Analyzing Extended Property Graphs with Apache Flink

Martin Junghanns, André Petermann, Niklas Teichmann, Kevin Gómez, Erhard Rahm
University of Leipzig & ScaDS Dresden/Leipzig
[junghanns,petermann,rahm]@informatik.uni-leipzig.de
[teichmann,gomez]@studserv.uni-leipzig.de

ABSTRACT
Graphs are an intuitive way to model complex relationships between real-world data objects. Thus, graph analytics plays an important role in research and industry. As graphs often reflect heterogeneous domain data, their representation requires an expressive data model including the abstraction of graph collections, for example, to analyze communities inside a social network. Further on, answering complex analytical questions about such graphs entails combining multiple analytical operations. To satisfy these requirements, we propose the Extended Property Graph Model, which is semantically rich, schema-free and supports multiple distinct graphs. Based on this representation, it provides declarative and combinable operators to analyze both single graphs and graph collections. Our current implementation is based on the distributed dataflow framework Apache Flink. We present the results of a first experimental study showing the scalability of our implementation on social network data with up to 11 billion edges.

CCS Concepts
• Information systems → Graph-based database models; Parallel and distributed DBMSs;

Keywords
Graph Data Models, Graph Analytics, Apache Flink

1. INTRODUCTION
Graphs are a simple, yet powerful data structure to model and to analyze relationships between 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 [5] but also for business intelligence [9, 14, 15] and the life sciences [13]. In a graph, entities like web sites, users, products or proteins can be modeled as vertices while their connections are represented by edges.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

NDA’16, June 26–July 01 2016, San Francisco, CA, USA
© 2016 ACM. ISBN 978-1-4503-4513-2/16/06...$15.00
DOI: http://dx.doi.org/10.1145/2980523.2980527

Graph data models are a prerequisite for the execution of graph algorithms, for example, to rank web sites or to analyze social networks. However, complex analytical problems often cannot be solved by a single algorithm as they require a composition of different techniques. Let us give an example: An analyst is interested in the maximum common subgraph among the largest communities inside a social network. To answer this analytical question, we need to identify communities using a specific community detection algorithm, aggregate the number of vertices for each community, select communities above a minimum vertex count and finally apply a dedicated algorithm to find the maximum common subgraph. Furthermore, real-world graphs are usually heterogeneous in terms of the objects they represent and their attached data. For example, vertices of a social network may represent users, groups or bands while relationships may express friendships, memberships or interests. However, even entities of the same type may have heterogeneous attributes, for example, different users may provide more or less information about themselves.

To manage such data and to answer analytical questions like sketched in our example, we identified the following data model requirements: First, a data model must be able to represent single graphs (e.g., the social network) as well as graph collections (e.g., identified communities). Second, it needs to support heterogeneous attributes without a fixed schema not only for vertices and edges but also for graphs (e.g., vertex count). Third, the data model should provide general-purpose operators (e.g., selection by vertex count) as well as support for use-case specific algorithms (e.g., community detection). Fourth, it shall allow the combination of multiple operators and algorithms to analytical programs.

Since these requirements are not met by previous graph data models, we propose the Extended Property Graph Model (EPGM). The EPGM supports not only single but also collections of heterogeneous graphs and includes a wide range of combinable analytical operators. These operators fulfill the closure property as they take single graphs or graph collections as input and result in single graphs or graph collections. The EPGM is implemented as part of GRADOOD [12], a new system for scalable graph analytics on top of the Hadoop ecosystem. GRADOOD is GPLv3-licensed and publicly available1. To the best of our knowledge, this is the first expressive graph data model including native support for graph collections and respective operators implemented on a distributed computing system. Our main contributions can be summarized as follows:

1http://www.gradoop.com/
• We propose the EPGM, a graph data model that supports not only single graphs but also graph collections with heterogeneous vertices and edges. Our model includes declarative operators for graph analytics.

• We describe the first implementation of the EPGM on top of Apache Flink\(^2\), a state-of-the-art distributed dataflow framework.

• We present first experimental results to show the scalability of our implementation by applying an analytical program to a social network with up to 11 billion edges.

