B. Mitschang et al. (Hrsg.): Datenbanksysteme für Business, Technologie und Web (BTW 2017),
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2017 103

Distributed Grouping of Property Graphs with Gradoop

Martin Junghanns[1] André Petermann[2] Erhard Rahm[3]

Abstract: Property graphs are an intuitive way to model, analyze and visualize complex relationships among heterogeneous data objects, for example, as they occur in social, biological and information networks. These graphs typically contain thousands or millions of vertices and edges and their entire representation can easily overwhelm an analyst. One way to reduce complexity is the grouping of vertices and edges to summary graphs. In this paper, we present an algorithm for graph grouping with support for attribute aggregation and structural summarization by user-defined vertex and edge properties. The algorithm is part of Gradoop, an open-source system for graph analytics. Gradoop is implemented on top of Apache Flink, a state-of-the-art distributed dataflow framework, and thus allows us to scale graph analytical programs across multiple machines. Our evaluation demonstrates the scalability of the algorithm on real-world and synthetic social network data.

Keywords: Graph Analytics, Graph Algorithms, Distributed Computing, Dataflow systems

1 Introduction

Graphs are a simple, yet powerful data structure to model and to analyze relationships among real-world data objects. The flexibility of graph data models and the variety of existing graph algorithms made graph analytics attractive to different domains, e.g., to analyze the world wide web or social networks but also for business intelligence and the life sciences [Ne10, Ma10, Pa11, Pe14a]. In a graph, entities like web sites, users, products or proteins can be modeled as vertices while their connections are represented by edges.

Real-world graphs are often heterogeneous in terms of the objects they represent. For example, vertices of a social network may represent users and forums while edges may express friendships or memberships. Further on, vertices and edges may have associated properties to describe the respective object, e.g., a user's age or the date a user became member of a forum. Property graphs [RN10, An12] are an established approach to express this kind of heterogeneity. Figure 1(a) shows a property graph that represents a simple social network containing multiple types of vertices (e.g., *User* and *Forum*) and edges (e.g., *follows* and *memberOf*). Vertices as well as edges are further described by properties in the form of key-value pairs (e.g., *name : Alice* or *since : 2015*). However, while small graphs are an intuitive way to visualize connected information, with vertex and edge numbers increasing up to millions and billions, it becomes almost impossible to understand the encoded information by mere visual inspection. Therefore, it is essential to provide graph grouping methods that reduce complexity and support analysts in extracting and understanding the underlying information [THP08, Ch08, Zh11]. The following examples highlight the analytical value of such methods:

[1] University of Leipzig, Database Group & ScaDS Dresden/Leipzig, junghanns@informatik.uni-leipzig.de
[2] University of Leipzig, Database Group & ScaDS Dresden/Leipzig, petermann@informatik.uni-leipzig.de
[3] University of Leipzig, Database Group & ScaDS Dresden/Leipzig, rahm@informatik.uni-leipzig.de

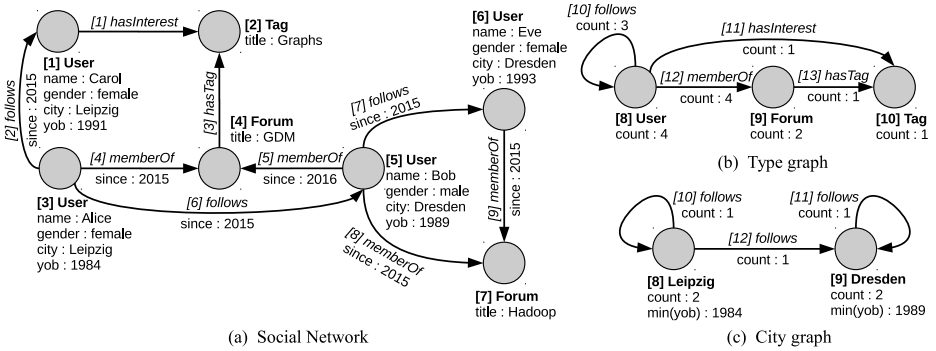


Fig. 1: (a) shows an example social network as input of the grouping operator; (b) shows the vertices and edges of (a) grouped according to their label including count aggregates stored in respective properties at the resulting vertices and edges; (c) shows the subgraph containing users and their mutual relationships grouped by users' location and edge labels including aggregate values expressing the oldest user's age per location and the number of edges among locations.

Example 1: Type Graph A simple graph analytical question is: *"Which types of entities are contained and how are they connected?"*. With regard to our social network example, the answer is shown in Figure 1(b). Here, each vertex represents a group of vertices from the original graph that share the same type. For example, vertex 8 represents the users *Alice*, *Carol*, *Bob* and *Eve*, while vertex 9 represents the forums *GDM* and *Hadoop*. Edges are grouped according to their incident vertices and their type, e.g., edge 11 represents all memberships among users and forums in the original graph. Furthermore, each vertex and edge in the summary graph stores the number of elements it represents as a new property. The resulting graph provides an overview of the underlying network and, thus, is a good starting point for more detailed analyses.

Example 2: City Graph In this example we want to group users by the city they live in, calculate the number of group members and find the smallest year of birth (yob) per group. Edges shall be grouped by their type and also being counted. To achieve this, we need to group users by the property *city* and aggregate each of these groups using the *yob* property. The resulting summary graph is shown in Figure 1(c) and reveals that our social network includes users from Leipzig and Dresden whereas the oldest person lives in Leipzig. This high level view further shows how relationships are distributed among groups.

The examples demonstrate the value of summary graphs to gain useful insights into large networks. Note that complex graph analytical questions often require the combination of multiple algorithms, e.g., Example 2 requires extracting a subgraph containing only users and their mutual relationships and replacing vertex labels by a certain property value before the summary graph can be computed.

To support analyses combining multiple techniques, we started developing Gradoop, a graph analytical system that implements the Extended Property Graph Model (EPGM) [Ju16]. The data model defines operators that can be combined to complex analytical programs on single property graphs and graph collections. Gradoop is open-source[4] and

[4] http://www.gradoop.com

built on top of Apache Flink [Al14, Ca15b], a scalable dataflow framework. Thus, the execution of the provided operators can be distributed across a cluster of machines. To our knowledge, no other distributed graph analytics framework provides a similar grouping functionality. Our main contributions can be summarized as follows:

- We formally introduce the grouping operator to flexibly compute summaries of property graphs by user-defined properties and aggregation functions.
- We discuss various analytical scenarios combining the grouping operator with other graph operators provided by the Extended Property Graph Model.
- We describe the implementation of the grouping operator in the context of Gradoop, our system for graph analytics on top of Apache Flink. We additionally introduce an optimization for unbalanced workloads.
- We present experimental results to evaluate the scalability of our implementation by applying the operator to real-world and synthetic social network data.

In Section 2, we provide the theoretical foundation of graph grouping in the context of the Extended Property Graph Model and discuss different application scenarios. Afterwards in Section 3, we describe the implementation of graph grouping utilizing operators of Apache Flink. The experimental evaluation is presented in Section 4. Finally, we discuss related work in Section 5 and conclude our work in Section 6.

2 Graph Grouping in the Extended Property Graph Model

First, we briefly introduce the used EPGM graph data model. Based thereon, we then formally define graph grouping and introduce the grouping operator in GrALa, our EPGM-specific DSL for graph analytics. We will further show how graph grouping is used to construct summary graphs either stand-alone or in combination with other graph operators.

