Declarative and distributed graph analytics with GRADOOP

Martin Junghanns
University of Leipzig &
ScaDS Dresden/Leipzig

Max Kießling
University of Leipzig &
ScaDS Dresden/Leipzig

Niklas Teichmann
University of Leipzig &
ScaDS Dresden/Leipzig

Kevin Gómez
University of Leipzig &
ScaDS Dresden/Leipzig

André Petermann
University of Leipzig &
ScaDS Dresden/Leipzig

Erhard Rahm
University of Leipzig &
ScaDS Dresden/Leipzig

ABSTRACT

We demonstrate Gradoop, an open source framework that combines and extends features of graph database systems with the benefits of distributed graph processing. Using a rich graph data model and powerful graph operators, users can declaratively express graph analytical programs for distributed execution without needing advanced programming experience or a deeper understanding of the underlying system. Visitors of the demo can declare graph analytical programs using the Gradoop operators and also visually experience two of our advanced operators: graph pattern matching and graph grouping. We provide real world and artificial social network data with up to 10 billion edges and allow running the programs either locally or on a remote research cluster to demonstrate scalability.

PVLDB Reference Format:
Martin Junghanns, Max Kießling, Niklas Teichmann, Kevin Gómez, André Petermann and Erhard Rahm. Declarative and distributed graph analytics with GRADOOP. *PVLDB*, 11 (12): 2006-2009, 2018.
DOI: https://doi.org/10.14778/3229863.3236246

1. INTRODUCTION

Currently, two major categories of systems focus on the management and analysis of graph data: graph database systems and distributed graph processing systems [7]. Graph database systems, such as Neo4j, focus on the efficient storing and transactional processing of graph data where multiple users can access a graph in an interactive way. They support expressive data models, such as the property graph model [13], which are suitable to represent heterogeneous graph data (see Figure 1). Furthermore, graph database systems often provide a declarative graph query language, e.g., Cypher [11], with support for graph traversals and pattern matching. However, graph database systems are typically less suited for high-volume data analysis and graph mining [4, 10, 14] and often do not support distributed processing on partitioned graphs which limits the maximum graph size to the resources of a single machine.

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/4.0/. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org.
Proceedings of the VLDB Endowment, Vol. 11, No. 12
Copyright 2018 VLDB Endowment 2150-8097/18/8.
DOI: https://doi.org/10.14778/3229863.3236246

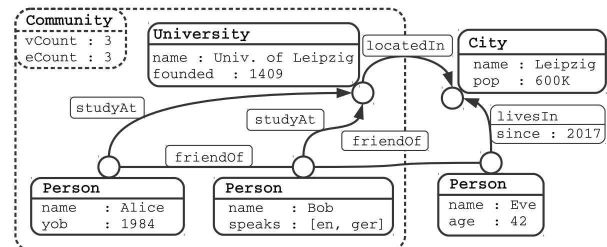

Figure 1: Heterogeneous social network with custom labels and properties on vertices, edges and graphs.

By contrast, parallel graph processing systems [1, 7] such as Google Pregel [9] or GraphX [15] process and analyze large-scale graph data in-memory on shared nothing clusters. They provide tailored computational models in which users need to implement predefined functions to express graph algorithms. However, these systems typically lack an expressive graph data model with support for heterogeneous entities and declarative graph operations. Especially the latter makes it difficult for users to formulate analytical tasks as this requires profound programming and system knowledge.

The comparison shows that the two categories have both strengths and restrictions. To combine and extend the former, we built Gradoop, our full-fledged framework for distributed, declarative graph analytics.[1] Gradoop is the first system that provides support for multiple, heterogeneous graphs, distributed graph querying and distributed graph algorithms within a single system. In addition, Gradoop provides several powerful graph operators to help data scientists, who are often non-professional programmers and rely on domain-specific abstractions to express complex analytical tasks. The main contributions of Gradoop are:

- The Extended Property Graph Model [5] to support heterogeneous graphs as well as collections of those.
- Composable graph operators, including analytical operators such as Cypher-based graph pattern matching [6] and graph grouping [8] as well as general operators for data transformation and aggregation.
- Integration of graph algorithms, e.g., page rank, community detection and frequent subgraph mining [12].
- Support for batch-oriented program execution and interactive operator execution including result visualization.

In the remainder we will briefly introduce the Gradoop system (Sec. 2) and describe our demonstration (Sec. 3).