The remainder of this article is organized as follows: In section 2, we present the EPGM and its operators. Afterwards in section 3, we give a brief introduction to Flink, its programming concepts and how the EPGM is mapped to these concepts. The results of our first experiments are reported in section 4. Finally, we discuss related work in section 5 and conclude our work in section 6.

2. EXTENDED PROPERTY GRAPH MODEL

We introduce a data model extending the popular property graph model [17] by support for graph collections and by combinable analytical operators. Graph collections are a natural way to represent logical partitions of a graph, e.g., communities in a social network [7] or business process executions [14]. Further on, graph collections are the result of certain graph algorithms, e.g., embeddings found by graph pattern matching [8] or frequent subgraph mining [11]. The EPGM supports operators for graphs and graph collections as well as their composition to analytical programs. In the following, we will discuss graph representation and provided operators in more detail.

2.1 Graph Representation

In its basic form, a directed graph \(G = (V, E)\) consists of a set of vertices \(V\) and a set of binary edges \(E \subseteq V \times V\). Several extensions of this basic abstraction have been proposed to define a graph data model [2, 3]. One of these models, the property graph model (PGM) [17], gained wide acceptance and is used in many graph database systems (e.g., Neo4j\(^3\) or Titan\(^4\)). A property graph is a directed, labeled and attributed multigraph. To express heterogeneity, type labels can be defined for vertices and edges (e.g., Person or likes). Attributes have the form of key-value pairs (e.g., name: Alice or age: 42) and are referred to as properties. Such properties are set at the instance level without an upfront schema definition. In contrast to the directed and labeled graph model RDF\(^5\), attributes are encapsulated in vertices and edges.

With regard to the requirements stated in our introduction, the PGM is missing support for graph collections and associated operators. To meet all requirements, we have developed the Extended Property Graph Model (EPGM). In this model, a database consists of multiple property graphs which we call logical graphs. These graphs are application-specific subsets from shared sets of vertices and edges, i.e., may have common vertices and edges. Additionally, not only vertices and edges but also logical graphs have a type label and can have different properties. Formally, we define the EPGM database as follows:

\[\text{Definition 1 (EPGM database).} \quad \text{An EPGM database } DB = (V, E, L, \tau, K, A, \kappa) \text{ consists of a vertex set } V = \{v_1\}, \text{ an edge set } E = \{e_k\} \text{ and a set of logical graphs } L = \{G_m\}. \text{ Vertices, edges and (logical) graphs are identified by the respective indices } i, k, m \in \mathbb{N}. \text{ An edge } e_k = (v_i, v_j) \text{ with } v_i, v_j \in V \text{ directs from } v_i \text{ to } v_j \text{ and supports loops (i.e., } i = j). \text{ There can be multiple edges between two vertices differentiated by distinct identifiers. A logical graph } G_m = (V_m, E_m) \text{ is an ordered pair of a subset of vertices } V_m \subseteq V \text{ and a subset of edges } E_m \subseteq E \text{ where } \forall (v_i, v_j) \in E_m: v_i, v_j \in V_m. \text{ Logical graphs may potentially overlap such that } \forall G_i, G_j \subseteq L: |V(G_i) \cap V(G_j)| \geq 0 \land |E(G_i) \cap E(G_j)| \geq 0. \text{ For the definition of type labels we use label alphabet } T \text{ and a mapping } \tau : (V \cup E \cup L) \times \kappa : (V \cup E \cup L) \times K \rightarrow T. \text{ Similarly, properties (key-value pairs) are defined by key set } K, \text{ value set } A \text{ and mapping } \kappa : (V \cup E \cup L) \times K \rightarrow A.\]

Figure 1 shows an example EPGM database \(DB\) of a simple social network. Formally, \(DB\) consists of the vertex set \(V = \{v_0, \ldots, v_9\}\) and the edge set \(E = \{e_0, \ldots, e_19\}\). Vertices represent persons, forums and interest tags, denoted by corresponding type labels (e.g., Person) and are further described by their properties (e.g., name: Alice). Edges describe the relationships between vertices and also have type labels (e.g., knows) and properties. The key set \(K\) contains all property keys, for example, name, city and since, while the value set \(A\) contains all property values, for example, Alice, Leipzig and 2015. Vertices with the same type label may have different property keys, e.g., \(v_0\) and \(v_1\).

