Management and Analysis of Big Graph Data: Current Systems and Open Challenges

Martin Junghanns[1], André Petermann[1], Martin Neumann[2] and Erhard Rahm[1]

[1]Leipzig University, Database Research Group
[2]Swedish Institute of Computer Science
{junghanns,petermann,rahm}@informatik.uni-leipzig.de, mneumann@sics.se

Abstract. Many big data applications in business and science require the management and analysis of huge amounts of graph data. Suitable systems to manage and to analyze such graph data should meet a number of challenging requirements including support for an expressive graph data model with heterogeneous vertices and edges, powerful query and graph mining capabilities, ease of use as well as high performance and scalability. In this chapter, we survey current system approaches for management and analysis of "big graph data". We discuss graph database systems, distributed graph processing systems such as Google Pregel and its variations, and graph dataflow approaches based on Apache Spark and Flink. We further outline a recent research framework called Gradoop that is build on the so-called Extended Property Graph Data Model with dedicated support for analyzing not only single graphs but also collections of graphs. Finally, we discuss current and future research challenges.

1 Introduction

Graphs are ubiquitous and the volume and diversity of graph data are strongly growing. The management and analysis of huge graphs with billions of entities and relationships such as the web and large social networks were a driving force for the development of powerful and highly parallel big data systems. Many scientific and business applications also have to process and analyze highly interrelated data that can be naturally represented by graphs. Examples of graph data in such domains include bibliographic citation networks [40], biological networks [30, 110] or customer interactions with enterprises [88]. The ability of graphs to easily link different kinds of related information make them a promising data organization for data integration [90] as demonstrated by the so-called linked open data web[1] or the increasing importance of so-called *knowledge graphs* providing consolidated background knowledge [87], e.g., to improve search queries on the web or in social networks.

The flexible and efficient management and analysis of "big graph data" holds high promise. At the same time it poses a number of challenges for suitable implementations in order to meet the following requirements:

[1] http://lod-cloud.net/

- *Powerful graph data model:* The graph data systems should not be limited to the processing of homogeneous graphs but should support graphs with heterogeneous vertices and edges of different types and with different attributes without requiring a fixed schema. This flexibility is necessary for many applications (e.g., in social networks, vertices may represent users or groups and relationships may express friendships or memberships) and is important to support the integration of different kinds of data within a single graph. Furthermore, the graph data model should be able to represent and process single graphs (e.g., the social network) as well as graph collections (e.g., identified communities within a social network). Finally, the graph data model should provide a set of powerful graph operators to process and analyze graph data, e.g., to find specific patterns or to aggregate and summarize graph data.
- *Powerful query and analysis capabilities:* Users should be enabled to retrieve and analyze graph data with a declarative query language. Furthermore, the systems should support the processing of complex graph analysis tasks requiring the iterative processing of the entire graph or large portions of it. Such heavy-weight analysis tasks include the evaluation of generic and application-specific graph metrics (e.g., pagerank, graph centrality, etc.) and graph mining tasks, e.g., to find frequent subgraphs or to detect communities in social networks. If a powerful graph data model is supported, the graph operators of the data model should be usable to simplify the implementation of analytical graph algorithms as well as to build entire analysis workflows including analytical algorithms as well as additional steps such as pre-processing the input graph data or post-processing of analysis results.
- *High performance and scalability:* Graph processing and analysis should be fast and scalable to very large graphs with billions of entities and relationships. This typically requires the utilization of distributed clusters and in-memory graph processing. Distributed graph processing demands an efficient implementation of graph operators and their distributed execution. Furthermore, the graph data needs to be partitioned among the nodes such that the amount of communication and dynamic data redistribution is minimized and the computational load is evenly balanced.
- *Persistent graph storage and transaction support:* Despite the need for an in-memory processing of graphs, a persistent storage of the graph data and of analysis results is necessary. It is also desirable to provide OLTP (Online Transaction Processing) functionality with ACID transactions [55] for modifying graph data.
- *Ease of use / graph visualization:* Large graphs or a large number of smaller graphs are inherently complex and difficult to browse and understand for users. Hence, it is necessary to simplify the use and analysis of graph data as much as possible, e.g., by providing powerful graph operators and analysis capabilities. Furthermore, the users should be able to interactively query and analyze graph data similar to the use of OLAP (Online Analytical Processing) for business intelligence. The definition of graph workflows should be supported by a graphical editor. Furthermore, there should be support for

visualization of graph data and analysis results which is powerful, customizable and able to handle big graph data.