2.1 Extended Property Graph Model

Gradoop is based on a semantically rich, schema-less graph data model called *Extended Property Graph Model* (EPGM) [Ju16]. In this model, a graph database consists of multiple possibly overlapping property graphs which are referred to as *logical graphs*. Vertices, edges and logical graphs have a type label (e.g., `User`, `memberOf` or `Community`) and may have an arbitrary set of attributes represented by key-value-pairs (e.g., `name : Alice`). Formally, an EPGM database is defined as follows:

Definition 1 *(EPGM Database). An EPGM database* $DB = \langle \mathcal{V}, \mathcal{E}, \mathcal{L}, K, T, A, \kappa \rangle$ *consists of vertex set* $\mathcal{V} = \{v_i\}$*, edge set* $\mathcal{E} = \{e_k\}$ *and a set of logical graphs* $\mathcal{L} = \{G_m\}$*. Vertices, edges and (logical) graphs are identified by the respective indices* $i, k, m \in \mathbb{N}$*. An edge* $e_k = \langle v_i, v_j \rangle$ *with* $v_i, v_j \in \mathcal{V}$ *directs from* v_i *to* v_j *and supports loops (i.e.,* $i = j$*). There can be multiple edges between two vertices differentiated by distinct identifiers. A* ***logical graph*** $G_m = \langle V_m, E_m \rangle$ *is an ordered pair of a subset of vertices* $V_m \subseteq \mathcal{V}$ *and a subset of edges* $E_m \subseteq \mathcal{E}$ *where* $\forall \langle v_i, v_j \rangle \in E_m : v_i, v_j \in V_m$*. Vertex, edge and graph properties are defined by key set* K*, value set* A *and mapping* $\kappa : (\mathcal{V} \cup \mathcal{E} \cup \mathcal{L}) \times K \rightarrow A$*. For the definition of type labels we use a label alphabet* $T \subseteq A$ *and a dedicated type property key* $\tau \in K$*.*

The EPGM defines a set of expressive operators to analyze logical graphs and collections of these. Since the in- and output of such operators are always logical graphs, the power of the

EPGM is based on the ability to combine multiple operators to graph analytical programs. For example, to achieve the summary shown in Figure 1(c), we have to extract the logical graph containing only vertices of type `User` including their mutual relationships and use the vertex property `city` as a new vertex label before applying the grouping operation. The Gradoop framework already provides operator implementations for graph pattern matching, subgraph extraction, graph transformation, set operations on multiple graphs as well as property-based aggregation and selection [Ju16].

2.2 Graph Grouping

EPGM operators are classified according to their input and output. For example, a *unary graph operator* takes a single logical graph as input and outputs either a new logical graph or a graph collection. Graph grouping belongs to the former ones as it outputs a single logical graph, which we call a *summary graph*:

Definition 2 *(Graph Grouping). For a given logical graph $G(V, E)$, a non-empty set of vertex grouping keys $K_v \subseteq K$, a set of edge grouping keys $K_\epsilon \subseteq K$ and sets of aggregate functions Λ_v and Λ_ϵ, the graph grouping operator produces a so-called summary graph $G'(V', E')$ containing super vertices and super edges. The resulting graph and its elements are added to the EPGM database, such that $\mathcal{L} \leftarrow \mathcal{L} \cup \{G'\}$, $\mathcal{V} \leftarrow \mathcal{V} \cup V'$ and $\mathcal{E} \leftarrow \mathcal{E} \cup E'$.*

Definition 3 *(Super Vertex). Let $V(G') = \{v'_1, v'_2, ..., v'_n\}$ be the vertex set of a summary graph G' and $s_v : V(G) \rightarrow V(G')$ a surjective function, then v'_i is called a super vertex and $\forall v \in V(G)$, $s_v(v)$ is the super vertex of v. Vertices in $V(G)$ are grouped based on their property values, such that for a given non-empty set of vertex grouping keys $K_v \subseteq K$, $\forall u, v \in V(G) : s_v(u) = s_v(v) \Leftrightarrow \forall k \in K_v : \kappa(u, k) = \kappa(v, k)$. A super vertex stores the properties representing the group, such that, $\forall k \in K_v, \forall u \in V(G) : \kappa(s_v(u), k) = \kappa(u, k)$.*

Definition 4 *(Super Edge). Let $E(G') = \{e'_1, e'_2, ..., e'_m\}$ be the edge set of a summary graph G' and $s_\epsilon : E(G) \rightarrow E(G')$ a surjective mapping, then e'_i is called a super edge and $\forall \langle u, v\rangle \in E(G), s_\epsilon(\langle u, v\rangle)$ is the super edge of $\langle u, v\rangle$. Edge groups are determined along the super vertices and a set of edge keys $K_\epsilon \subseteq K$, such that $\forall \langle u, v\rangle, \langle s, t\rangle \in E(G) : s_\epsilon(u, v) = s_\epsilon(s, t) \Leftrightarrow s_v(u) = s_v(s) \wedge s_v(v) = s_v(t) \wedge \forall k \in K_\epsilon : \kappa(\langle u, v\rangle, k) = \kappa(\langle s, t\rangle, k)$. Analogous to super vertices, a super edge stores the properties representing the group, such that $\forall k \in K_\epsilon, \forall \langle u, v\rangle \in E(G) : \kappa(\langle s_v(u), s_v(v)\rangle, k) = \kappa(\langle u, v\rangle, k)$.*

Definition 5 *(Aggregates). Additionally, sets of associative and commutative vertex and edge aggregate functions $\Lambda_v = \{\alpha_v : \wp(V(G)) \rightarrow A\}$ and $\Lambda_\epsilon = \{\alpha_\epsilon : \wp(E(G)) \rightarrow A\}$ can be used to compute aggregated property values for super vertices and edges. The resulting value is stored at the respective super entity using a key determined by $f_\alpha : \Lambda_v \cup \Lambda_\epsilon \rightarrow K$, e.g., $\forall \alpha_v \in \Lambda_v, \forall v' \in V(G') : \kappa(v', f_\alpha(\alpha_v)) = \alpha_v(\{u \in V(G) \mid s_v(u) = v'\})$.*

In our introductory example we apply the grouping operator on a graph G representing the social network in Figure 1(a). To compute the summary graph G' of Figure 1(b), we first need to specify two sets of grouping keys. To group vertices and edges by their type label we use the type property key τ such that $K_v = K_\epsilon = \{\tau\}$. Additionally, we define two aggregate functions $\alpha_{v_{count}} : V \mapsto |V|$ and $\alpha_{\epsilon_{count}} : E \mapsto |E|$ and assign them to a property key $f_\alpha(\alpha_{v_{count}}) = f_\alpha(\alpha_{\epsilon_{count}}) =$ count.

The resulting summary graph consists of three super vertices $V(G') = \{v_8, v_9, v_{10}\}$ and four super edges $E(G') = \{v_{10}, v_{11}, v_{12}, v_{13}\}$. Considering vertex v_8 as an example one can see that

it takes the type label `User` of its underlying vertex group, i.e., $V_{v_8} = \{v_1, v_3, v_5, v_6\} \subset V(G)$, and shows an additional property (`count`) that refers to the result of the aggregate function: $\kappa(v_8, f_\alpha(\alpha_{v_{count}})) = \alpha_{v_{count}}(V_{v_8}) = 4$. Edges are grouped by the super vertices of their incident vertices as well as their label. For example, edge e_{10} represents all `follows` edges connecting vertices represented by v_8, i.e., $E_{e_{10}} = \{e_2, e_6, e_7\} \subset E(G)$. On the other hand, edge e_{12} represents all `memberOf` edges pointing from a `User` to a `Forum`. Like super vertices, super edges also store an additional property referring to the result of the aggregate function, e.g., $\kappa(e_{10}, f_\alpha(\alpha_{\epsilon_{count}})) = \alpha_{\epsilon_{count}}(E_{e_{10}}) = 3$.