The sample database contains the set of logical graphs \(L = \{G_0, G_1, G_2\}\), where each graph represents a community inside the social network, in particular specific interest groups (e.g., Databases). Each logical graph has a dedicated subset of vertices and edges, for example, \(V(G_0) = \{v_0, v_1\}\) and \(E(G_0) = \{e_0, e_1\}\). Considering \(G_0\) and \(G_2\), one can see that vertex and edge sets may overlap since \(V(G_0) \cap V(G_2) = \{v_0, v_1\}\) and \(E(G_0) \cap E(G_2) = \{e_0, e_1\}\). Note that also logical graphs have type labels (e.g., Community) and may have properties, which can be used to describe the graph by annotating it with specific metrics (e.g., vertexCount: 3) or general information about that graph (e.g., interest: Databases).

Logical graphs, such as those of our example, are either declared explicitly or output of a graph algorithm, e.g., community detection or graph pattern matching. In both cases, they can be used as input for subsequent operators.

2.2 Operators

The EPGM provides operators for single logical graphs and graph collections; operators may also return single logical graphs or graph collections. Here, a graph collection \(G \in \mathcal{L}\) is a n-tuple of logical graphs and may contain duplicate elements. Collections are ordered to support application-specific sorting and position-based selection of logical graphs. In the following, we use the terms collection and graph collection as well as graph and logical graph interchangeably. Table 1 lists our analytical operators together with their corresponding pseudocode syntax for calling them in our domain specific language GrALa (Graph Analytical Language). The syntax adopts the concept of higher-order functions for several operators (e.g., to use aggregate or predicate functions as operator arguments). Based on the input of operators, we distinguish between graph operators and collection operators as well as unary and binary operators (single graph/collection vs. two graphs/collections as input). There are

---

\(^2\)http://flink.apache.org/

\(^3\)http://www.neo4j.com/

\(^4\)http://thinkaurelius.github.io/titan/

\(^5\)http://www.w3.org/RDF/
The resulting graph is a modified version of the input graph with maps an input graph and edges including respective labels and properties. In the following, we discuss the operators in more detail. Since the scope of this article is the data model and its analytical capabilities, complex operators (e.g., pattern matching and grouping) are only sketched and will be thoroughly described in future publications.

**Aggregation.** An operator often used in analytical applications is aggregation, where a set of values is mapped to a single value of significant meaning. In the EPGM, we support aggregation at the graph level. Formally, the operator applies the user-defined aggregate function \( \alpha : \mathcal{L} \to A \) and provides a property \( \nu \) and the graph structure. In the following example, we define \( \alpha \) as well as property keys and values, but preserve identifiers and properties (e.g., \{name \] Gdatabases G\}).

\[
\text{gamma} = (gIn, gOut => gOut[‘topic’] = gIn[‘interest’])
\]

\[
\text{nu} = (vIn, vOut => vOut.label = vIn[‘name’])
\]

\[
\text{epsilon} = (eIn, eOut => eOut.label = eIn.label)
\]

\[
\text{outGraph} = \text{db.G[0].transform(gamma, nu, epsilon)}
\]

The graph transformation function \( \text{gamma} \) takes the current graph instance \( gIn \) and the new graph instance \( gOut \) as input. The latter is a copy of the current graph with omitted type label and properties. The function determines that graph label and all graph properties are removed, except graph property interest, which is renamed to topic. The definition of vertex and edge transformations is analogous.

**Pattern Matching.** A fundamental operation of graph analytics is the retrieval of subgraphs isomorphic to a user-defined pattern graph. For example, given a social network, an analyst may be interested in all pairs of users who are members of the same forum with a specific tag.

