...).add_edges(...)
# Load the known labels
known_fraud = load_from('oss://.../fraud_score.tsv')
graph = graph.add_column('lpa_score', known_fraud)
# Call LPA algorithm
g = sess.gremlin(graph).traversal_source()
g.V().process('lpa')
    .with('lpaProperty', '$lpa_score')
    .with('epochs', 20)
    .withProperty('$lpa_score', 'lpa_score')
# Load a GraphSage Model for prediction
from graphscope.learning.models import GraphSage
model = GraphSage('oss://../some_model')
sampler = g.sample(...)
        .toTensorFlowDataset()
scores = model.predict(feed=sampler, config=...)
# Add "fraud" score back to graph
graph = graph.add_column('score', scores)
```

Data scientists can also conduct interactive explorations on the result graph thanks to the full Gremlin support. For example, to check any "hot" items (bought more than 10000 times in total) bought by a specific user (u1324) has a high "fraud" score.

```
g.V().has('id', 'u1324').out('buy').where(__.in('buy').
     count().is(gt(10000))).order().by('score', desc)
```

**Link Prediction**. Link prediction has found wide applications in marketing and recommendation. There have been a number of link prediction methods developed (see [40] for a survey). The information of tremendous users in our e-commerce platform can be easily maintained in the form of a big graph. Each vertex denotes a user of the platform, and each edge refers to the relationship between the two connected users. The link prediction application tends to predict those edges linking the entities that belong to the same group or family, denoted as linkage edges. After acquiring reliable linkage relationships, we can extract more accurate shopping preferences from the users and do better recommendation.

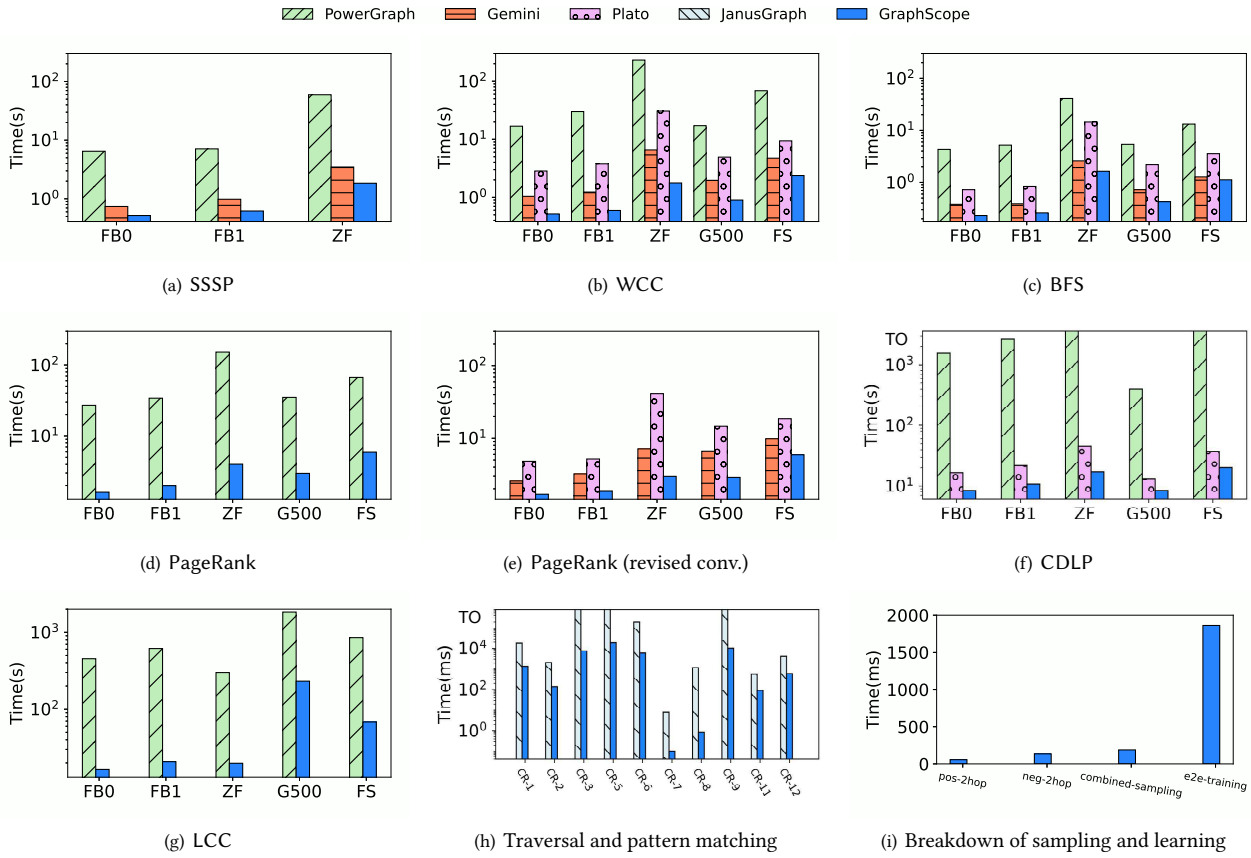
```
# load and build a graph from various sources
graph = sess.g().add_vertices(...).add_edges(...)