2.3 Graph Grouping in GrALa

In the remainder of this paper, we will use a domain specific language called GrALa (**Gr**aph **A**nalytical **La**nguage) that has been introduced in [Ju16] to express graph analytical programs in the EPGM.[5] In GrALa, graph operators are higher-order functions that can be parameterized with user-defined functions to express custom logic, e.g., for aggregation or filtering. The operator signature for graph grouping is defined as follows:

```
LogicalGraph.groupBy(
  vertexGroupingKey[], vertexAggregateFunction[],
  edgeGroupingKey[], edgeAggregateFunction[]) : LogicalGraph
```

While the first argument is a list of vertex grouping keys $K_\nu \subseteq K$, the second argument refers to a list of user-defined vertex aggregate functions Λ_ν. Analogously, the third and fourth argument are used to define edge grouping keys and edge aggregate functions. The operator returns a new logical graph that represents the resulting summary graph. The following listing demonstrates how the operator is parameterized to compute the result of our introductory example of Figure 1(b):

```
1  LogicalGraph socialNetwork = // initialize logical graph ...
2  LogicalGraph summaryGraph = socialNetwork.groupBy(
3    [:label],
4    [(superVertex, vertices -> superVertex['count'] = vertices.size())],
5    [:label],
6    [(superEdge, edges -> superEdge['count'] = edges.size())])
```

In line 1 we initialize a logical graph representing our social network either from a data source or from previously executed graph operators. Next, we declare a new variable `summaryGraph` and initialize it using the result of the grouping operator. In Figure 1(b) the social network is grouped according to type labels of vertices and edges. Therefore, we define vertex and edge grouping keys in line 3 and 5 respectively. In GrALa the symbol `:label` refers to the dedicated property key τ representing the type label. Although our example requires only a single grouping key, it is also possible to define multiple keys to further differentiate vertices and edges (e.g., `[:label, 'city']`). In lines 4 and 6 we specify anonymous aggregate functions for vertices and edges using common Lambda notation (e.g., `(parameters -> function body)`). The vertex aggregate function in line 4 counts the number of vertices represented by a super vertex. The parameters of that function are the super vertex $v' \in V(G')$ and the vertex set $\{v \in V(G) \mid s_\nu(v) = v'\}$. The size of that set is stored as a new property of the super vertex using the property key `count`.

[5] The Gradoop framework implements GrALa using the Java programming language.

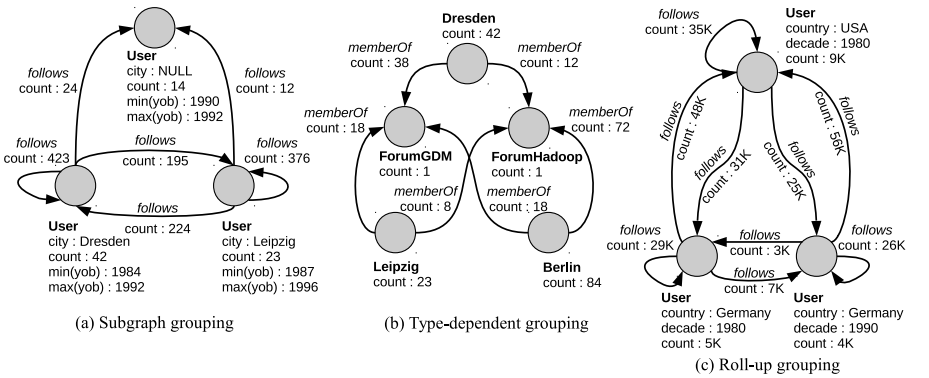


Fig. 2: Exemplary results for the subgraph and transformation based graph analytical programs. (a) shows a summary graph computed from a subgraph of the input graph; (b) shows a summary graph of a heterogeneous input graph; (c) shows a summary graph representing a roll-up operation

2.4 Analytical examples

A fundamental feature of the EPGM is its ability to compose graph operators to complex analytical programs. In the following, we will exemplify the abilities of the framework with a focus on graph grouping. Necessary operators will be briefly introduced, a detailed description can be found in [Ju16].

Subgraph grouping

Just like graph grouping, *subgraph* is a unary graph operator that outputs a single logical graph. The operator's arguments are user-defined predicate functions. Input vertices and edges will only be passed to the output graph if the respective predicate function evaluates to `true`. If there is only a single function defined, the operator extracts vertex-induced and edge-induced subgraphs, respectively.

The combination of subgraph and grouping operator allows the creation of partial summaries as illustrated by Figure 2(a). Script 1 lists the corresponding GrALa program. First, we extract a subgraph of a given social network that contains solely vertices of type `User` and edges of type `follows`. Therefore, we define vertex and edge predicate functions in lines 3 and 4. Both functions take a single element as input and define a condition on the corresponding type label to check if it matches the required label. During operator execution, the predicate functions are executed for each vertex and edge contained in the input graph. Since the output of the subgraph operator is a logical graph, it can be directly used as input for the grouping operator at line 5. In addition to the label, we further group vertices (users) based on the city they live in. We also provide further vertex aggregate functions to compute the minimum and maximum year of birth inside each user group. For simplicity, we use pre-defined aggregate functions provided by GrALa (e.g., `COUNT()`). In the resulting summary graph every vertex represents a group of users that share the same property value for the property key `city`. Additionally, vertices store the results of the specified aggregate functions as new properties. Since the EPGM is a schema-less data model, vertex and edge instances do not necessarily have a specified property. In Figure 2(a) this is reflected by a dedicated vertex representing all vertices without property *city*.

```
1  LogicalGraph summaryGraph = socialNetwork
2    .subgraph(
3      (vertex -> vertex[:label] == 'User'),
4      (edge -> edge[:label] == 'follows'))
5    .groupBy(
6      [:label, 'city'], [COUNT(), MIN('yob'), MAX('yob')],
7      [:label], [COUNT()])
```

Script 1: Graph grouping applied to a specific subgraph.

```
1  LogicalGraph summaryGraph = socialNetwork
2    .subgraph((edge -> edge[:label] == 'memberOf'))
3    .transform((vertex -> {
4      if (vertex[:label] == 'User') then vertex[:label] = vertex['city']
5      else vertex[:label] = vertex[:label] + vertex['title']}))
6    .groupBy([:label], [COUNT()], [:label], [COUNT()])
```

Script 2: Type-dependent grouping using vertex transformation.

With such a declarative program, an analyst is now able to explore the structure and properties of entities and their mutual relationships contained in the underlying social network. For example, one can easily see that there are more follower relations among users located in the same city than among users from different cities. The program also demonstrates the flexibility of the operator: by adding the property key `gender` to the vertex grouping keys, the super vertices in Figure 2(a) could be further divided into groups with respect to users' gender and reveal more information about the relations between gender groups from different cities.

Type-dependent grouping of heterogeneous graphs

In the previous example, we applied graph grouping to a homogeneous subgraph containing solely vertices of type `User` and edges of type `follows`. However, it might also be necessary to apply graph grouping on a heterogeneous graph, i.e., to define type-dependent vertex and edge grouping keys [YG16]. Within the EPGM, this can be achieved by combining the grouping operator and a preceding *graph transformation*. The transformation operator is a unary graph operator that allows the modification of graph, vertex and edge data whereas graph structure remains unchanged. The operator is parameterized by user-defined transformation functions that either have graph, vertex or edge data as input. Within these functions the analyst is able to modify type label and properties of the particular instance. The operator's output is a new logical graph with identical structure as the input graph but modified data of its elements. Transformation is typically helpful in data integration and ETL scenarios [Pe14a, Pe14b] or to pre-process graphs for subsequent graph operators as in our next example.

In Script 2, we create a summary graph showing different forums and their members under consideration of their cities to analyze local interests. Therefore, we first extract an edge-induced subgraph from our social network that solely contains edges of type `memberOf` since this relationship connects users with forums. In line 3, we apply a transformation function on the vertices of our previously extracted subgraph. The function modifies a vertex with respect to its current label: for users, we replace the label by the property value

```
1  LogicalGraph summaryGraph = socialNetwork
2    .subgraph((edge -> edge[:label] == 'follows'))
3    .transform((vertex -> {
4      vertex['decade'] = vertex['yob'] - (vertex['yob'] mod 10)
5      vertex['country'] = getCountry(vertex['city']) }))
6    .groupBy([:label, 'country', 'decade'], [COUNT()], [:label], [COUNT()])
```

Script 3: Graph grouping on property hierarchies using vertex transformation.

associated to their city attribute and, for forums, we concatenate original label and unique forum title to create a new label. The modified graph is then used as input for the grouping operator in line 6. Since we projected all necessary information to vertex labels, we can now group vertices and edges by label and additionally count the particular group members.