[1] `www.gradoop.com`

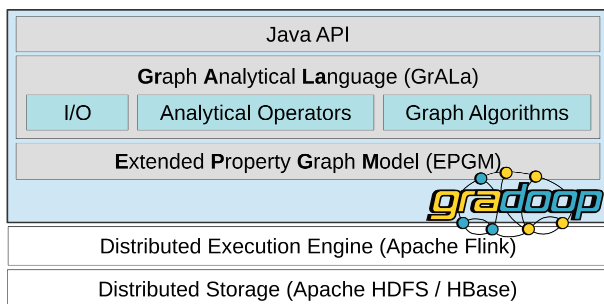

Figure 2: High-level architecture of the Gradoop system.

2. THE GRADOOP SYSTEM

With Gradoop, we provide an open source framework for scalable analytics of large, semantically expressive graphs. To achieve horizontal scalability of storage and processing capacity, Gradoop runs on shared nothing clusters and utilizes existing open source frameworks for distributed data storage and processing. The difficulties of distributing data and computation are hidden beneath a graph abstraction allowing the user to focus on the problem domain.

Figure 2 shows the Gradoop architecture. Analytical programs are defined within our **Gr**aph **A**nalytical **La**nguage (GrALa), which is a domain specific language for the **E**xtended **P**roperty **G**raph **M**odel (EPGM) [5]. GrALa contains operators for accessing graphs in the underlying storage as well as for applying graph operations and analytical graph algorithms to them. Operator execution is carried out by the distributed execution engine which spreads the computation across the available machines. When the computation of an analytical program is completed, results may be either written back to the storage layer or presented to the user. In the following, we briefly explain the core components.

Distributed Storage. Gradoop supports several ways to store EPGM compliant graph data. To abstract the specific storage, GrALa offers two interfaces: *DataSource* to read and *DataSink* to write graphs. An analytical program typically starts with one or more data sources and ends in at least one data sink (see Fig. 3). We provide several implementations for file-based storage, e.g., CSV and JSON, that allow reading and writing graphs from the local file system or the Apache Hadoop Distributed File System (HDFS). Gradoop in addition supports Apache HBase, that introduces database capabilities on top of the HDFS.

Distributed Execution Engine. Within Gradoop, the EPGM and GrALa provide a graph abstraction for the user. However, the actual implementation of the data model and its operators are transparent to the user and hidden within the distributed execution engine. Generally, this can be an arbitrary data management system that allows implementing graph operators. Gradoop utilizes a distributed dataflow system to achieve horizontal scalability. Well-known systems are Apache Spark [16] and Apache Flink [2]. Those systems provide high-level APIs that enable fast application development by abstracting from the complexities of distributed computing. For our current proof of concept, we chose Apache Flink, a distributed batch and stream processing framework, that allows executing arbitrary dataflow programs in a data-parallel and distributed manner. A dataflow system provides two fundamental programming abstractions: *datasets* and *transformations* among them. A dataset is a collection of arbitrary data objects partitioned over a cluster of machines. A transformation is a parallel operation that is executed on the elements of one or two input datasets and produces a new dataset. Well-known transformations are *map* and *reduce*, but also relational operators are supported, e.g., *selection* (*filter*), *join* and *grouping*. Application logic is expressed by user-defined functions which are arguments of transformations and applied to dataset elements during execution. The dataflow system handles data distribution, load balancing and failure management. In addition, Apache Flink provides several libraries that can be combined and integrated within a Gradoop program, e.g., for graph processing, machine learning and SQL.

Table 1: Analytical graph operators and algorithms available in Gradoop organized by their input type, i.e., logical graph or graph collection. (* auxiliary operators)

	Analytical Operators		Graph Algorithms
	Unary	**Binary**	
Logical Graph	Aggregation Pattern Matching [6] Transformation Grouping [8] Subgraph *Call**	Combination Overlap Exclusion Equality	PageRank Community Detection Connected Components Single Source Shortest Path Summarization Hyperlink-Induced Topic Search
Graph Coll.	Selection Distinct Limit *Apply** *Reduce** *Call**	Union Intersection Difference Equality	Frequent Subgraph Mining [12]

Extended Property Graph Model. The EPGM [5] describes how graphs and their elements (vertices and edges) are structured within Gradoop. It is an extension of the property graph model [13], which is used in various graph database systems. To facilitate integration of heterogeneous data, it does not enforce any kind of schema, but the graph elements can have different type labels and attributes (see Figure 1). The latter are exposed to the user and can be accessed within graph operators. For enhanced analytical expressiveness, the EPGM supports handling of multiple, possibly overlapping graphs within a single analytical program [5]. Graphs, as well as vertices and edges, are first-class citizens of the data model and can have their own properties. Within Gradoop, EPGM elements are represented through specific data types and organized within datasets of the underlying dataflow system. The concrete organization is determined by a graph layout, for example, the default layout maps a single graph to three datasets. Gradoop already provides a set of layouts that are beneficial within specific analytical scenarios.