Numerous systems have been developed to manage and analyze graph data, in particular graph database systems as well as different kinds of distributed graph data systems, e.g., for Hadoop-based cluster architectures. *Graph database systems* typically support semantically rich graph data models and provide a query language and OLTP functionality, but mostly do not support partitioned storage of graphs on distributed infrastructures as desirable for high scalability (Section 2). The latter aspects are addressed by distributed systems that we roughly separate into distributed graph processing systems and graph dataflow systems. *Distributed graph processing systems* include vertex-centric approaches such as Google Pregel [78] and its variations and extensions including Apache Giraph [4], GPS [101], GraphLab [76], Giraph++ [109] etc. (Section 3). On the other hand, *distributed graph dataflow systems* (Section 4) are graph-specific extensions (e.g., GraphX and Gelly) of general-purpose distributed dataflow systems such as Apache Spark [118] and Apache Flink [21]. These systems support a set of powerful operators (map, reduce, join, etc.) that are executed in parallel in a distributed system separately or within analytical programs. The data between operators is streamed for a pipelined execution. The graph extensions add graph-specific operators and processing capabilities for the simplified development of analytical programs including graph data.

Early work on distributed graph processing on Hadoop was based on the *MapReduce* programming paradigm [103, 100]. This simple model has been used for the development of different graph algorithms, e.g., [75, 49, 71]. However, MapReduce has a number of significant problems [27, 81] that are overcome with the newer programming frameworks such as Apache Giraph, Apache Spark and Apache Flink. In particular, MapReduce is not optimized for in-memory processing and tends to suffer from extensive overhead for disk I/O and data redistribution. This is especially a problem for iterative algorithms that are commonly necessary for graph analytics and can involve the execution of many expensive MapReduce jobs. For these reasons, we will not cover the MapReduce-based approaches for graph processing in this chapter.

In this chapter, we give an overview about the mentioned kinds of graph data systems, and evaluate them with respect to the introduced requirements. In particular we discuss graph database systems and their main graph data models, namely the resource description framework [70] and the property graph model [97] (Sec. 2). Furthermore we give a brief overview about distributed graph processing systems (Sec. 3) and graph dataflow systems with focus on Apache Flink (Sec. 4). In Section 5, we outline a new research prototype supporting distributed graph dataflows called Gradoop (Graph analytics on Hadoop). Gradoop implements the so-called Extended Property Graph Data Model (EPGM) with dedicated support for analyzing not only single graphs but also collections of graphs. In Section 6, we compare the introduced system categories w.r.t. introduced requirements in a summarizing way. Finally, we discuss current and future research challenges (Sec. 7) and conclude.

2 Graph Databases

Research on graph database models started in the nineteen-seventies, reached its peak popularity in the early nineties but lost attention in the two-thousands [23]. Then, there was a comeback of graph data models as part of the NoSQL movement [35] with several commercial *graph database systems* [22]. However, these new-generation graph data models arose with only few connections to early rather theoretical work on graph database models. In this section, we compare recent graph database systems to identify trends regarding used data models and their application scope as well as their analytical capabilities and suitability for "big graph data" analytics.

2.1 Recent graph database systems

Graph database systems are based on a *graph data model* representing data by graph structures and providing graph-based operators such as neighborhood traversal and pattern matching [22]. Table 1 provides an overview about recent graph database systems including supported data models, their application scope and the used storage approaches. The selection claims no completeness but shows representatives from current research projects and commercial systems with diverse characteristics.

Supported data models: The majority of the considered systems supports one or both of two data models, in particular the property graph model (PGM) and the resource description framework (RDF). While RDF [70] and the related query language SPARQL [57] are standardized, for the PGM [97] there exists only the industry-driven de facto standard Apache TinkerPop[2]. TinkerPop also includes the query language Gremlin [96]. A more detailed discussion of both data models and their query languages follows in subsequent paragraphs.

A few systems are using generic graph models. We use the term *generic* to denote graph data models supporting arbitrary user-defined data structures (ranging from simple scalar values or tuples to nested documents) attached to vertices and edges. Such generic graph models are also used by most *graph processing systems* (see Section 3). The support for arbitrary data attached to vertices and edges is a distinctive feature of generic graph models and can be seen as a strength and a weakness at the same time. On the one hand, generic models give maximum flexibility and allow users to model other graph models like RDF or the PGM. On the other hand, such systems cannot provide built-in operators related to vertex or edge data as the existence of certain features like type labels or attributes are not part of the database model.