An exemplary summary graph is illustrated by Figure 2(b). The grouping operator created a vertex for each city representing users from that city. Since labels of forum vertices contain the unique forum title, the grouping operator created super vertices that represent a single vertex from the input graph. However, users in the neighborhood of these vertices and their relations to the forums have been grouped. By looking at the number of memberships stored at super edges, an analyst is now able to conclude about the interests of local user groups.

Graph grouping along property hierarchies

Another application in which transformation increases the flexibility of graph grouping is the consideration of property (or dimension) hierarchies. Here, an analyst wants to group a graph on different levels, e.g., by using a dimension hierarchy like *time* or *location*. This type of operation is also known as roll-up in data warehousing or graph OLAP scenarios [Ch08, Zh11, YWZ12]. In our social network example, users store year of birth as well as the city they live in. In Script 3, we use transformation to compute coarser levels of dimensional hierarchies by mapping years to decades (line 4) and cities to corresponding countries (line 5). The results are stored in new vertex properties and the respective property keys are used as vertex grouping keys in the subsequent grouping operator. An exemplary result of the program can be seen in Figure 2(c). In contrast to Figure 2(a), vertices now represent users that live in the same country and were born in the same decade. Using such a high-level view, the summary graph provides useful insights about a network which may contain millions or even billions of users.

3 Implementation of Graph Grouping in Gradoop

For the implementation of graph grouping and Gradoop operators in general, we have to cope with two major challenges of big data analytics: the operator's flexible integration in complex analyses as shown in our examples and its scalability for very large graphs with billions of edges. A now established approach to solve the latter problem is the massively parallel computation on shared-nothing clusters, e.g., based on the Hadoop ecosystem.

Especially promising is the utilization of distributed dataflow systems such as Apache Spark [Za12] and Apache Flink [Ca15b] that, in contrast to the older MapReduce framework [DG08], offer a wider range of operators and keep data in main memory between the execution of operators. The major challenges of implementing graph operators in these systems are identifying an appropriate graph representation, an efficient combination of

the primitive dataflow operators and the minimization of data exchange among different machines.

Our implementation of graph grouping is, like the implementation of all Gradoop operators, based on Apache Flink. A basic version of the implementation has also been contributed to Apache Flink.[6] We first give a brief introduction to Apache Flink and its programming concepts and explain the mapping of the EPGM to these concepts. We then outline the implementation of graph grouping including an optimization for unbalanced data distribution.

3.1 Apache Flink

Apache Flink [Ca15b] supports the declarative definition and execution of distributed dataflow programs. The basic abstractions of such programs are *DataSets* and *Transformations*. A DataSet is an immutable, distributed collection of arbitrary data objects, e.g., Pojos or tuple types, and transformations are higher-order functions that describe the construction of new DataSets either from existing ones or from data sources. Application logic is encapsulated in user-defined functions (UDFs), which are provided as arguments to the transformations and applied to DataSet elements.

Well-known transformations have been adopted from the MapReduce paradigm [DG08]. While the *map* transformation expects a bijective UDF that maps each element of the input DataSet to exactly one element of the output DataSet, the *reduce* transformation aggregates all input elements to a single one. Further transformations are known from relational databases, e.g., *join*, *group-by*, *project* and *distinct*. Table 1 introduces the transformations available in Apache Flink's DataSet API that are relevant for this paper. In addition, Apache Flink provides libraries for analytical tasks such as machine learning, graph processing and relational operations. To describe a dataflow, a program may include multiple chained transformations and library calls. During execution Flink handles program optimization as well as data distribution and parallel processing across a cluster of machines.

3.2 Gradoop

The Gradoop open-source library is a complete implementation of the EPGM and its operators on top of Apache Flink's DataSet API. It can be used standalone or in combination with any other library available in the Flink ecosystem. Gradoop uses three object types to represent EPGM data model elements: *graph head*, *vertex* and *edge*. A graph head represents the data associated to a single logical graph. Vertices and edges not only carry data but also store their graph membership as they may be contained in multiple logical graphs. In the following, we show a simplified definition of the respective types:

```
class GraphHead { Id; Label; Properties }
class Vertex    { Id; Label; Properties; GraphIds }
class Edge      { Id; Label; Properties; SourceId; TargetId; GraphIds }
```

Each type contains a system managed identifier (`Id`) represented by a 128-bit *universally unique identifier*[7]. Furthermore, each element has a label of type string and a set of properties. Since EPGM elements are self-descriptive, properties are represented by a key-value map whereas the property key is of type String and the property value is encoded in a byte array.

[6] https://issues.apache.org/jira/browse/FLINK-2411

[7] docs.oracle.com/javase/7/docs/api/java/util/UUID.html

Name	Description
Map	The map transformation applies a user-defined map function to each element of the input DataSet. Since the function returns exactly one element, it guarantees a one-to-one relation between the two DataSets. `DataSet<IN>.map(udf: IN -> OUT) : DataSet<OUT>`
Filter	The filter transformation evaluates a user-defined predicate function to each element of the input DataSet. If the function evaluates to `true`, the particular element will be contained in the output DataSet. `DataSet<IN>.filter(udf: IN -> Boolean) : DataSet<IN>`
Project	The projection transformation takes a DataSet containing a tuple type as input and forwards a subset of user-defined tuple fields to the output DataSet. `DataSet<TupleX>.project(fields) : DataSet<TupleY> (X,Y in [1,25])`

Join	The join transformation creates pairs of elements from two input DataSets which have equal values on defined keys (e.g., field positions in a tuple). A user-defined join function is executed for each of these pairs and produces exactly one output element.
	`DataSet<L>.join(DataSet<R>).where(leftKeys).equalTo(rightKeys)`
	`  .with(udf: (L,R) -> OUT) : DataSet<OUT>`

ReduceGroup	DataSet elements can be grouped using custom keys (similar to join keys). The ReduceGroup transformation applies a user-defined function to each group of elements and produces an arbitrary number of output elements. `DataSet<IN>.groupBy(keys).reduceGroup(udf: IN[] -> OUT[]) : DataSet<OUT>`
CombineGroup	The CombineGroup transformation is similar to a ReduceGroup transformation but is not guaranteed to process all elements of a group at once. Instead, it processes sub-groups of all elements stored in the same partition (i.e., are processed by the same worker). Thus, it can be used to decrease data volume before a ReduceGroup transformation which otherwise may require shuffling data over the network. `DataSet<IN>.groupBy(keys).combineGroup(udf: IN[] -> OUT[]) : DataSet<OUT>`

Tab. 1: Subset of Apache Flink DataSet transformations. We define `DataSet<T>` as a DataSet that contains elements of type `T` (e.g., `DataSet<String>` or `DataSet<Tuple2<Int,Int>>`).

The current implementation supports values of all primitive Java types and BigDecimal. Vertices and edges maintain their graph membership in a dedicated set of graph identifiers (`GraphIds`). Edges additionally store the identifiers of their incident vertices. To represent a graph collection, Gradoop uses a separate Flink DataSet for each element type. A logical (property) graph is a special case of a graph collection in which the graph head DataSet contains a single object:

```
class LogicalGraph {
    DataSet<GraphHead> graphHead;
    DataSet<Vertex> vertices;
    DataSet<Edge> edges;
}
```

EPGM operators are implemented using Flink transformations on a logical graph's or a graph collection's DataSets. For example, the subgraph operator is implemented using the `filter` transformation to apply user-defined predicate functions and the `join` transformation to preserve consistency in the output graph. More complex operators like graph pattern matching use a large number of different Flink transformations including iterations.