# compute CN, AA and RA scores for each pair of nodes
g = sess.gremlin(graph).traversal_source()
g.V().process('feature_extraction')
    .with('featureProperty', '$cn_aa_ra')
    .withProperty('$cn_aa_ra', 'cn_aa_ra')
# predict linkage edges with GNN
model = GraphSage(number_of_layers=2, ...)
sampler = g.sample(...)
        .toTensorFlowDataset()
embeddings = model.train(feed=sampler, ...)
sim_scores = embeddings.toNumpy().pairedDistances()
```

With GraphScope, we implement the link prediction task as shown above. We first build a large user network, by combing data collected from multiple sources. For each pair of vertices, we next compute three features, namely, common neighbors (CN), Adamic-Adar index (AA) and resource allocation index (RA) by a PIE program `feature_extraction`. This PIE program is just a straightforward translation of the sequential algorithms from NetworkX[3]. In the end, those features and other useful information are fed to subsequent GNN models to predicate linkage edges.

## 7 EVALUATION

In this section, we evaluate GraphScope's capability of efficient processing of large graphs for both synthetic workloads and the real-world applications described in Section 6.

[3]https://networkx.org/documentation/stable/reference/algorithms/link_prediction.html




(a) SSSP (b) WCC (c) BFS

(d) PageRank (e) PageRank (revised conv.) (f) CDLP

(g) LCC (h) Traversal and pattern matching (i) Breakdown of sampling and learning

**Figure 13: Performance evaluation**

## 7.1 Synthetic Workloads

Using synthetic workloads, we evaluated the performance of GraphScope on queries involving each single type of graph operations. We adopted the datasets and queries from the LDBC benchmark [30] for analytical, traversal and pattern matching queries. For GNN sampling queries which are not covered under the scope of the LDBC, we compared work-flows with and without the optimization strategies introduced in GraphScope.

All the experiments in this subsection were conducted on a managed Kubernetes cluster with 8 nodes, each was equipped with two 26-core Intel(R) Xeon(R) Platinum CPUs at 2.50GHz, and 768G memory. The nodes were connected with a network at 50Gbps.

**Exp-1. Graph Analytics Performance**. We first evaluated the graph analytics performance of GraphScope. The tests were deployed in containers, each with 16 cores and 256GB memory. We compared GraphScope with state-of-the-art parallel graph systems, including PowerGraph [25], Gemini [66] and Plato [3]. We tested all the six analytical queries of LDBC: (a) SSSP (single-source shortest paths), which computes the lengths of the shortest paths from a given source vertex to all other vertices in an edge-weighted graph, (b) WCC (weakly connected component) that decides the connected component each vertex belongs to in a directed graph, (c) BFS (breadth-first search), (d) PageRank that computes the PageRank score [44] for each vertex, (e) CDLP (community detection using label propagation), which finds communities using the parallel version of the label propagation algorithm [49], and (f) LCC (local clustering coefficient), which computes the degree to which the neighbors of each vertex form a clique within the graph.