Graph Analytical Language. Programs are specified using declarative GrALa operators. These operators can be composed as they are closed over the EPGM, i.e., take graphs as input and produce graphs. From a user perspective, graphs and graph operators are the analogy to datasets and

```java
// init Flink execution environment
ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment();
// create default Gradoop config
GradoopFlinkConfig config = GradoopFlinkConfig.createConfig(env);
// create DataSource
JSONDataSource dataSource = new JSONDataSource(inputDir, config);
// lazily read the graph from the DataSource
LogicalGraph socialNetwork = dataSource.getLogicalGraph();
// perform analytics
LogicalGraph resultGraph = analyze(socialNetwork);
// create DataSink
CSVDataSink dataSink = new CSVDataSink(outputDir, config);
// write result in CSV format
dataSink.write(resultGraph);
// trigger program execution
env.execute();
```

Figure 3: GrALa program to read and write graph data.

transformations. Table 1 shows available analytical operators and graph algorithms categorized by their input [5]. Besides general operators for graph transformation or aggregation, Gradoop also provides *pattern matching* [6] capabilities known from graph database systems and analytical operators, e.g., for *graph grouping* [8]. Each category contains auxiliary operators, e.g., to *apply* unary graph operators on each graph in a graph collection or to *call* external algorithms. GrALa already integrates a set of well-known graph algorithms (e.g., page rank or connected components), which can be seamlessly integrated into an operator composition. Within Gradoop, an operator call is mapped to a series of transformations on the underlying datasets representing the graph or the graph collection on which it is called. For example, calling the grouping operator on a single graph performs map, filter, grouping and join transformations on its vertices and edges, whereas the pattern matching operator abstracts a complete cost-based query engine that - depending on the search pattern - generates individual compositions of dataset transformations.

Application Programming Interface. Gradoop offers a Java API containing the EPGM abstraction including all operators defined within GrALa. This way, users can specify analytical programs (see Figures 3 and 4) and execute them either locally for testing or on a cluster. Graphs can optionally be initialized from existing datasets which allows for pre-processing data within the dataflow system. Graphs also expose the underlying datasets, which enables post-processing using any available library provided by the dataflow system.

Performance evaluation. We already evaluated Gradoop and its operators extensively in previous publications. Our benchmarks were performed on a shared nothing cluster using artificial [3] and real-world heterogeneous graphs with up to 10 billion edges. In [6], we demonstrated that our Cypher-based pattern matching operator achieves good scalability for increasing computing resources and near perfect scalability for increasing data sets. The graph grouping operator, presented in [8], shows a similar behavior and achieved runtimes of a few seconds on real-world data. A benchmark for a comprehensive GrALa program including several Gradoop operators and a community detection algorithm was conducted in [5] and achieved near-linear scalability. For our implementation of Frequent Subgraph Mining [12], we were also able to show high scalability for increasing data sets, decreasing minimum support thresholds and increasing computing resources.

```java
return socialNetwork
  // 1) extract subgraph using vertex and edge predicates
  .subgraph(
    vertex -> vertex.getLabel().toLowerCase().equals(person),
    edge -> edge.getLabel().toLowerCase().equals(knows))
  // 2) project elements to necessary information
  .transform(
    // keep graph label and all properties
    TransformationFunction.keep(),
    // keep necessary vertex properties
    (current, transformed) -> {
      transformed.setLabel(current.getLabel());
      transformed.setProperty(city, current.getPropertyValue(city));
      transformed.setProperty(gender, current.getPropertyValue(gender));
      transformed.setProperty(label, current.getPropertyValue(birthday));
      return transformed;
    },
    // keep only edge label
    (current, transformed) -> {
      transformed.setLabel(current.getLabel());
      return transformed;
    })
  // 3a) compute communities via label propagation
  .callForGraph(new GellyLabelPropagation(maxIterations, label))
  // 3b) separate communities into a graph collection
  .splitBy(label)
  // 4) compute number of vertices per community
  .apply(new ApplyAggregation(new VertexCount()))
  // 5) select communities with a minimum number of vertices
  .select(g -> g.getPropertyValue(vertexCount).getLong() > threshold)
  // 6) combine selected communities to a single graph
  .reduce(new ReduceCombination())
  // 7) group that graph by vertex properties
  .groupBy(Lists.newArrayList(city, gender))
  // 8) count vertices of summary graph
  .aggregate(new VertexCount())
  // 9) count edges of summary graph
  .aggregate(new EdgeCount());
```