3.3 Graph Grouping

The graph grouping operator computes a summary graph by applying a series of transformations to the vertex and edge DataSets of an input graph. The algorithmic idea is to group vertices according to user-defined grouping keys and to create a super vertex for each resulting group. Then, a mapping between the original vertices and the super vertices is

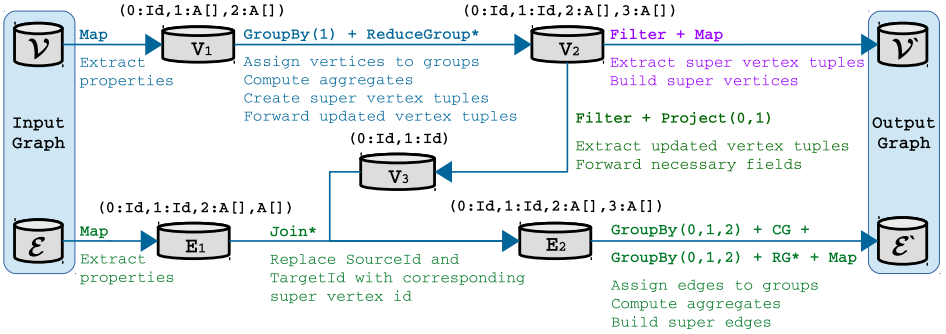


Fig. 3: Dataflow implementation of the graph grouping operator using Flink DataSets and transformations. The dataflow is subdivided into three phases: (1, blue) grouping vertices, (2, purple) building super vertices and (3, green) building super edges. Intermediate DataSets always contain tuples, whose fields can be referred to by their position, e.g., `GroupBy(1)` or `Project(0,1)`. Transformations denoted with * require inter-partition communication between workers in a cluster.

used to update source and target identifiers for each edge. Finally, edges are also grouped by their new source and target vertex and optionally by user-defined grouping keys.

Figure 3 illustrates the corresponding dataflow program from an input logical graph $G(V, E)$ to an output summary graph $G'(V', E')$. For the sake of clarity, we grouped multiple transformations (e.g., `Filter + Map`) and omitted intermediate results when possible. A comprehensive pseudocode description can be found online.[8] We use different colors to denote the three phases of the algorithm: (1, blue) grouping vertices, (2, purple) building super vertices and (3, green) building super edges. In Figure 4, we additionally illustrate a dataflow instance that computes the "Type Graph" of Figure 1(b). Using the abstraction and the concrete example, we will now discuss the three phases.

Phase 1: Grouping vertices

In a distributed dataflow framework it is important to reduce data volume whenever possible in order to avoid unnecessary network traffic. Thus, the first phase starts with mapping each vertex $v \in V(G)$ to a tuple representation containing only vertex identifier and a list of property values needed for vertex grouping and aggregation.[9] For example, in Figure 4, we map vertex v_1 to the tuple `(1,[User],[1])` as we require the vertex label for the creation of super vertices and the property value 1 to compute the `count` aggregate. Note that grouping and aggregate values need to be ordered. Vertices (and edges) without a required grouping value show the `Null`-value instead. Missing aggregate values are replaced by a default value determined by the particular aggregate function.[10] In Figures 3 and 4, the intermediate result containing vertex tuples is represented by DataSet V_1.

In the second step of phase 1, vertex tuples are grouped by the previously extracted grouping values (position 1 inside the tuple). Each group is then processed by a ReduceGroup function

[8] http://dbs.uni-leipzig.de/file/grouping_pseudocode.pdf

[9] In Figure 3, lists of property values are denoted by the type `A[]`.

[10] The `count` aggregate function is implemented as a special case of the `sum` aggregate function. In that case, the property value "1" is added automatically.

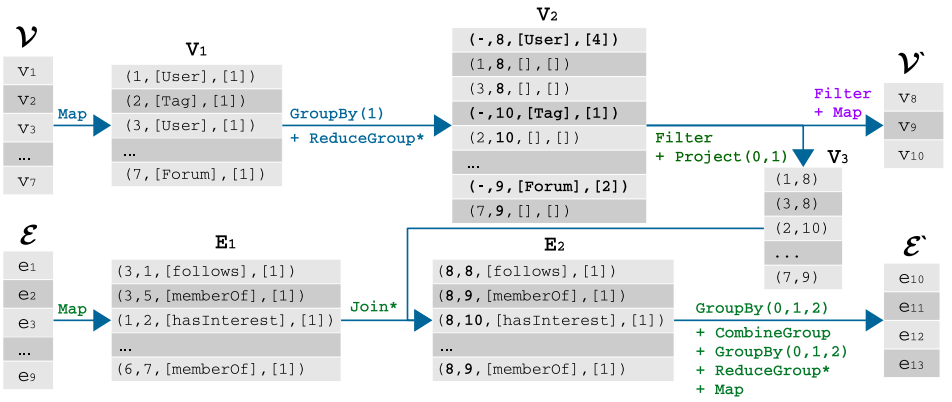


Fig. 4: Dataflow using the graphs from Figure 1(a) as input and Figure 1(b) as output.

which has two main tasks: (1) creating a *super vertex tuple* for each group and (2) creating a map between original vertex id and corresponding super vertex id. A super vertex tuple consists of a new vertex id, the property values representing the group and the results of the provided aggregate functions. In Figure 4, the intermediate DataSet V_2 contains three highlighted super vertex tuples, one for each vertex label in the input graph. For example, the tuple `(-,8,[User],[4])` represents all vertex tuples with a grouping value `User` including the result of the `count` aggregate function (`4`). Additionally, the ReduceGroup function outputs a super vertex-mapping for every member, e.g., `(1,8,[],[])` is derived from tuple `(1,[User],[1])` as vertex `1` belongs to super vertex `8`.

Phase 2: Building super vertices

In this phase, we construct the final super vertices V' from the previously created super vertex tuples. Therefore, we need to filter the tuples from the intermediate DataSet V_2 and use a map transformation to construct a new `Vertex` instance for each tuple. In Figure 4, the super vertex tuple `(-,8,[User],[4])` is mapped to super vertex v_8 of Figure 1(b), which stores the grouping and aggregate values as new properties.

Phase 3: Building super edges

In the last phase, we update the edges of input DataSet E according to their super vertices and group them to super edges E'. Similar to phase 1, we first reduce data volume by mapping all edges $e \in E$ to necessary information. In DataSet E_1, each edge is represented by a tuple containing its source and target id as well as required grouping and aggregate values. For the next step, we also require the super vertex-mappings from the intermediate DataSet V_2. The mappings are extracted using a filter transformation and we further reduce their memory footprint by projecting only necessary fields. DataSet V_3 now represents a complete mapping between original vertex id and super vertex id. In the example of Figure 4, vertex ids 1, 3, 5 and 6 are associated with vertex id 8, while 2 is represented by vertex id 10. We now join DataSets E_1 and V_3 on the original vertex ids to update the source and target vertex id for each edge tuple. Technically, we require separate join operations for both identifiers (see pseudocode for details). DataSet E_2 represents the updated edge tuples, e.g., tuple `(3,1,[follows],[1])` is updated to `(8,8,[follows],[1])`, since source id (3) and target

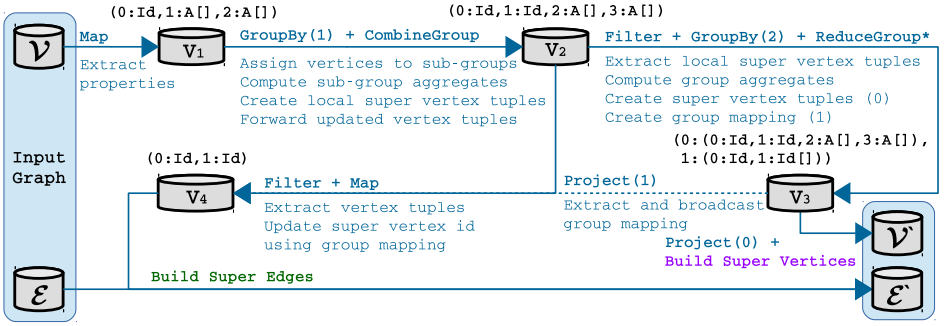


Fig. 5: Optimized dataflow for Phase 1.

id (1) are both represented by super vertex id 8. In contrast, tuple `(3,5,[memberOf],[1])` is mapped to `(8,9,[memberOf],[1])` as source and target vertex belong to different super vertices (i.e., users and forums).