We used four datasets provided by LDBC: (1) real-life social network com-friendster (FS) [1] with 65.6$M$ vertices and 1.81$B$ edges, and (2) synthetic graphs datagen-9_0-fb (FB0) with 12.9$M$ vertices and 1.05$B$ edges, datagen-9_1-fb (FB1) with 16.1$M$ vertices and 1.34$B$ edges, datagen-9_1-zf (ZF) with 434.9$M$ vertices and 1.04$B$ edges, and graph500-26 (G500) with 32.8$M$ vertices and 1.05$B$ edges. Since FS and G500 are not edge-weighted, we only evaluated SSSP over FB0, FB1 and ZF. In addition, we did not tested SSSP on Plato, CDLP on Gemini, and LCC on Gemini and Plato since the corresponding implementation codes are not available.

Figures 13(a)-13(g) show the elapsed time for executing graph analytics in different systems. Note that the results for running PageRank on Gemini and Plato are not consistent with those verified by the LDBC, hence we made our best efforts to revise the convergence condition in our PageRank algorithm such that it has the same outputs as the competitors. Figure 13(e) reports the results under such revised convergence condition for PageRank. We can see that GraphScope substantially outperforms the competitors in all cases. It is on average 34.7× (resp. 1.9× and 5.1×) faster than PowerGraph (resp. Gemini and Plato), up to 130.9× (resp. 3.7× and 17.4×). Most of the tasks can be done in less than 100 seconds on GraphScope. In contrast, PowerGraph cannot terminate in an hour (marked as TO) when performing CDLP over ZF and FS.

**Table 1: Statistics of the synthetic graph and the output sizes of the positive and negative sampling.**

| #vertices | #edges | #attributes | graph size | pos-samples | neg-samples |
|---|---|---|---|---|---|
| 300M | 3B | 100 | 242GB | 145TB | 364TB |

**Table 2: Evaluation time of the cybersecurity application.**

| System | Language | Execution Time (Secs) |
|---|---|---|
| GraphScope | Gremlin | 0.49 |
| ODPS | SQL | ~1,200 |

**Exp-2. Graph Traversal and Pattern Matching Performance**. For comparison, we consider Gremlin queries from the Social Network Benchmark defined by LDBC to model industrial use cases on a social network akin to Facebook. We chose 10 out of 14 complex read queries (denoted as CR-1 to CR-14) from LDBC's Interactive Workload. These queries contain the features of both graph traversal and pattern matching. For example, the query CR-3 contains the traversal of finding friends of persons $P$ and friends of friends of $P$, as well as matching the pattern that persons $P$ have authored the posts in the given countries. As a result, we conducted this experiment to evaluate the integrated performance of traversal and pattern matching instead of looking into the individual. We allowed each query to run for at most 1 hour, and marked an TO if a query can not terminate in time. We generated a large LDBC dataset with scale factor 100 using generator provided by LDBC. The generated dataset has $283M$ vertices and $1.754B$ edges.

Note that we only compared with JanusGraph, as it is the only system that can handle graphs at this scale. JanusGraph cannot process query in parallel, and we ran GraphScope in a single container for fair comparison. The graphs were stored in 8 nodes for JanusGraph, and one single node[4] for GraphScope.

We ran each LDBC query on GraphScope, recorded its latency and compared to JanusGraph. The results are reported in Figure 13(h). As shown there, GraphScope outperforms JanusGraph at the scale of orders of magnitude in all cases. It is on average 260× faster than JanusGraph, up to 1430×. JanusGraph failed to answer many queries (CR-3,CR-5, CR-9) due to `TIMEOUT` (TO). Although GraphScope is designed to scale in a cluster, it can further benefit from multi-core parallelism in a single node to improve query performance, especially for large queries, as depicted in Figure 13(h).

**Exp-3. GNN Sampling Performance.** GraphScope integrates the distributed graph sampling engine into the GNN training framework, allowing us to optimize the end-to-end training performance by properly pipelining various stages during the GNN training. To demonstrate the benefits of this pipelined approach in GraphScope, we trained a GraphSAGE model on a synthetic graph (described in Table 1). The sampling stage consists of two parts: a 2-hop positive edge sampling and a 2-hop negative edge sampling. In the negative sampling, we took 5 negative samples for each edge. The fan-outs of the first and second hops in both sampling were set to 10 and 2, respectively. We present (1) the output sizes of the positive and negative sampling, and (2) the breakdown of the sampling and training time in the end-to-end GraphSAGE training process.