Figure 4: GrALa program to analyze a social network.

3. DEMONSTRATION DESCRIPTION

Our demonstration is separated into two parts. First, we want to give visitors the opportunity to inspect and manipulate existing example GrALa programs written in the Java programming language. All example programs can be executed on demo data, but also remotely on our research cluster. We provide real-world graphs and artificial social network data with up to 10 billion edges generated by the LDBC data generator [3]. The second part of the demonstration focuses on the analytical value of two of our graph operators for which we created a browser-based user interface. Here, the operators can be parameterized and executed, the result is presented to the user by graph visualization.

Programmatic demonstration. Gradoop already comes with a set of example programs[2] that show basic functionality like reading and writing graphs from different data sources and sinks (see Figure 3) as well as the application of basic graph operators. Advanced examples, like the one shown in Figure 4, demonstrate the composition of multiple operators to answer a specific analytical question. In the example, we analyze a social network by extracting relevant information using the *subgraph* and *transformation* operators, applying a community detection algorithm (label propagation) and using *aggregation* to compute the number of users per community on which we *select* those that are above a given threshold. Finally, we compute a summary graph by *grouping* the *combined* communities based on user attributes.

The example illustrates the abstraction level of a Gradoop program. A user does not need to be concerned about graph data structures, operator implementations and distributed execution details. The very same program can be executed

[2] https://git.io/vbAbr

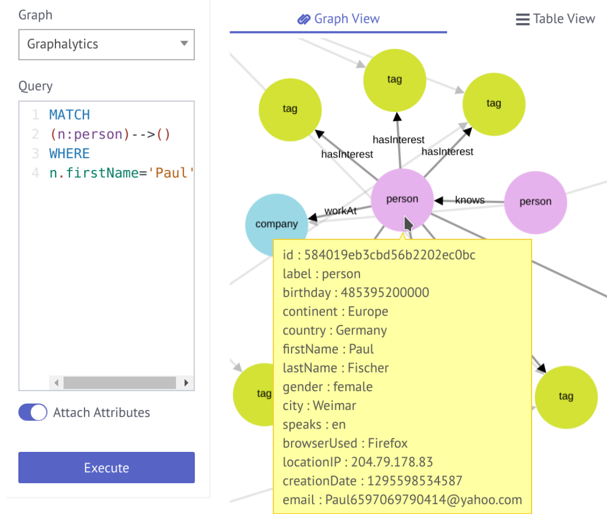

Figure 5: Computing vertex neighborhoods via Cypher.

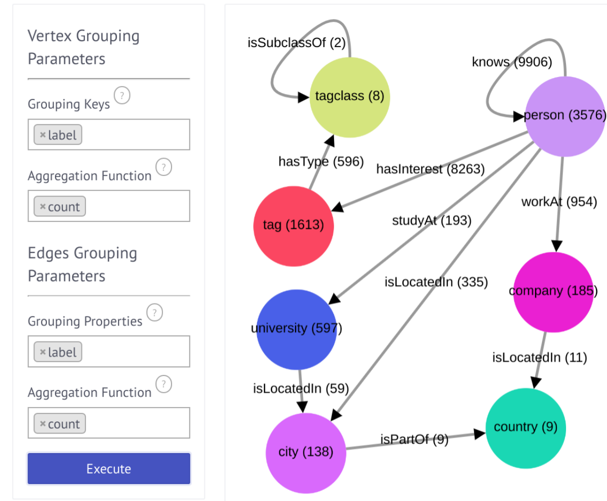

Figure 6: Computing a summary graph via grouping.

locally for tests but also unmodified on a cluster using large-scale datasets. Visitors will be able to execute those examples, manipulate them and write their own analytical programs using all GrALa operators.