Application scope: Most graph databases focus on OLTP workload, i.e., CRUD operations (create, read, update, delete) for vertices and edges as well as transaction and query processing. Queries are typically focused on small portions of the graph, for example, to find all friends and interests of a certain user. Some of the considered graph databases already show built-in support for graph analytics, i.e., the execution of graph algorithms that may involve processing the whole

[2] `http://tinkerpop.apache.org/`

	Data Model			Scope		Storage		
	RDF/SPARQL	*PGM/TinkerPop*	*Generic*	*OLTP/Queries*	*Analytics*	*Approach*	*Replication*	*Paritioning*
Apache Jena TBD [5]	✓/✓			✓		native		
AllegroGraph [2]	✓/✓			✓		native	✓	
MarkLogic [12]	✓/✓			✓		native	✓	✓
Ontotext GraphDB [9]	✓/✓			✓		native	✓	
Oracle Spatial and Graph [13]	✓/✓			✓		native	✓	
Virtuoso [43]	✓/✓			✓		relational	✓	✓
TripleBit [117]	✓/✓			✓		native		
Blazegraph [16]	✓/✓	✓/✓		✓	✓	native RDF	✓	✓
IBM System G [33, 114]	✓/✓	✓/✓	✓	✓	✓	native PGM, wide column store	✓	✓
Stardog [15]	✓/✓	✓/✓		✓	○	native RDF	✓	
SAP Active Info. Store [99]		✓/-		✓		relational		
ArangoDB [11]		✓/✓		✓		document store	✓	✓
InfiniteGraph [10]		✓/✓		✓		native	✓	✓
Neo4j [83]		✓/✓		✓		native	✓	
Oracle Big Data [6]		✓/✓			✓	key value store	✓	✓
OrientDB [18]		✓/✓		✓		document store	✓	✓
Sparksee [79]		✓/✓		✓		native	✓	
SQLGraph [106]		✓/✓		✓		relational		
Titan [17]		✓/✓		✓	○	wide column store, key value store	✓	✓
HypergraphDB [61]			✓	✓		native		

Table 1. Comparison of Graph database systems

graph, for example to calculate the pagerank of vertices [78] or to detect frequent substructures [107]. These systems thus try to include the typical functionality of graph processing systems by different strategies. IBM System G and Oracle Big Data provide built-in algorithms for graph analytics, for example pagerank, connected components or k-neighborhood [33]. The only system capable to run custom graph processing algorithms within the database is Blazegraph by its gather-apply-scatter (see Section 3) API[3]. Additionally, the current version of TinkerPop includes the virtual integration of graph processing systems in graph databases, i.e., from the user perspective graph processing is part of the database system but data is actually moved to an external system. However, indicated by a circle in the analytics column in Table 1, we could identify only two systems currently implementing this functionality.

[3] `http://wiki.blazegraph.com/wiki/index.php/RDF_GAS_API`

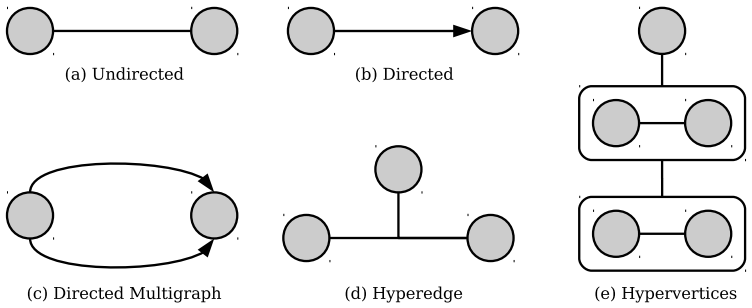


Fig. 1. Comparison of graph structures.

Storage techniques: The majority of the considered graph databases is using a so-called *native* storage approach, i.e., the storage is tailored to characteristics of graph database models, for example, to enable efficient edge traversal. A typical technique of graph-optimized storage are adjacency lists, i.e., storing edges redundantly attached to their connected vertices [33]. By contrast, some systems implement the graph database on top of alternative data models such as relational or document stores. IBM System G and Titan are offering multiple storage options. The used storage approach is generally no hint for database performance [106]. Most systems can utilize computing clusters by replicating the entire database on each node to improve read performance. About half of the considered systems also has some support for partitioned graph storage and distributed query processing. Systems with non-native storage typically inherited data partitioning from the underlying storage technique but provide no graph-specific partitioning strategy. For example, OrientDB treats vertices as typed documents and implements partitioning by type-wise sharding.

2.2 Graph data models

A graph is typically represented by a pair $G = \langle V, E \rangle$ of vertices V and edges E. Many extensions have been made to this simple abstraction to define rich graph data models [22, 23]. In the following, we introduce varying characteristics of graph data models with regard to the represented graph structure and attached data. Based on that, we discuss RDF and the property graph model in more detail.