To support such queries, we provide the pattern matching operator, where a pattern graph \( G^* \) and a predicate \( \varphi : \mathcal{L} \to \{\text{true}, \text{false}\} \) are the operator arguments. Pattern matching is applied to a graph \( G \) and returns a graph collection \( G' = \{G' \mid G' \subseteq G \land \varphi(G') = \text{true}\} \) containing all matches, for example:

\[
\text{outCollection} = \text{db.G.match(“(a:Person)<-[e:hasMember]-(b:Forum)}
\]

\[
(c:Person)<-[f:hasMember]-\text{db.G.match(“(b)-[g:hasTag]->(d:Tag (name = ‘Databases’))”)”}
\]

The shown pattern graph reflects our social network query. For GrALa, we adopted the basic concept of describing graph patterns using ASCII characters from Neo4j Cypher\(^6\), where \( (a)[-e]\) denotes an edge \( e \) from vertex \( a \) to vertex \( b \). The predicate function \( \varphi \) is embedded into the pattern by defining type labels and properties. In the example, we describe a pattern of four vertices and three edges, which are assigned to variables \( (a,b,c,d \text{ for vertices; } e,f,g \text{ for edges}) \). Variables are optionally followed by a label (e.g., \( a:Person \)) and properties (e.g., \( \text{name = ‘Databases’} \)). The operator is called for the logical graph representing the whole

\(^6\)http://neo4j.com/docs/2.3.1/cypher-query-lang.html
operator is also suitable to declare vertex-induced or edge-induced subgraphs respectively. 

The goal of this example is to group persons in the graph of Figure 1 by the city they live in and to calculate the number of group members. Furthermore, we want to group edges between users living in different cities as well as such living in the same city. First, we use the subgraph operator to describe the input graph for grouping consisting of all persons and their mutual relationships. In line 6, we define the vertex grouping keys. Here, we want to group vertices by their type label (denoted by the symbol :label) and property key city. Edges are grouped only by type label (line 9). In lines 7 and 10, we define the vertex and edge aggregate functions. Both receive the super entity (i.e., superVertex, superEdge) and the set of group members (i.e., vertices, edges) as input. Both functions apply the aggregate function count() to compute aggregated property values for grouped vertices and edges, e.g., the average age of persons in a group or the number of group members. The aggregate value is stored at the super vertex and super edge respectively. The following example shows the application of our grouping operator using GrALA:

outGraph = dbG .subgraph(  (v => v.label == 'Person'),  (e => e.label == 'hasModerator'))

We use nested predicate functions to filter vertices and edges based on the relevant type labels. Applied to the database graph (dbG), the operator returns a graph described through G′ = {⟨v1, v2⟩ | (v1, v2) ∈ E(G) ∧ ϕ((v1, v2)) = true ∧ v1, v2 ∈ V(G′)}. By omitting either a vertex or an edge predicate function exclusively, the operator is also suitable to declare vertex-induced or edge-induced subgraphs respectively.

**Grouping.** The groupBy operator determines a structural grouping of vertices and edges to condense a graph and thus helps to uncover insights about patterns hidden in the graph. Let G′ be the grouped graph of G, then each vertex in V(G′) represents a group of vertices in V(G); edges in E(G′) represent a group of edges between the vertex group members in V(G). More formally, V(G′) = {v′1, v′2, ..., v′n} where v′i is called a super vertex and ∀v ∈ V(G), s0(v) is the super vertex of v. Vertices are grouped based on their property values, such that for a given set of property keys Kv ⊆ K, ∀u, v ∈ V(G) : s0(u) = s0(v) ⇒ ∀k ∈ K

\[ K_v : k(u, k) = k(v, k) = k(s_0(u), k). \]

Furthermore, E(G′) = \{e1′, e2′, ..., e′n\} where e′i is called a super edge and s1(u, v) is the super edge of (u, v). Edge groups are determined among the super vertices and a set of edge keys K′e ⊆ K′e such that V′(u, v) : ⟨s, t⟩ ∈ E(G) : s1(u, v) = s1(s, t) ⇔ s0(u) = s0(s) ∧ s0(v) = s0(t) ∧ ∀k ∈ K′e : k(u, v, k) = k(s, t, k) = k(s0(u), s0(v), k). Additionally, the vertex and edge aggregate functions γv : P(V) → A and γe : P(E) → A are used to compute aggregated property values for grouped vertices and edges, e.g., the average age of persons in a group or the number of group members. The aggregate value is stored at the super vertex and super edge respectively. The following example shows the application of our grouping operator using GrALA:

outGraph = dbG .subgraph(  (v => v.label == 'Person'),  (e => e.label == 'hasModerator'))

 groupedBy(  [:label, 'city'],  (superVertex, vertices => superVertex['count'] = vertices.count()),  [:label],  (superEdge, edges => superEdge['count'] = edges.count()))