As presented in Table 1, the sample size is three orders of magnitude larger than the graph size. Storing such a huge sample externally could easily prolong the end-to-end training time for multiple hours. To solve this problem, GraphScope samples the graph in mini-batches, and feeds the generated sample of each batch immediately into the training stage in a pipelined approach, instead of materializing them externally in order to be accessed by the training stages later. Figure 13(i) shows a breakdown of the average sampling time and the end-to-end training time per iteration during the training. These results clearly illustrate that GraphScope can overlap almost all the sampling and training stages by pipelining their execution, and thus optimize the end-to-end training time.

[4]Note that JanusGraph is properly warmed up to reduce the cost of pulling data from remote storage.

## 7.2 Real-World Applications

GraphScope has been widely deployed in production at Alibaba, and it supports thousands of graph jobs every day. Next, we report results from production for the applications described in Section 6.

**Exp-4. Cybersecurity Monitoring**. We report the evaluation results of using GraphScope on the cybersecurity application. As stated in Section 6, the Gremlin queries were executed on the graphs between domains and IPs to detect malicious domains and infected IPs. Our graph was extracted from the real-world AliCloud production environment, which contains around 600 millions vertices (including both domains and IPs) and 3.8 billions edges.

In this case, the security experts have collected a number of malicious domains related to "Double-Gun", a serious Trojan occurred on the Internet at the end of 2018. At first, we executed the Gremlin query `Q1` starting from these known malicious domains to detect unknown malicious domains and IPs. Then we ran the Gremlin query `Q2` starting from all malicious IPs to find other infected IPs. For comparison, we also ran the equivalent SQL queries of `Q1` and `Q2` on ODPS, the database warehouse service in AliCloud. The end-to-end execution time is shown in Table 2.

Clearly, executing Gremlin queries on GraphScope is more than $2,400\times$ faster than running SQL queries in ODPS. Both `Q1` and `Q2` could be regarded as two-hop traversals on graphs. By using GraphScope, we totally omit the costly join operations in SQL queries, so the time efficiency is greatly improved.

**Exp-5. Fraud Detection**. We also evaluated the performance of fraud detection over the transaction graphs of Taobao, which contained all transactions in a period of 30 days. We tested two implementations following the steps described in Section 6: an initial native attempt and the optimized version using GraphScope.

The native approach is inefficient to handle this complex work-flow. Since there is no one-stop solution, each part of the work-flow was conducted on an isolated system. For label propagation, it failed to load the data to ODPS Graph since it is too large. As a compromise, we used an optimized UDF defined on MaxCompute to execute this algorithm. It took 10 hours to run 20 epochs of the propagation. For learning task, it took 3.4 hours for 10 epochs in the system PAI, with an extra I/O cost from MaxCompute storage of 44.1 mins. At last, MaxCompute needed to load the results of the learning from the disk, which took 36.4 mins, and ran a set of Gremlin queries in 1.5 seconds on average. To sum up, it took about 14.6 hours to process the end-to-end work-flow, which is hard to meet the requirement of timeliness in the fraud detection.

With GraphScope, the work-flow can be processed seamlessly in a single system. It took 21.9 mins to load the graph, and then the partitioned graph data was resident in the Vineyard distributedly until the end of the job. It took 86.5 mins and 3.3 hours for label propagation and graph learning, respectively, and on average 0.2 seconds for the same set of traversal queries in Gremlin. There were no extra cost for I/O between these processes. This solution achieved a large performance improvement on end-to-end time with 5.1 hours only, which is 2.86× faster than the native approach.

**Exp-6. Link Prediction**. We next evaluated the performance of a graph analytical operation in the link prediction (see Section 6). It computes the features of CN, RA and AA over a large real-world user network with more than 66 billion edges.

We compared the performance of GraphScope with another baseline that was previously deployed over Giraph. The baseline stores the graph in relational tables and checks every pair of vertices using SQL queries, combined with user-defined similarity functions for feature computing. GraphScope improved its performance by more than 2.7×. This ODPS Graph baseline was only able to produce the CN features alone with more than 7 hours. In contrast, GraphScope produced all CN, RA and AA features within 2.5 hours.