Graph structures: Figure 1 shows a comparison of different graph structures. Graph structures mainly differ regarding their edge characteristics. First, edges can be either undirected or directed. While edges of an *undirected* graph (Fig. 1a) are 2-element sets of vertices, the ones of a *directed* graph are ordered pairs. The order of vertices in these pairs indicates a direction from *source* to *target* vertex. In drawings and visualizations of directed graphs, arrowheads are used to express edge direction (Fig.1b). In *simple* undirected or directed graphs, between any two vertices there may exist only one edge for undirected graphs and one edge in each direction for directed graphs. By contrast, *multigraphs* allow an arbitrary number of edges between any pair of vertices. Depending on the edge

definition, multigraphs are directed or undirected. Most graph databases use directed multigraphs as shown by Fig. 1c.

The majority of applied graph data models support only binary edges. A graph supporting n-ary edges is called *hypergraph* [39]. In a hypergraph model edges are non-empty sets of vertices, denoted by *hyperedges*. Fig. 1d shows a hypergraph with a ternary hyperedge. From the graph databases of Table 1 only HypergraphDB supports hypergraphs by default. A graph data model supporting edges not only between vertices but also between graphs is the *hypernode model* [91]. In this model we distinguish between primitive vertices and graphs in the role of vertices, the so-called *hypervertices*. Fig. 1e shows a graph containing hypervertices. Except an early research prototype, there is no graph database system explicitly supporting this data model. However, using the concept of *n-quads*, it is possible to express hypervertices using RDF [34].

Vertex- and edge-specific data: Another variation of graph data models relates to their support for data attached to the graph structure, i.e., their data content. Figure 2 illustrates different ways of attaching data to vertices and edges. The simplest form are *labeled graphs* where scalar values are attached to vertices or edges. For graph data management, labels are distinguished from identifiers, i.e., labels do not have to be distinct. An important special case of a labeled graph is a *weighted graph*, where edges show numeric labels (see Fig. 2a). Further on, labels are often used to add semantics to the graph structure, i.e., to give vertices and edges a type. Fig. 2b shows a vertex-labeled graph where labels express different types of vertices. A popular semantic model using vertex and edge labels is the *Resource Description Framework* (RDF) [70], where labels may be identifiers, blank or literals. Fig. 2c shows an example RDF graph.

Graph models supporting multiple values per vertex or edge are called *attributed*. Fig. 2d shows an example vertex-attributed graph. The shown graph is *homogeneous* as all vertices represent the same type of entities and show a fixed schema (name, age, gender). A popular attributed model used by commercial graph databases is the *property graph model* (PGM) [97]. A property graph is a directed multigraph where an arbitrary set of key-value pairs, so-called *properties*, can be attached to any vertex or edge. The key of a property provides a meaning about its value, e.g., a property `name:Alice` represents a name attribute with value Alice. Property graphs additionally support labels to provide vertex and edge types.

Resource Description Framework: In its core, RDF is a machine-readable data exchange format consisting of *(subject, predicate, object)* triples. Considering subjects and objects as vertices and predicates as edges, a dataset consisting of such triples forms a directed labeled multigraph. Labels are either internationalized resource identifiers (IRIs), literals such as numbers and strings or so-called *blank nodes*. The latter is used to reflect vertices not representing an actual resource. There are domain constraints for labels depending on the triple position. Subjects are either IRIs or blank nodes, predicates must be IRIs and objects may be IRIs, literals or blank nodes. In contrast to other graph models, RDF

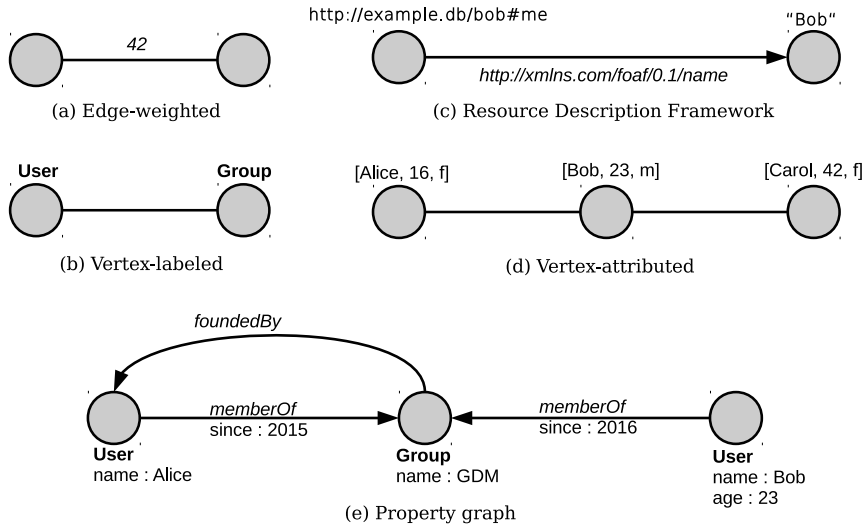


Fig. 2. Different variants of data attached to vertices and edges.