The goal of this example is to group persons in the graph of Figure 1 by the city they live in and to calculate the number of group members. Furthermore, we want to group edges between users living in different cities as well as such living in the same city. First, we use the subgraph operator to describe the input graph for grouping consisting of all persons and their mutual relationships. In line 6, we define the vertex grouping keys. Here, we want to group vertices by their type label (denoted by the symbol :label) and property key city. Edges are grouped only by type label (line 9). In lines 7 and 10, we define the vertex and edge aggregate functions. Both receive the super entity (i.e., superVertex, superEdge) and the set of group members (i.e., vertices, edges) as input. Both functions apply the aggregate function count()
on the set of grouped entities to compute the group size. The resulting value is stored as a new property count at the super vertex and super edge respectively. Figure 2 shows the resulting logical graph of the grouping example.

Binary Graph Operators. The EPGM includes binary graph operators to compare two input graphs. The equality operator determines if two logical graphs are equal according to a given equality function \( \xi : \mathcal{L} \times \mathcal{L} \rightarrow \{\text{true}, \text{false}\} \). We provide two implementations of \( \xi \): \text{:identity} and \text{:data}.

The first one determines equality based on vertex and edge identifiers, thus evaluates to true, if both input graphs contain the same instances. The second one compares the input graphs using a canonical form [11] representing labels and properties of the contained vertices and edges. Further binary graph operators are adopted from set-theory and determine the union (combination operator), intersection (overlap) and difference (exclusion) of two graphs resulting in a new graph. We denote these operators by dedicated terms to distinguish them from set-theoretic operators on graph collections based on graph identifiers.

Auxiliary Operators. In addition to the presented graph and collection operators, advanced graph analytics often requires the use of application-specific graph mining algorithms. One application can be the extraction of subgraphs that cannot be achieved by pattern matching, e.g., the detection of communities in a social network [7]. To support the plug-in of external algorithms, we provide generic \text{call} operators, which may have graphs and graph collections as input or output. Depending on the output type, we distinguish between so-called callForGraph (single graph result) and callForCollection operators.

Furthermore, it is often necessary to execute a unary graph operator on more than one graph, for example, to compute an aggregated value for all graphs in a collection. Not only the previously introduced operators aggregation, transformation and grouping, but all other operators with single logical graphs as in- and output (i.e., \( op : \mathcal{L} \rightarrow \mathcal{L} \)) can be executed on each element of a graph collection using the \text{apply} operator. Let \( G = (G_1, G_2, ..., G_n) \) be an input collection, then the output is \( G' = (op(G_1), op(G_2), ..., op(G_n)) \) under preservation of cardinality and order. The following example shows an aggregation which computes the edge count and is applied to all logical graphs in the collection inCollection:

\[
\text{outcollection} = \text{inCollection}.\text{apply}(g \mapsto g.\text{aggregate}(\text{"edgeCount"}, \text{(h \mapsto h.E.count())}))
\]

Similarly, in order to apply a binary operator on a graph collection, we adopt the \text{reduce} operator as often found in functional programming languages. The operator takes a graph collection and a commutative binary graph operator as input. The binary operator \( op : \mathcal{L} \times \mathcal{L} \rightarrow \mathcal{L} \) is initially applied on the first pair of elements of the input collection which results in a new graph. This result graph and the next element from the input collection are then the new arguments for the binary operator and so on. In this way, the binary operator is applied on pairs of graphs until all elements of the input collection are processed and the final graph is computed. In the following example, we call the reduce operator parametrized with the combine operator on all graphs in the given collection to compute a single graph:

\[
\text{outGraph} = \text{inCollection}.\text{reduce}(g, h \mapsto g.\text{combine}(h))
\]

3. IMPLEMENTATION

The most recent approaches to large-scale graph analytics are libraries on top of distributed dataflow frameworks, e.g., GraphX on Apache Spark [18] or Gelly on Apache Flink. These libraries are well suited for executing iterative graph algorithms on distributed graphs in combination with general data transformation operators provided by the underlying frameworks. However, the implemented graph data models have no support for collections and are generic,
which means arbitrary user-defined data can be attached to vertices and edges. In consequence, model-specific operators, for example, such based on properties, need to be user-defined, too. Hence, using those libraries to solve complex analytical problems becomes a laborious task.