**Exp-7. Online Recommendation Services**. We evaluated the performance of graph learning based real-world online recommendation services in Taobao, an e-commerce website which serves for billions of users and merchandises. The graph dataset is a heterogeneous graph containing billions of vertices (users and items) and tens of billions of edges (user-item, user-user and item-item interactions), where each kind of vertex contains about 30+ properties, such as age, gender and title. Based on the graph, the bipartite-GraphSAGE model is applied to learn representation for each user and item. Then some items are recommended to a specific user by computing the similarities of user-item pairs.

Previously the training pipeline of the model consists of a data preprocessing phase using a distributed big data engine, and a training phase, which took 23.6 hours to train a model end-to-end. In comparison, GraphScope can pipeline the sampling process that generates training data, and the training phase, reducing the end-to-end latency by 4.5×, to 5.2 hours. More importantly, GraphScope is an integrated IDE and largely simplifies the procedure of algorithm exploration, making in-time model update realistic to keep up with the trend shift on the e-commerce platform.

**Summary**. We find the following. (1) By incorporating various types of optimizations, GraphScope already outperforms state-of-the-art systems that are designed for different types of graph queries. (a) For iterative queries, it is on average 34.7× (resp. 5.1×) faster than PowerGraph (resp. Plato); (b) for traversal and pattern matching, it beats JanusGraph by 260× on average; and (c) its pipelining execution of all sampling and training stages in graph learning is beyond the reach of existing GNN systems. (2) Better still, GraphScope performs well in real-world applications. (a) GraphScope improves the performance of SQL-based solutions by more than 2400× (resp. 2.7×) in cybersecurity (resp. link prediction); (b) it is also 2.86× faster than artificially assembled approach in multi-staged fraud detection; and (c) it reduces the training time of complicated models over real e-commerce heterogeneous graphs from 23.6 to 5.2 hours.

## 8 RELATED WORK

Our contributions are multi-faceted, spanning across programming interface and scalable computation of a wide range of graph algorithms. We discuss GraphScope's novelties in these areas.

**Programming Interfaces**. Graph queries are typically expressed via graph traversal and pattern matching. Correspondingly, Gremlin [51] and Cypher [21] are the most popular query languages. However, they are not suitable for describing iterative graph algorithms, for which graph processing frameworks [25, 26, 39, 66] are often called for instead. It is also hard to write distributed graph algorithms using these frameworks, making them a privilege for experienced users only [19]. In contrast, GraphScope extends Gremlin with a set of data-parallel operators to provide a unified programming interface for complex graph algorithms. Other notable research projects in parallel declarative languages, such as Cilk [12], can be leveraged by GraphScope in theory, but they are not particularly tailored for distributed graph computation.

**Graph Databases**. Gremlin is widely supported by many graph databases, such as Neo4j [42], OrientDB [38], JanusGraph [32], and cloud-based services like Cosmos DB [50] and Neptune [10]. However, their query processing is limited to single process. In light of this, several distributed graph systems emerge such as Trinity [53], Wukong+S [64], Grasper [14] and A1 [13]. Trinity offers a programming model that is much less flexible than Gremlin. Grasper adopts Gremlin but provides a limited subset of the language constructs (e.g., the lack of nested-loop support). Wukong+S and A1 leverage RDMA for ultra-low latency to serve micro-second queries with high concurrency, which is not the main target of GraphScope.

**Graph Processing Systems**. In contrast to many existing systems that deal with batch-oriented iterative graph processing, such as Pregel [39], PowerGraph [25], GraphX [26] and Gemini [66], GraphScope preserves the elegant and well-formed data and computation model from the graph database research in a declarative language that allows user-defined functions. In addition, GraphScope uses dataflow as a unified computation model and recasts system optimizations developed in the context of specialized graph processing systems as dataflow optimizations. GraphScope outperforms many of the state-of-the-art graph systems accordingly.

## 9 CONCLUSION

We are witnessing the rise of a new type of graph applications that combine a wide range of graph algorithms into a single workload. Often experimental in nature and operating on large-scale datasets, this new type of graph applications needs a system that is similarly easy to program, scalable, and inter-operable. In this paper, we present a system, GraphScope, that takes on the challenge of building a unified engine for diverse big graph computations while at the same time offering a powerful and concise declarative programming interface.