also allows edges between edges and vertices, which can be used to add schema information to the graph. For example, the type of an edge `:alice,:knows,:bob` can be further qualified by another edge `:knows,:isA,:Relationship`. A schema describing an RDF database is a further RDF graph containing metadata and is often referred to as *ontology* [31]. RDF is most popular in the context of the semantic web where its major strengths are standardization, the availability of web knowledge bases to flexibly enrich user databases and the resulting reasoning capabilities over linked RDF data [112]. Kaoudi and Manolescu [66] comprehensively survey recent approaches to manage large RDF graphs and consider additional systems not listed in Table 1.

Property Graph Model: While RDF is heavily considered in research, the PGM and its de-facto standard Apache TinkerPop found lower interest so far. However, many commercial graph database products use TinkerPop and the approach appears to gain public interest, e.g., in popularity rankings of database engines[4]. With one exception, all of the considered PGM databases support TinkerPop. The TinkerPop property graph model describes a directed labeled multigraph with properties for vertices and edges. Basically, the PGM is schema-free, i.e., there is no dependency between a type label and the allowed property keys. However, some of the systems, for example Sparksee, use labels strictly to represent vertex and edges types and require a fixed schema for all of their instances. Other systems like ArangoDB manage schema-less graphs, i.e., labels may indicate types but can be coupled with arbitrary properties at the same time. In most of the databases upfront schema definition is optional.

[4] http://db-engines.com/en/ranking/graph+dbms

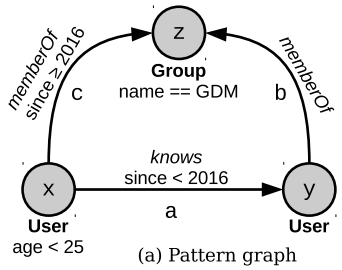
Property graphs with a fixed schema can be represented using RDF. However, representing edge properties requires reification. In the standard way[5], a logical relationship `db:alice,schema:knows,db:bob` is represented by a blank node `_:bn` and dedicated edges are used to express subject, object and predicate (e.g., `_:bn,rdf:subject,db:alice`). Properties are expressed analogously to vertices (e.g. `_:bn,schema:since,2016`). In consequence, every PGM edge is expressed by $3 + m$ triples, where m is the number of properties. Two of the graph databases of Table 1 store the PGM using RDF but both are using alternative, non-standard ways of reification. Stardog is using n-quads [34] for PGM edge reification. N-quads are extended triples where the fourth position is an IRI to identify a graph. Used for edge reification, each of such graphs represents an PGM edge [38]. Blazegraph follows a further, non-standard approach to reification and implements custom RDF and SPARQL extensions [58].

2.3 Query Language Support

In [22], Angles named four operators specific to graph databases query languages: adjacency, reachabilty, pattern matching and aggregation queries. *Adjacency* queries are used to determine the neighborhood of a vertex while *reachability* queries identify if and how two vertices are connected. Reachability queries are also used to find all vertices reachable from a start vertex within a certain number of traversal steps or via vertices and edges meeting given traversal constraints. *Pattern matching* retrieves subgraphs (embeddings) isomorphic to a given pattern graph. Pattern matching is an important operator for data analytics as it requires no specific start point but can be applied to the whole graph. Figure 3a shows an example pattern graph representing an analytical question about social network data. Finally, *aggregation* is used to derive aggregated, scalar values from graph structures. In contrast to Angles, we use the term aggregation instead of *summarization*, as the latter is also used to denote structural summaries of graphs [108]. Such summarization queries are not supported by any of the considered systems.

Most of the recent graph database systems either support SPARQL for RDF or TinkerPop Gremlin for the property graph model. Both query languages support adjacency, reachability, pattern matching and aggregation queries. Fig. 3c and 3d show example pattern matching queries equivalent to the pattern graph of Fig. 3a expressed in SPARQL and Gremlin. The result are pairs of `User`s who are member of the same `Group` with name `GDM`. Further on, one `User` should be younger than 25, member since 2016 and already knew the other user before 2016. The query was chosen to highlight syntactical differences and involves predicates related to labels and properties of vertices and edges. To support edge predicates, the SPARQL query relates to edge properties expressed by standard reification. While such complex graph patterns in SPARQL are expressed by a composition triple patterns and literal predicates (`FILTER`), the Gremlin equivalent is a composition of traversal chains, similar to the syntax of object-oriented programming languages.