Since the updated tuples in E_2 are logically connecting super vertices, we can group them by source and target id as well as their grouping property values. The creation of *super edge tuples* is done in two consecutive steps. First, we use a CombineGroup transformation to group edge tuples per data partition. Depending on the data distribution, each partition may contain multiple sub-groups. For each of these sub-groups, the CombineGroup function produces exactly one *local super edge tuple* that stores the grouping values and the results of the aggregate functions for that sub-group. After the CombineGroup step, there might be two tuples representing the same edge group, e.g., `(8,8,[follows],[2])` and `(8,8,[follows],[1])`. To create exactly one tuple for each edge group, we again need to group the local super edge tuples by the same fields, but this time followed by a ReduceGroup function to guarantee that all tuples representing the same edge group are processed together. Since the computation logic of both functions is identical, we can use the same UDF in the combine and reduce step. The output is one super edge tuple, e.g., `(8,8,[follows],[3])`, that represents the whole group. Finally, each super edge tuple is mapped to a new `Edge` that stores grouping and aggregate values as new properties.

After phase 3, the computed super vertices V' and edges E' are used as parameters to instantiate a new logical graph G'. During instantiation, a new `GraphHead` is created and graph memberships of super vertices and edges are updated. The logical graph is then returned to the program and can be sent to either a data sink or subsequent operations.

3.4 Handling unbalanced workloads

In distributed computing, the application of a local combination step is generally useful as it potentially reduces network traffic [DG08, Ma10]. However, the efficiency of the combine step depends on the physical data distribution: if each worker has one element of each group, the CombineGroup function has no effect. Nevertheless, it is advisable to add a combine step if the computation logic permits it. In phase 3, we use a CombineGroup transformation to map edge sub-groups to local super edge tuples which are then shuffled across the network in the subsequent ReduceGroup transformation. For edges, this step is straightforward as each sub-group can be processed independently. However, for vertices, the algorithm needs

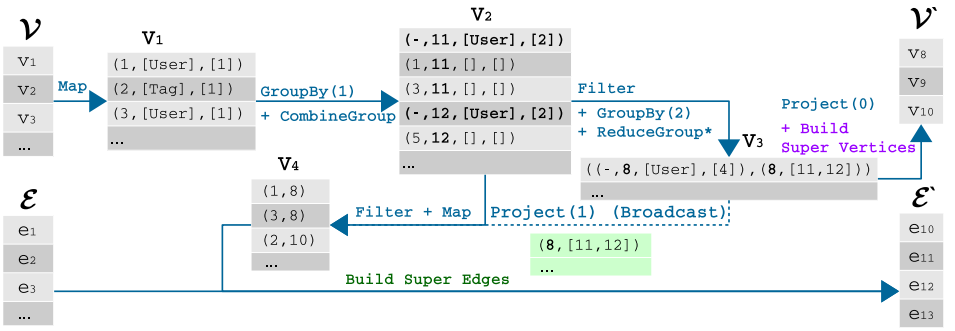


Fig. 6: Optimized dataflow using the graphs from Figure 1(a) as input and Figure 1(b) as output.

to guarantee that all vertices of a group are assigned to the same super vertex. In Figure 4, this constraint is met by applying a ReduceGroup function directly on the grouped vertex tuples. Yet, this leads to an unbalanced workload if the group size distribution is skewed since all group members need to be transferred to the same worker.

Figure 5 illustrates an alternative approach for phase 1. We again start with building vertex tuples using a map transformation. We then use a CombineGroup transformation on the grouped vertex tuples which creates DataSet V_2. The UDF logic is identical to the original phase, however, we now create *local* super vertex tuples potentially representing the same vertex group. Figure 6 shows an example: V_2 contains two local super vertex tuples representing vertices of type `User` and mapping tuples contain local super vertex ids. We now filter local super vertex tuples from V_2. Up to this point, the dataflow does not require any data shuffling over the network. The reduced DataSet is then grouped again by the grouping values and processed in a ReduceGroup transformation. Here, we apply a different logic: in addition to the final super vertex tuple, we also create a mapping from the final super vertex id to all local super vertex ids representing the same group. In Figure 6, DataSet V_3 contains tuples composed of two tuples: the super vertex tuple (e.g., `(-,8,[User],[4])`) and the respective group mapping (e.g., `(8,[11,12])`).

We extract the super vertex tuples from the composed tuples of V_3 using projection and create final super vertices in phase 2. Prior to phase 3, we need to update the mappings between vertices and local super vertices in V_2. After filtering these tuples from V_2, we replace the local super vertex id by the global one in a map transformation. To achieve this, we use a Flink feature called *broadcasting*, which allows distributing an entire DataSet to all workers in a cluster and reading it in a UDF context. In Figure 6, we highlighted the DataSet that is being broadcasted to the map function. In this function, we just need to determine the super vertex id which maps to the current local super vertex id of the tuple. In the example of Figure 6, the vertex tuple `(1,11,[],[])` is updated to `(1,8)` since the local super vertex id 11 is represented by super vertex id 8. The resulting DataSet V_4 is identical to DataSet V_3 in Figure 3 and is used to create super edges in the final phase.

In contrast to regular ones, broadcast DataSets are kept in-memory on each worker. Since the size of group mappings depends on the number of vertex groups and data distribution, memory consumption of the broadcast DataSet may vary heavily. Before applying the broadcasting approach, one needs to consider that computation will stop if a broadcast DataSet exceeds the available main memory of a worker.

Name	$\|V\|$	$\|E\|$	Disk size	$\|T_V\|$	$\|T_E\|$	$\|V_{\tau=Person}\|$	$\|E_{\tau=knows}\|$
GA.10	260 K	16.6 M	4.5 GB	8	8	235 K (90.2%)	10.2 M (61.2%)
GA.100	1.7 M	147 M	40.2 GB	8	8	1.67 M (98.4%)	101 M (68.9%)
GA.1000	12.7 M	1.36 B	372 GB	8	8	12.67 M (99.8%)	1.01 B (74.4%)
Pokec	1.6 M	30.6 M	5.6 GB	1	1	1.6 M (100%)	30.6 M (100%)

Tab. 2: Statistics of the social network datasets used in the benchmarks.

4 Evaluation

In the experiments we evaluate the scalability of our implementation with respect to increasing computing resources and data volume. We further analyze the operators' runtime according to the number of grouping keys and aggregate functions. Finally, we study the effect of a combiner in the vertex grouping phase as described in Section 3.4.

4.1 Experimental setup

We perform our experiments using datasets generated by the *Graphalytics* (GA) benchmark for graph processing platforms [Er15, Ca15a]. The generator creates heterogeneous social networks with a schema similar to our examples and structural characteristics like those of real-world networks: node degree distribution based on power-laws and skewed property value distributions [Er15]. We additionally use the Pokec social network[11] containing users including their properties and mutual friendship relations. Table 2 provides an overview about the used datasets. Since several experiments use GA subgraphs solely containing vertices of type *User* and edges of type *knows*, we include their respective share in the table.

Table 3 shows the different configurations of the grouping operator used in the experiments. Configurations 1 to 4 compute type graphs and are mainly used in the scalability experiments. In configurations 5 to 13, we use a varying numbers of vertex and edge grouping keys as well as none, one or multiple aggregate functions. The used datasets either explicitly or implicitly fulfill the specified properties. For example, in Graphalytics, a users' location is encoded by an edge to a vertex of type `City`. We thus align all datasets to a common schema in a preceding ETL step.

The benchmarks are performed on a cluster with 16 worker nodes. Each worker consists of an E5-2430 6(12) 2.5 Ghz CPU, 48 GB RAM, two 4 TB SATA disks and runs openSUSE 13.2. The nodes are connected via 1 Gigabit Ethernet. Our evaluation is based on Hadoop 2.6.0 and Flink 1.0.3. We run Apache Flink standalone with 6 threads and 40 GB memory per worker. In our experiments, we vary the number of workers by setting the parallelism parameter to the respective number of threads (e.g., 2 workers correspond to 12 threads). The datasets are stored in HDFS (default settings) using a Gradoop specific JSON format and distributed using hash-based partitioning. We tested both operator implementations described in the preceding section either using a ReduceGroup function (RG) or an additional CombineGroup function (CG). Runtimes are reported by Flink's execution environment and include reading the input graph from HDFS and writing the summary graph to HDFS. In the subsequent results, each datum represents the average runtime of five executions.