## ACKNOWLEDGMENTS

We are grateful to Alibaba GraphScope team members for their support. Thanks also to the anonymous VLDB review committee for their valuable comments and suggestions.

# REFERENCES


[1] 2012. SNAP Datasets. http://snap.stanford.edu/data

[2] 2019. etcd. https://github.com/etcd-io/etcd

[3] 2020. Plato. https://github.com/Tencent/plato

[4] 2021. GAIA: A System for Interactive Analysis on Distributed Graphs Using a High-Level Language. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association. https://www.usenix.org/conference/nsdi21/presentation/qian-0

[5] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*. 265–283.

[6] Christopher R. Aberger, Susan Tu, Kunle Olukotun, and Christopher Ré. 2016. EmptyHeaded: A Relational Engine for Graph Processing. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. ACM, New York, NY, USA, 431–446.

[7] Khaled Ammar, Frank McSherry, Semih Salihoglu, and Manas Joglekar. 2018. Distributed Evaluation of Subgraph Queries Using Worst-Case Optimal Low-Memory Dataflows. *Proc. VLDB Endow.* 11, 6 (Feb. 2018), 691–704.

[8] Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan Reutter, and Domagoj Vrgoč. 2017. Foundations of Modern Query Languages for Graph Databases. *ACM Comput. Surv.* 50, 5, Article 68 (Sept. 2017), 40 pages. https://doi.org/10.1145/3104031

[9] Renzo Angles and Claudio Gutierrez. 2008. Survey of graph database models. *ACM Computing Surveys (CSUR)* 40, 1 (2008), 1–39.

[10] Bradley R Bebee, Daniel Choi, Ankit Gupta, Andi Gutmans, Ankesh Khandelwal, Yigit Kiran, Sainath Mallidi, Bruce McGaughy, Mike Personick, Karthik Rajan, et al. 2018. Amazon Neptune: Graph Data Management in the Cloud.. In *International Semantic Web Conference (P&D/Industry/BlueSky)*.

[11] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. 2008. Fast unfolding of communities in large networks. *Journal of statistical mechanics: theory and experiment* 2008, 10 (2008), P10008.

[12] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. 1995. Cilk: An Efficient Multithreaded Runtime System. *SIGPLAN Not.* 30, 8 (Aug. 1995), 207–216. https://doi.org/10.1145/209937.209958

[13] Chiranjeeb Buragohain, Knut Magne Risvik, Paul Brett, Miguel Castro, Wonhee Cho, Joshua Cowhig, Nikolas Gloy, Karthik Kalyanaraman, Richendra Khanna, John Pao, et al. 2020. A1: A distributed in-memory graph database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 329–344.

[14] Hongzhi Chen, Bowen Wu, Shiyuan Deng, Chenghuan Huang, Changji Li, Yichao Li, and James Cheng. 2020. High Performance Distributed OLAP on Property Graphs with Grasper. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2705–2708.

[15] Dask Development Team. 2016. *Dask: Library for dynamic task scheduling*. https://dask.org

[16] Wenfei Fan, Ruochun Jin, Muyang Liu, Ping Lu, Xiaojian Luo, Ruiqi Xu, Qiang Yin, Wenyuan Yu, and Jingren Zhou. 2020. Application driven graph partitioning. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1765–1779.

[17] Wenfei Fan, Chao Tian, Qiang Yin, Ruiqi Xu, Wenyuan Yu, and Jingren Zhou. 2021. Incrementalizing Graph Algorithms. In *Proceedings of the 2021 ACM SIGMOD International Conference on Management of Data*. 2705–2708.

[18] 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 International Conference on Management of Data*. 495–510.

[19] Wenfei Fan, Wenyuan Yu, Jingbo Xu, Jingren Zhou, Xiaojian Luo, Qiang Yin, Ping Lu, Yang Cao, and Ruiqi Xu. 2018. Parallelizing Sequential Graph Computations. *ACM Trans. Database Syst.* 43, 4, Article 18 (Dec. 2018), 39 pages. https://doi.org/10.1145/3282488

[20] Steven Fortune and James Wyllie. 1978. Parallelism in random access machines. In *Proceedings of the tenth annual ACM symposium on Theory of computing*. 114–118.