[5] `https://www.w3.org/TR/rdf-schema/#ch_reificationvocab`



(a) Pattern graph

```
MATCH  (x:User)-[a:knows]->(y:User),
       (x)-[b:memberOf]->(z:Group),
       (y)-[c:memberOf]->(z)
WHERE  x.age < 25 AND z.name = 'GDM' AND
       a.since < 2016 AND c.since >= 2016
RETURN y.name, x.name
```

(b) Cypher

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX : <http://example.schema#>
SELECT ?y_name ?x_name
{
  ?x rdf:type :User;  :name ?x_name; :age ?x_age. FILTER(?x_age < 25).
  ?y rdf:type :User;  :name ?y_name.
  ?z rdf:type :Group; :name 'GDM'.
  ?a rdf:subject ?x; rdf:predicate :knows;    rdf:object ?y; :since ?a_since. FILTER(?a_since < 2016).
  ?b rdf:subject ?y; rdf:predicate :memberOf; rdf:object ?z .
  ?c rdf:subject ?x; rdf:predicate :memberOf; rdf:object ?z; :since ?c_since. FILTER(?c_since >=2016).
}
```

(c) SPARQL

```
g.V().match(
        __.as('x').has(label, 'User').and().has('age', lt(25)),
        __.as('y').has(label, 'User'),
        __.as('z').has(label, 'Group').and().has('name', 'GDM'),
        __.as('x').outE('knows').has('since', lt(2016)).inV().as('y'),
        __.as('y').out('memberOf').as('z'),
        __.as('x').outE('memberOf').has('since', gte(2016)).inV().as('z')
).select('y', 'x').by('name')
```

(d) Gremlin

Fig. 3. Comparison of pattern matching queries.

Beside this, there are also some vendor-specific query languages or vendor-specific SQL extensions. However, these languages miss pattern matching. A notable exception is Neo4j Cypher[7]. In Cypher, pattern graphs are described by ASCII characters where predicates related to vertices and edges are separated within a `WHERE` clause. Cypher is currently exclusively available for Neo4j but it is planned to make it an open industry standard similar to Gremlin. Participants of the respective openCypher[6] project are i.a. Oracle and databricks (Apache Spark), which could make Cypher available to more graph database and graph processing systems in future. A common limitation of SPARQL, Gremlin and Cypher is the representation of pattern matching query results in the form of tables or single graphs (SPARQL CONSTRUCT). In consequence, it is not possible to evaluate the embeddings in more detail, e.g., by visual comparison, and to execute any further graph operations on query results. A recently proposed solution to this problem is representing the result of pattern matching queries by a collection of graphs (see Section 5).

[6] http://www.opencypher.org/

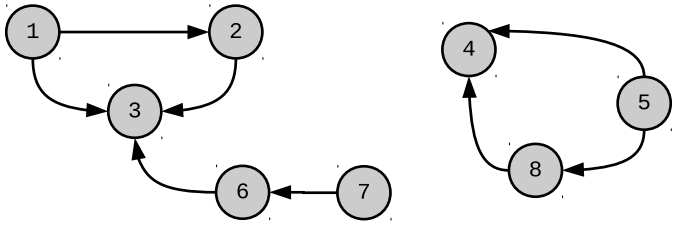


Fig. 4. Directed graph with two weakly connected components.

3 Graph Processing

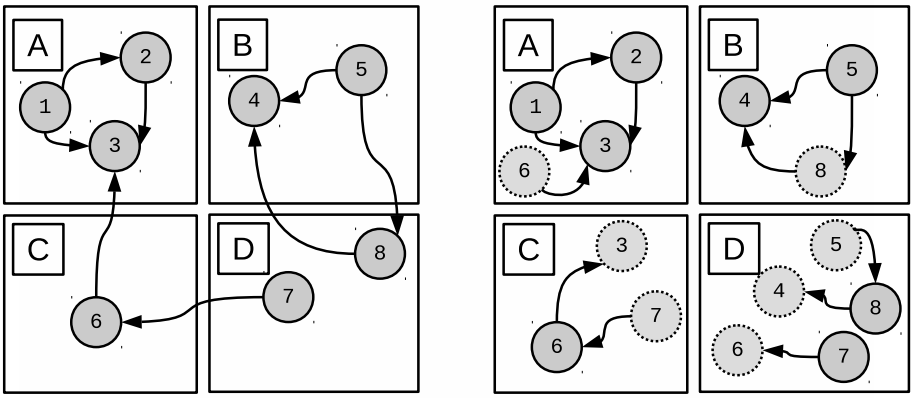
Many algorithms for graph analytics such as *pagerank*, *triangle counting* or *connected components* need to iteratively process the whole graph while other algorithms such as *single source shortest path* might require access to a large portion of it. Graph databases excel at querying graphs but usually cannot efficiently process large graphs in an iterative way. Such tasks are the domain of distributed graph processing frameworks.

In this section, we focus on dedicated distributed graph processing systems such as Pregel [78] and its derivates. More general dataflow systems like Apache Flink or Apache Spark, which also provide graph processing capabilities, will be discussed in the next section. Our presentation focuses on the popular vertex-centric processing model and its variations like partition- or graph-centric processing. To illustrate different programming models, we show their use to compute weakly connected components (WCC) of a graph. A connected component is a subgraph where each pair of vertices is connected via a path. For weakly connected components the edge direction is ignored, i.e., the graph is considered to be undirected. Figure 4 shows an example graph with two weakly connected components $V_{C_1} = \{1, 2, 3, 6, 7\}$ and $V_{C_2} = \{4, 5, 8\}$.