[11] https://snap.stanford.edu/data/soc-pokec.html

Config.	Vertex keys	Vertex aggregate functions	Edge keys	Edge aggregate functions
1	:label	-	-	-
2	:label	COUNT()	-	-
3	:label	COUNT()	:label	-
4	:label	COUNT()	:label	COUNT()
5	city	-	-	-
6	city	COUNT()	-	-
7	city, gender	-	-	-
8	city, gender	COUNT()	-	-
9	city, gender	COUNT(), MIN(yob)	-	-
10	city, gender	COUNT(), MIN(yob), MAX(yob)	-	-
11	city	COUNT()	-	COUNT()
12	city	COUNT()	-	COUNT(), MIN(since)
13	city	COUNT()	-	COUNT(), MIN(since), MAX(since)

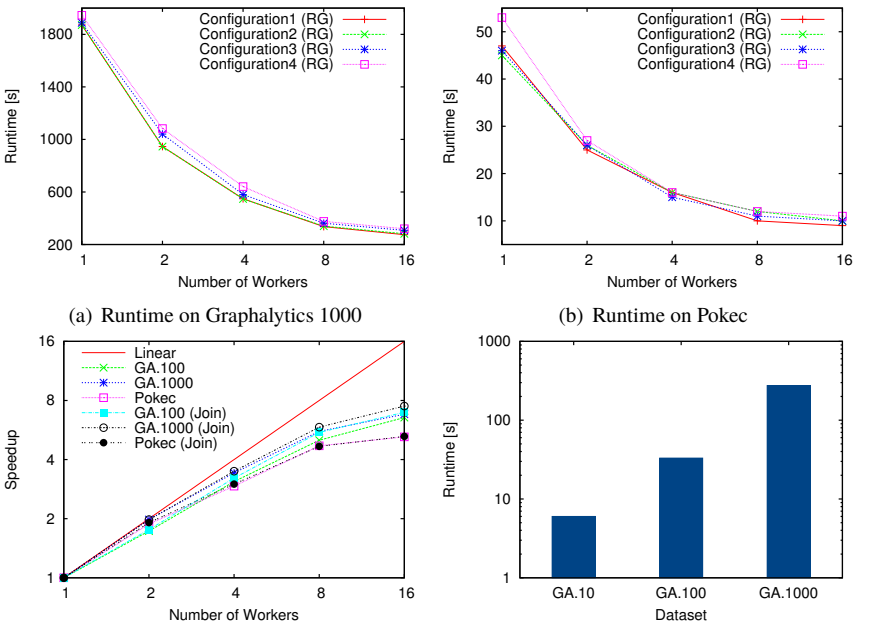
Tab. 3: Different configurations for the grouping operator.

4.2 Experimental results

Scalability We first evaluate absolute runtime and relative speedup of our implementation using configurations 1 to 4 on Graphalytics 100, 1000 and Pokec. We execute the operator on each dataset using an increasing number of workers for each run. The runtime results for Graphalytics 1000 and Pokec are shown in Figure 7(a) and 7(b), respectively. For the largest graph with 1.3 B edges, we could decrease the runtime from about 30 minutes on a single machine to 4.5 minutes on 16 workers. For the real-world network with 30 M edges, we could reduce runtimes to only 10 seconds. Using configuration 4, the execution requires the most time due to the highest data volume and computational cost caused by type labels and aggregates. However, on both datasets, the runtime is close to configurations 1 to 3. Figure 7(c) illustrates the relative speedup for configuration 1 including Graphalytics 100. One can see that up to four workers, the speedup is nearly linear and degrades after this. We assume that this is due to the two data intensive join transformations in phase 3. Here, the edge tuples need to be shuffled across the cluster which makes the network the limiting resource (also because of its limited bandwidth of only 1 Gbps). In Figure 7(c), we added the speedup solely for the join operation which aligns with the speedup of the complete runtime and thus verifies our assumption.

We also evaluated scalability with increasing data volume and a fixed number of workers. The results in Figure 7(d) show that the runtime increases almost linearly with the data volume. Using configuration 1, the operator execution required about 30 seconds on GA.100 (40 GB) and 275 seconds on GA.1000 (372 GB).

Operator parametrization In configurations 5 to 13, we parameterized the operator with a varying number of grouping keys and aggregate functions. We execute the operator on the `User`-subgraphs of GA.100 and GA.1000 as well as on the Pokec dataset using 16 workers. The results are shown in Figure 8. The first observations are that the execution times for all configurations vary only slightly and scale almost linearly from GA.100 to GA.1000. By looking at config. 5 and 7 (or 6 and 8), we can also observe that a higher number of vertex grouping keys only leads to a small increase in runtime (e.g., 319 seconds for config. 5 and 350 seconds for config. 7 on GA.1000.SG). The increase is caused by a higher number of



(a) Runtime on Graphalytics 1000

(b) Runtime on Pokec

(c) Speedup for configuration 1 (RG)

(d) Runtime for configuration 1 (RG) on 16 workers

Fig. 7: Evaluation results for the scalability benchmarks.

resulting super vertices and edges, for example, config 5 on GA.1000.SG leads to 1149, while config. 7 leads to 2298 super vertices.

In configurations 7 to 10 we increased the number of vertex aggregate functions and in configurations 11 to 13 the number of edge aggregate functions. The latter could only be executed on Graphalytics since Pokec has no edge properties. The results in Figure 8 show that a higher number of aggregate functions does not lead to a higher runtime. This is due to the fact, that in our Reduce- and CombineGroup functions, all aggregates are computed in a single iteration over the vertex and edge tuples, respectively.

Optimization for data skew We finally study the effect of an additional combination step in the vertex grouping phase. Since the type label distributions of our datasets are skewed [Er15], we use configurations 1 to 4 to compute type graphs on 16 workers. The results are shown in Figure 9. We executed the operator with each configuration using a ReduceGroup function (RG) and an additional CombineGroup function (CG). For the synthetic datasets, the benefit is generally small but more distinctive for the larger dataset. However, for the real-world dataset, we could achieve an average runtime reduction of about 15%. We believe that this is caused by the fact, that Pokec contains only one vertex type. Thus, in the ReduceGroup implementation, one worker needs to process all vertices, while in the CombineGroup implementation, load distribution and network traffic are improved due to a local preprocessing of vertices. For the synthetic datasets this effect is not that strong since the load distribution is generally better due to the higher number of type labels, i.e., vertex groups are processed by multiple workers.

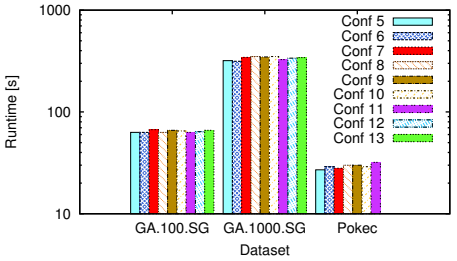


Fig. 8: Runtime for different configurations.

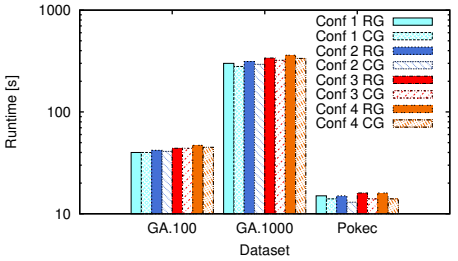


Fig. 9: Comparison of vertex grouping phases.

5 Related Work

Gradoop in general is related to graph processing frameworks on top of distributed dataflow systems, e.g., GraphX on Apache Spark [Xi13] and Gelly on Apache Flink [Ca15b]. These libraries focus on the efficient execution of iterative graph algorithms. However, in contrast to the EPGM, the implemented graph data models are generic, which means arbitrary user-defined data can be attached to vertices and edges. In consequence, model-specific operators, e.g., graph grouping, need to be user-defined, too. Hence, using those libraries to solve complex analytical problems becomes a laborious programming task. In contrast, Gradoop targets the data scientist by providing an abstraction from the underlying framework, in particular an expressive data model and declarative operators.