Visual demonstration. To illustrate the analytical value of graph pattern matching and graph grouping, we demonstrate these two operators using a web application[3].

Graph pattern matching is applied to find specific patterns within a logical graph [6]. To declare a query, we adopted core features of Cypher [11], the graph query language of Neo4j. Cypher uses so called ASCII-art to describe a pattern, for example, in the social network of Figure 1, we want to look for friends that study at the same University. In Cypher, this can be expressed using the following query:

```
MATCH (p1:Person)-[:friendOf]-(p2:Person),
      (p1)-[:studyAt]->(u:University)<-[:studyAt]-(p2)
RETURN p1, p2
```

The pattern matching operator is parameterized with a Cypher query and applied on a logical graph. The result of that operation is a graph collection in which each graph represents a subgraph of the input graph that matches the pattern. Figure 5 shows our demo application in which we query the neighborhood of all persons named *Paul*. The resulting graphs can be either visualized or displayed in a tabular view. In the demonstration, users can explore the graphs by writing Cypher queries and executing them on our demo data. We will show more complex queries including path searches and complex predicates.

Graph grouping is applied to reduce the complexity of large-scale graphs by summarizing the graph structure by user-defined vertex and edge properties [8]. This way, a user is able to gain insights that are not derivable by looking at the raw data. A simple example is shown in Figure 6. Here, we group vertices and edges by their label to find out what is connected and how. Each vertex and edge in the summary graph represents a group of vertices and edges of the input graph. During grouping, a user can apply several aggregate functions (e.g., count, min, max) whose results are stored as new vertex and edge properties. In the demonstration, we will show more examples involving multiple grouping keys, label-dependent grouping as well as graph roll up and drill down.

[3]`https://github.com/dbs-leipzig/gradoop_demo`

4. REFERENCES

[1] O. Batarfi et al. Large scale graph processing systems: survey and an experimental evaluation. *Cluster Computing*, 18(3):1189–1213, 2015.

[2] P. Carbone et al. Apache Flink: Stream and Batch Processing in a Single Engine. *IEEE Data Eng. Bull.*, 38(4):28–38, 2015.

[3] O. Erling et al. The LDBC social network benchmark: Interactive workload. In *Proc. SIGMOD*, 2015.

[4] Y. Guo et al. How Well Do Graph-Processing Platforms Perform? An Empirical Performance Evaluation and Analysis. In *Proc. IPDPS*, pages 395–404, 2014.

[5] M. Junghanns et al. Analyzing Extended Property Graphs with Apache Flink. In *Proc. SIGMOD NDA Workshop*, 2016.

[6] M. Junghanns et al. Cypher-based Graph Pattern Matching in Gradoop. In *Proc. SIGMOD GRADES Workshop*, 2017.

[7] M. Junghanns et al. Management and Analysis of Big Graph Data: Current Systems and Open Challenges. In *Handbook of Big Data Technologies*, pages 457–505. 2017.

[8] M. Junghanns, A. Petermann, and E. Rahm. Distributed Grouping of Property Graphs with GRADOOP. In *Proc. BTW*, pages 103–122, 2017.

[9] G. Malewicz et al. Pregel: A System for Large-scale Graph Processing. In *Proc. SIGMOD*, pages 135–146, 2010.

[10] R. McColl et al. A Performance Evaluation of Open Source Graph Databases. In *Proc. PPAA*, pages 11–18, 2014.

[11] openCypher. Cypher Query Lang. Ref. `https://github.com/opencypher/openCypher/blob/master/docs/openCypher9.pdf`, accessed: Feb. 2018.

[12] A. Petermann et al. DIMSpan - Transactional Frequent Subgraph Mining with Distributed In-Memory Dataflow Systems. In *Proc. BDCAT*, pages 237–246, 2017.

[13] M. A. Rodriguez and P. Neubauer. Constructions from dots and lines. *ASIS&T Bull.*, 36(6), 2010.

[14] B. Shao, H. Wang, and Y. Xiao. Managing and Mining Large Graphs: Systems and Implementations. In *Proc. SIGMOD*, pages 589–592, 2012.

[15] R. S. Xin et al. GraphX: A Resilient Distributed Graph System on Spark. In *Proc. SIGMOD GRADES Workshop*, 2013.

[16] M. Zaharia et al. Apache Spark: A Unified Engine for Big Data Processing. *Commun. ACM*, 59(11):56–65, 2016.