3.1 General architecture

The different programming models are based on a general architecture of a distributed graph processing framework. The architecture uses a *master node* for coordination and a set of *worker nodes* for the actual distributed processing. The input graph is partitioned among all worker nodes, typically using hash or range-based partitioning on vertex labels. In the vertex-centric model, a worker node stores for each of its vertices the vertex value, all outgoing edges including their values and vertex identifiers (ids) of all incoming edges. Figure 5a shows our example graph partitioned across four worker nodes A, B, C and D. Different frameworks extend upon this structure such as Giraph++ [109] where each worker node also stores a copy of each vertex that resides on a different worker but has a connection to a vertex on the worker node (Fig. 5b).

All graph processing systems discussed in this section use a directed generic multigraph model as introduced in Section 2. Vertices have a unique identifier K,



(a) Vertex-centric (b) Graph-centric

Fig. 5. Partitioned input graph for different computation models.

e.g., of type 64bit-integer. Vertices and edges may store a generic value further referred to as VV (vertex value) and EV (edge value). All frameworks allow the exchange of messages passed along edges and denoted by M.

3.2 Think Like a Vertex

The "Think Like a Vertex" or vertex-centric approach has been pioneered by Google Pregel in 2010 [78]. Ever since many frameworks have adopted or extended it [101, 68, 51, 74, 4, 105]. To write a program in a Pregel-like model, a so called *vertex compute function*[7] has to be implemented. This function consists of three steps: Read all incoming messages, update the internal vertex state (i.e., its value) and send information (i.e., messages) to its neighbors. Note that each vertex only has a local view of itself and its immediate neighbors. Any other information about the graph necessary for computation has to be sent along the edges. This paradigm is similar to the actor-based programming model [20] as implemented by Akka [1] or Quasar [14].

Vertex functions are executed in synchronized *supersteps*. In each superstep each worker node executes the compute function for all of its active vertices, marks them inactive if the `voteToHalt()` function is called and gathers their output messages. When all workers have finished, the gathered messages are delivered synchronously. Vertices that receive messages are then marked active. This is repeated until there is no active vertex at the end of a superstep. Note that the synchronization barrier between supersteps ensures that each vertex will only receive messages produced in the previous superstep. This execution model is called the *bulk synchronous parallel* (BSP) model [111]. Figure 6 shows an example of such an execution.

Let's see how WCC can be implemented using Apache Giraph [4], an open-source implementation of the Pregel model. Listing 1.1 shows a subset of Giraph's

[7] We use vertex compute function and vertex function interchangeably throughout this section.