Graph grouping is related to the area of online analytical processing (OLAP) on graphs. Here, attributes of the graph elements are considered as dimensions and a summary graph (also denoted by aggregate graph) is one of many cuboids in a so-called graph cube. Most of the publications focus on constructing and querying graph cubes from homogeneous or heterogeneous input graphs. GraphOLAP [Ch08] first discusses grouping of single graphs and roll-up/drill-down and slice/dice operations by overlaying and filtering graphs, respectively. In [THP08], the authors introduce SNAP, an algorithm to construct summary graphs based on user-defined vertex keys and relationship types, and k-SNAP, which produces k super vertices by separating vertices depending on structural similarity. GraphCube [Zh11] extends the concepts of [Ch08] by the definition of crossboid queries enabling analysis through different levels of graph aggregations. While the previous approaches focus on homogeneous [Ch08, THP08, Zh11], vertex-attributed [THP08, Zh11] input graphs, HMGraph [YWZ12] and GRAD [Gh15] enable analyzing heterogeneous, vertex-attributed [YWZ12] graphs. HMGraph introduces additional operators on vertex-dimensions while GRAD proposes an extension to the property graph model that simplifies the definition of hierarchies in a graph cube. In [YG16], the authors introduce type-dependent grouping of heterogeneous information networks and, like [THP08], also group vertices based on a similarity function using graph entropy.

There are two previous approaches to compute graph cubes in a distributed fashion: Distributed GraphCube [DGS13] on Apache Spark and Pagrol [Wa14] on Hadoop MapReduce. While both are distributed implementations of GraphCube and thus work only on homogeneous graphs, Pagrol additionally considers edge attributes as dimensions. The authors show

that their implementations scale for the computation of the complete graph cube [Wa14] as well as single cuboids [DGS13].

In contrast to the existing Graph OLAP approaches, the graph grouping operator of Gradoop is not focusing on the creation of a graph cube containing all possible summary graphs/cuboids of a heterogeneous network. Instead, we built a framework that not only focuses on graph grouping, but also allows the flexible integration of summary graphs in combination with other complex graph operations. With regard to real-world data, which is typically semi-structured and highly dynamic, we consider our approach to be advantageous in comparison to pre-computing a complete graph cube. Furthermore, our operator provides user-defined aggregation functions, is able to handle semi-structured, heterogeneous data and allows for the interactive computation and exploration of summary graphs.

6 Conclusion and Future Work

We introduced a graph operator for the efficient, distributed grouping of large-scale, semi-structured property graphs. As the operator is part of the Extended Property Graph Model, it can be flexibly supplemented with other graph operators to express various analytical problems. Operator implementations have been contributed to the Gradoop graph analytics framework as well as to Apache Flink. One major challenge was the efficient mapping of graph grouping semantics to the abstractions provided by Flink's batch API and, at the same time, considering the absence of shared memory and the reduction of network traffic. In our experimental evaluation, we could demonstrate that our implementation scales well with graph size and can achieve very low response times on real-world networks which is a first step towards interactive exploration of large, distributed graphs. We could also show that the implementation handles skewed data distributions by leveraging Flinks combiner and broadcasting capabilities. However, our experiments revealed that a major limitation for scalability is data shuffling across the cluster. To further improve scalability of graph grouping and Gradoop operators in general, we are looking into different graph representations and graph partitioning strategies.

Besides runtime optimization, we see multiple directions for future work. First, as targeted users of Gradoop are data scientists, one goal is to improve our DSL's declarativity, e.g., to explicitly support type-dependent grouping or OLAP operations. Furthermore, we aim to add new features required by graph OLAP scenarios, for example, efficient roll-up/drill down operations and a distributed caching mechanism for summary graphs. Finally, since many real-world graphs are highly dynamic, i.e., their structure changes over time, we will investigate in implementing graph operators on dynamic graphs or rather graph streams. Here, the major challenge is the definition of a graph stream model and operator semantics.

7 Acknowledgments

This work is partially funded by the German Federal Ministry of Education and Research under project ScaDS Dresden/Leipzig (BMBF 01IS14014B).

References

[Al14] Alexandrov, A. et al.: The Stratosphere Platform for Big Data Analytics. The VLDB Journal, 23(6), December 2014.

[An12] Angles, R.: A Comparison of Current Graph Database Models. In: Proc. ICDEW. 2012.

[Ca15a] Capotă, M. et al.: Graphalytics: A Big Data Benchmark for Graph-Processing Platforms. In: Proc. GRADES. 2015.

[Ca15b] Carbone, P. et al.: Apache Flink™: Stream and Batch Processing in a Single Engine. IEEE Data Eng. Bull., 38(4), 2015.

[Ch08] Chen, C.; Yan, X.; Zhu, F.; Han, J.; Yu, P. S.: Graph OLAP: Towards online analytical processing on graphs. In: Proc. ICDM. 2008.

[DG08] Dean, J.; Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters. Commun. ACM, 51(1), January 2008.

[DGS13] Denis, B.; Ghrab, A.; Skhiri, S.: A distributed approach for graph-oriented multidimensional analysis. In: Proc. Big Data. Oct 2013.

[Er15] Erling, O. et al.: The LDBC Social Network Benchmark: Interactive Workload. In: Proc. SIGMOD. 2015.

[Gh15] Ghrab, A.; Romero, O.; Skhiri, S.; Vaisman, A.; Zimányi, E.: A Framework for Building OLAP Cubes on Graphs. In: Proc. ADBIS. 2015.

[Ju16] Junghanns, M.; Petermann, A.; Teichmann N.; Gómez K.; Rahm E.: Analyzing Extended Property Graphs with Apache Flink. In: Proc. SIGMOD NDA Workshop. 2016.

[Ma10] Malewicz, G. et al.: Pregel: A System for Large-scale Graph Processing. In: Proc. SIGMOD. 2010.

[Ne10] Newman, M.: Networks: An Introduction. 2010.

[Pa11] Pavlopoulos, G. A. et al.: Using graph theory to analyze biological networks. BioData Mining, 4(1), 2011.

[Pe14a] Petermann, A.; Junghanns, M.; Müller, R.; Rahm, E.: BIIIG: Enabling business intelligence with integrated instance graphs. In: Proc. ICDE Workshops. 2014.

[Pe14b] Petermann, A.; Junghanns, M.; Müller, R.; Rahm, E.: Graph-based Data Integration and Business Intelligence with BIIIG. PVLDB, 7(13), 2014.

[RN10] Rodriguez, M. A.; Neubauer, P.: Constructions from Dots and Lines. arXiv:1006.2361v1, 2010.

[THP08] Tian, Y.; Hankins, R. A.; Patel, J. M.: Efficient Aggregation for Graph Summarization. In: Proc. SIGMOD. 2008.

[Wa14] Wang, Z. et al.: Pagrol: Parallel graph olap over large-scale attributed graphs. In: Proc. ICDE. 2014.

[Xi13] Xin, R. S.; Gonzales, J. E.; Franklin, M. J.; Stoica, I.: GraphX: A Resilient Distributed Graph System on Spark. In: Proc. GRADES. 2013.

[YG16] Yin, D.; Gao, H.: A flexible aggregation framework on large-scale heterogeneous information networks. Journal of Information Science, 2016.

[YWZ12] Yin, M.; Wu, B.; Zeng, Z.: HMGraph OLAP: A Novel Framework for Multi-dimensional Heterogeneous Network Analysis. In: Proc. DOLAP. 2012.

[Za12] Zaharia, M. et al.: Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In: Proc. NSDI. 2012.

[Zh11] Zhao, P.; Li, X.; Xin, D.; Han, J.: Graph Cube: On Warehousing and OLAP Multidimensional Networks. In: Proc. SIGMOD. 2011.