[21] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. 2018. Cypher: An evolving query language for property graphs. In *Proceedings of the 2018 International Conference on Management of Data*. 1433–1445.

[22] Michael L. Fredman and Robert Endre Tarjan. 1987. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM* 34, 3 (1987), 596–615.

[23] Linton C Freeman. 1977. A set of measures of centrality based on betweenness. *Sociometry* (1977), 35–41.

[24] Per Fuchs, Peter Boncz, and Bogdan Ghit. 2020. EdgeFrame: Worst-Case Optimal Joins for Graph-Pattern Matching in Spark. In *Proceedings of the 3rd Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)* (Portland, OR, USA) *(GRADES-NDA'20)*. Association for Computing Machinery, New York, NY, USA, Article 4, 11 pages. https://doi.org/10.1145/3398682.3399162

[25] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI*.

[26] Joseph E Gonzalez, Reynold S Xin, Ankur Dave, Daniel Crankshaw, Michael J Franklin, and Ion Stoica. 2014. Graphx: Graph processing in a distributed dataflow framework. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*. 599–613.

[27] Hadoop-Gremlin. 2020. https://tinkerpop.apache.org/docs/3.4.6/reference/#hadoop-gremlin

[28] Aric Hagberg, Pieter Swart, and Daniel S Chult. 2008. *Exploring network structure, dynamics, and function using NetworkX*. Technical Report. Los Alamos National Lab.(LANL), Los Alamos, NM (United States).

[29] Daniel S. Hirschberg, Ashok K. Chandra, and Dilip V. Sarwate. 1979. Computing connected components on parallel computers. *Commun. ACM* 22, 8 (1979), 461–464.

[30] Alexandru Iosup, Tim Hegeman, Wing Lung Ngai, Stijn Heldens, Arnau Prat-Pérez, Thomas Manhardto, Hassan Chafio, Mihai Capotă, Narayanan Sundaram, Michael Anderson, et al. 2016. LDBC Graphalytics: A benchmark for large-scale graph analysis on parallel and distributed platforms. *PVLDB* 9, 13 (2016), 1317–1328.

[31] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. 2007. Dryad: distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*. 59–72.

[32] JanusGraph. 2021. https://github.com/JanusGraph/janusgraph

[33] George Karypis and Vipin Kumar. 1998. Multilevelk-Way Partitioning Scheme for Irregular Graphs. *JPDC* 48, 1 (1998), 96–129.

[34] Arijit Khan. 2016. Vertex-Centric Graph Processing: The Good, the Bad, and the Ugly. *CoRR* abs/1612.07404 (2016). arXiv:1612.07404 http://arxiv.org/abs/1612.07404

[35] Longbin Lai, Lu Qin, Xuemin Lin, Ying Zhang, Lijun Chang, and Shiyu Yang. 2016. Scalable Distributed Subgraph Enumeration. *Proc. VLDB Endow.* 10, 3 (November 2016), 217–228. https://doi.org/10.14778/3021924.3021937

[36] Longbin Lai, Zhu Qing, Zhengyi Yang, Xin Jin, Zhengmin Lai, Ran Wang, Kongzhang Hao, Xuemin Lin, Lu Qin, Wenjie Zhang, Ying Zhang, Zhengping Qian, and Jingren Zhou. 2019. Distributed Subgraph Matching on Timely Dataflow. *Proc. VLDB Endow.* 12, 10 (June 2019), 1099–1112. https://doi.org/10.14778/3339490.3339494

[37] Chuck Lam. 2010. *Hadoop in action*. Manning Publications Co.

[38] OrientDB library. 2020. https://github.com/orientechnologies/orientdb

[39] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. 135–146.

[40] Víctor Martínez, Fernando Berzal, and Juan Carlos Cubero Talavera. 2017. A Survey of Link Prediction in Complex Networks. *ACM Comput. Surv.* 49, 4 (2017), 69:1–69:33.