We implemented the EPGM on top of Apache Flink to provide new features for graph analytics and to benefit from existing capabilities to large-scale data and graph processing at the same time. In this section, we will briefly introduce Flink and its programming concepts. We will further show how the EPGM graph representation and a subset of the introduced operators are mapped to those concepts.

### 3.1 Apache Flink

Apache Flink is the successor of the former research project Stratosphere [1] and supports the declarative definition and distributed execution of analytical programs on data flows sourced from streaming and batch data. The basic abstractions of such programs are data sets and transformations. A data set is a collection of arbitrary data objects and transformations describe the transition of one data set to another. For example, let \( X, Y \) be data sets, then a transformation could be seen as a function \( f: X \rightarrow Y \). Example transformations are map, where for each input object \( x \in X \) there is exactly one output object \( y \in Y \), and reduce, where all input objects are aggregated to a single one. Further transformations are well known from relational databases, e.g., join, group-by, project, union and distinct. To express application logic, transformations are parameterized with user-defined functions. A Flink program may include multiple chained transformations. When executed, Flink handles program optimization as well as data distribution and parallel execution across a cluster of machines.

### 3.2 Graph Representation

We use three object types to represent EPGM data model elements: graph head, vertex and edge. A graph head represents the data associated with a single logical graph. Vertices and edges also carry associated data, but additionally need to manage their graph membership as they may be contained in multiple logical graphs. In the following, we show a simplified definition of the respective types:

- **GraphHead**: \(<\text{Id}, \text{Label}, \text{Properties}>\)
- **Vertex**: \(<\text{Id}, \text{Label}, \text{Properties}, \text{GraphIds}>\)
- **Edge**: \(<\text{Id}, \text{Label}, \text{SrcId}, \text{TgtId}, \text{Properties}, \text{GraphIds}>\)

Each type contains an identifier (Id). As many EPGM operators create new entities (e.g., graph heads in binary graph operators and vertices/edges during grouping), we require identifiers to be generated independently in a distributed environment. Thus, we implemented identifiers using a 128-bit universally unique identifier\(^8\). 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, where the property key is of type string and the property value is encoded in a byte array. Our current implementation supports values of all primitive Java types. 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, we use a dedicated Flink data set for each element type. In our implementation, a logical graph is a special case of a graph collection where the graph head data set contains a single object. The runtime representation of graph \( G_0 \) in Figure 1 can be sketched as follows:

```
LogicalGraph g0 = {
  <GraphHead> graphHead = {
    <0, 'Community', ['interest': 'Databases', ...]>,
  },
  <Vertex> vertices = {
    <0, 'Person', ['name': 'Alice', ...], (0, 2)>,
    <1, 'Person', ['name': 'Bob', ...], (0, 2)>
  },
  <Edge> edges = {
    <0, 'knows', 1, 1, ('since': 2014), (0, 2)>,
    <1, 'knows', 1, 0, ('since': 2014), (0, 2)
  }
}
```

One can see, indicated by the respective graph identifiers, that associated vertices and edges are shared with logical graph \( G_2 \).

Since we use Flink data sets for graph representation, a graph analytical program is not limited to EPGM operators, but can also benefit from all libraries offered by Flink (e.g., for relational operations, machine learning or graph processing).

### 3.3 Operators

We implemented the GraALa domain specific language using the Java programming language. Each EPGM operator is mapped to a sequence of Flink transformations on the respective data sets. In Flink, program execution needs to be triggered explicitly, for example, by writing the result to a file or a database. As none of our operator implementations includes such triggers, multiple operators can be chained and executed as a single Flink program. We show the general idea of mapping graph operations to data set transformations for two of our operators. Listing 1 shows

---

\(^8\)docs.oracle.com/javase/7/docs/api/java/util/UUID.html
the implementation of the exclusion operator, where the resulting logical graph consists of vertices and edges that are contained in the first, but not in the second input graph. The idea of the implementation is to filter vertices and edges based on their graph membership. This is straightforward for vertices, however, for edges the implementation needs to ensure that source and target vertex are contained in the resulting vertex data set.

In lines 1 and 2, we extract the identifier of the second graph by applying a map transformation on its graph head data set. The transformation is parameterized with a user-defined function ID_ONLY, which extracts the identifier from a graph head. The resulting data set contains a single id object, which is then used to filter vertices of the first graph that are not contained in a graph with this id. The filter transformation takes a user-defined function NOT_IN_GRAPH_FILTER as argument and calls that function for each vertex in the data set. To make the graph id available to the filter function, we use Flink's concept of broadcasting (line 5) to send the data set to all workers of the cluster. The resulting data set then contains only vertices for which the filter function evaluates to true. For edges, we apply the same procedure to pre-filter edges, but also need to compute two semi-joins (line 9 to 12) to ensure that source and target vertex are contained in the new vertex set.

In line 13, we create a new logical graph. Its graph head including a new id is created by the constructor. Also, graph membership is updated for all vertices and edges contained in outV and outE using map transformations.

The concept of filter and broadcast is also used for the selection operator, whose implementation is presented in Listing 2. Here, we use the filter transformation to apply the user-defined predicate function on the graph head data set associated with the input collection. In line 3, we extract the identifiers of the filtered graph heads and use the resulting data set to filter those vertices and edges from the input collection that are contained in at least one of those filtered graphs. The latter is done by our IN_ANY_GRAPH_FILTER function, which evaluates to true, if the id set of the corresponding vertex or edge contains one of the given graph identifiers. In line 10, we create a new graph collection from the filtered graph heads, vertices and edges.

4. PRELIMINARY EXPERIMENTS

We evaluate our EPGM implementation on a cluster with 16 worker nodes. Each worker consists of an E5-2430 6(12) 2.5 Ghz CPU, 4GB RAM, two 4TB 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-SNAPSHOT (commit: adbeec2). We run Apache Flink with 12 threads and 40GB memory per worker.

We perform our experimental studies using datasets generated by the Graphalytics benchmark for graph processing platforms. The generator creates heterogeneous social network graphs that have a fixed schema similar to our example in Figure 1 and mimics structural characteristics of real-world networks [4]. Table 2 shows the datasets used throughout the benchmark. The scale factor (SF) denotes the increase of edges of type knows. Since our benchmark primarily involves vertices of type Person and edges of type knows, we added the particular ratio to the table.

The graph analytical program used for benchmarking is presented in Listing 3. The input is the entire social network as a single logical graph. First, we extract the subgraph containing only persons and their mutual relationships. The resulting graph is then transformed to a representation which is limited to information necessary for subsequent operators. Note, that we rename the vertex property property to see the relations between those groups. Edges are grouped along their incident vertices. By applying group-by (line 11) and group the combined graph by the vertex properties city and gender to see the relations between those groups. Edges are grouped along their incident vertices. By applying group-wise counting, we can find out how many vertices and edges are represented by their respective super entities. In lines 24 and 25, we use aggregation to compute how many super entities are contained in the resulting logical graph. The source code for our benchmark program is available online.9

In Figure 3, we show the results of our first experiments to evaluate the scalability of our implementation. For each con-
In the first experiment, we evaluated execution time for a fixed graph size (GA.100) but an increasing number of workers. In Figure 3(b), one can see execution times ranging from 1,057 seconds on a single worker to 104 seconds on 16 workers. Figure 3(c) shows the speedup for the same experiment and reveals a nearly linear speedup for up to 8 workers and a slight speedup decrease for 16 workers.

In the second experiment, we evaluated execution time of analytical programs on heterogeneous, schema-free graphs. Our first experiments show that we benefit from the underlying framework and its approach to distributed computing. The implementation is open-source, functioning and can be extended to new use cases. Our future work will focus on graph pattern matching and general improvements through graph partitioning, program optimization and reduced memory consumption.

7. ACKNOWLEDGMENTS

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

8. REFERENCES