[41] Amine Mhedhbi and Semih Salihoglu. 2019. Optimizing Subgraph Queries by Combining Binary and Worst-Case Optimal Joins. *Proc. VLDB Endow.* 12, 11 (July 2019), 1692–1704.

[42] Justin J Miller. 2013. Graph database applications and concepts with Neo4j. In *Proceedings of the Southern Association for Information Systems Conference, Atlanta, GA, USA*, Vol. 2324.

[43] Hung Q Ngo, Christopher Ré, and Atri Rudra. 2014. Skew Strikes Back: New Developments in the Theory of Join Algorithms. *SIGMOD Rec.* 42, 4 (Feb. 2014), 5–16.

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

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

[46] Mars Project. 2020. https://github.com/mars-project/mars

[47] Lu Qin, Longbin Lai, Kongzhang Hao, Zhongxin Zhou, Yiwei Zhao, Yuxing Han, Xuemin Lin, Zhengping Qian, and Jingren Zhou. 2020. Taming the Expressiveness and Programmability of Graph Analytical Queries. *arXiv preprint arXiv:2004.09045* (2020).

[48] Xiafei Qiu, Wubin Cen, Zhengping Qian, You Peng, Ying Zhang, Xuemin Lin, and Jingren Zhou. 2018. Real-Time Constrained Cycle Detection in Large Dynamic Graphs. *Proc. VLDB Endow.* 11, 12 (Aug. 2018), 1876–1888. https://doi.org/10.14778/3229863.3229874

[49] Usha Nandini Raghavan, Réka Albert, and Soundar Kumara. 2007. Near linear time algorithm to detect community structures in large-scale networks. *Physical review E* 76, 3 (2007), 036106.

[50] Rob Reagan. 2018. Cosmos DB. In *Web Applications on Azure*. Springer, 187–255.

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

*Languages*. 1–10.

[52] Christopher J. Rossbach, Yuan Yu, Jon Currey, Jean-Philippe Martin, and Dennis Fetterly. 2013. Dandelion: A Compiler and Runtime for Heterogeneous Systems. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania) *(SOSP '13)*. Association for Computing Machinery, New York, NY, USA, 49–68. https://doi.org/10.1145/2517349.2522715

[53] Bin Shao, Haixun Wang, and Yatao Li. 2013. Trinity: A distributed graph engine on a memory cloud. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. 505–516.

[54] George M Slota, Sivasankaran Rajamanickam, Karen Devine, and Kamesh Madduri. 2017. Partitioning trillion-edge graphs in minutes. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 646–655.

[55] The HIVE project. 2020. http://hadoop.apache.org/hive/. [Online; accessed 17-September-2020].

[56] Apache TinkerPop. 2019. http://tinkerpop.apache.org/

[57] Charalampos Tsourakakis, Christos Gkantsidis, Bozidar Radunovic, and Milan Vojnovic. 2014. Fennel: Streaming graph partitioning for massive scale graphs. In *WSDM*. 333–342.

[58] Random Walks. 2010. http://en.wikipedia.org/wiki/Random_walk

[59] 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*.

[60] Jizhe Wang, Pipei Huang, Huan Zhao, Zhibo Zhang, Binqiang Zhao, and Dik Lun Lee. 2018. Billion-Scale Commodity Embedding for E-Commerce Recommendation in Alibaba. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (London, United Kingdom) *(KDD '18)*. Association for Computing Machinery, New York, NY, USA, 839–848. https://doi.org/10.1145/3219819.3219869

[61] Ke Yang, MingXing Zhang, Kang Chen, Xiaosong Ma, Yang Bai, and Yong Jiang. 2019. Knightking: a fast distributed graph random walk engine. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 524–537.

[62] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. 2008. DryadLINQ: A System for General-purpose Distributed Data-parallel Computing Using a High-level Language. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (San Diego, California) *(OSDI'08)*. USENIX Association, Berkeley, CA, USA, 1–14. http://dl.acm.org/citation.cfm?id=1855741.1855742

[63] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. USENIX Association, San Jose, CA, 15–28. https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/zaharia

[64] Yunhao Zhang, Rong Chen, and Haibo Chen. 2017. Sub-millisecond stateful stream querying over fast-evolving linked data. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 614–630.

[65] 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.

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