```
long getSuperstep(); // returns the current iteration
void sendMsg(K id, M msg);
void sendMsgToAllEdges(M msg);
void voteToHalt();
K getVertexId();
VV getVertexValue();
void setVertexValue(VV vertexValue);
Iterator<K> getNeighbors();
Iterable<M> getMessages();
void aggregateValue(aggregatorName, aggregatedValue);
AV getAggregatedValue(aggregatorName);
```

Listing 1.1: Subset of the Apache Giraph API used to write a vertex function.

API that is used to implement the vertex function. The `getSuperstep()` function allows to write algorithms that change behavior depending on the current superstep. This is often used for initialization. As mentioned before, `voteToHalt()` tells the framework that the vertex program should not be executed for this particular vertex in the next superstep unless the vertex receives any messages. Note that this is vital for the termination of the program and should be called. The other functions allow the user to access the vertex identifier, incoming messages and neighboring vertex identifiers.

Listing 1.2 shows a (simplified) implementation of WCC using the introduced API. The basic idea is that vertices propagate their label along the edges until

```
void compute(Vertex v) {
  if (getSuperstep() == 0)
    v.setValue(v.getVertexID())
    sendMessageToAllEdges(v.getVertexValue())
  else
    minValue = min(v.getMessages())
    if (minValue < v.getVertexValue())
      v.setVertexValue(minValue)
      sendMessageToAllEdges(v.getVertexValue())
  v.voteToHalt();
}

void combine(M message1, M message2) {
  return min(message1, message2)
}
```

Listing 1.2: WCC in Apache Giraph. The vertex function will be executed for each vertex in the graph. Messages sent by the vertices are stored at the worker and delivered at the end of the current superstep. The execution loops until no vertex has received any message.

convergence. After termination, each vertex stores a component id which will be equal to the smallest vertex id that can be reached from this vertex. This value will be the same for each vertex in a component and thus identifies a component. In superstep 0, we initialize the component id with the vertex id and send the value to all neighbors. In each subsequent superstep, each vertex computes the smallest component id among all received messages; if it is smaller than the currently stored value, it is replaced and the new value is sent to all neighbours. Each vertex always votes to halt at the end of each superstep. As a result, no message will be sent, if no vertex has changed its component id within a superstep and the algorithm terminates. Figure 6 shows the WCC execution for the graph shown in Figure 5a resulting in two connected components represented by the identifiers 1 and 4.

Variants Various vertex-centric graph processing systems provide specific features and optimization techniques, for example, to mutate the graph or to reduce network traffic and computation time. In the following, we will discuss the most differentiating features as shown in Table 2.

Aggregation: Certain graph algorithms need global knowledge in terms of aggregated values such as the number of vertices in the graph or the total sum of all vertex values. In the basic model, this can be achieved by creating a vertex that is connected to all other vertices. However, this approach creates vertices with a huge amount of incident edges that will take longer to process than a regular graph vertex. This will decrease performance since workers have to wait for each other at the end of each superstep. Additionally, these special purpose edges and vertices require specific programming logic in the vertex program which increases complexity. Many frameworks (see column aggregation in Table 2) require the user to provide a function that is run on the master node between *supersteps* for this purpose. For example, to calculate the sum of all vertex values, each vertex would send its value to the master node (API method `aggregateValue()`,

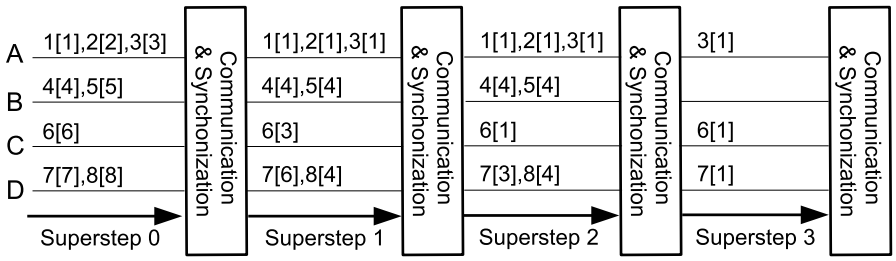


Fig. 6. Vertex-centric WCC computation for the graph of Figure 4. We show the vertex value at the end of each superstep. Initially, the vertices use their ids as initial vertex values. In any superstep, each vertex changes its value to the minimum among all messages and its own value. A vertex function will only be executed in the initial superstep of if the vertex has received any messages in the previous superstep. Looking at vertex 1 and 7, one can see how vertex id 1 is propagated through the component. Note that we omitted the 4th superstep that solely consists of vertex 6 processing a message received from vertex 7.

	language	*Programming Model*	*BSP*	*asynchronous*	*generic scheduler*	*aggregation*	*add vertex/edge*	*remove vertex/edge*	*combiner*
Pregel [78]	C++	Pregel	✓						
Giraph [4]	Java	Pregel	✓			✓	✓	✓	✓
GPS [101]	Java	Pregel	✓			✓		✓	✓
Mizan [68]	C++	Pregel	✓			✓			✓
GraphLab [76]	C++	GAS	✓	✓	✓	✓	✓		n.a.
GraphChi [74]	C++, Java	Pregel		✓	✓	✓	✓	✓	n.a.
Signal/Collect [105]	Java	Scatter-Gather	✓	✓					n.a.
Chaos [98]	Java	Scatter-Gather	✓	✓					n.a.
Giraph++ [109]	Java	Partition-Centric	✓			✓	✓	✓	✓
GraphX [52]	Scala, Java	GAS	✓			✓			n.a.
Gelly [8]	Scala, Java	GSA, Scatter-Gather	✓			✓	✓	✓	n.a.

Table 2. Key features of the discussed graph processing systems (n.a., not applicable).

Listing 1.1) which aggregates them and makes the results accessible in the next superstep (`getAggregatedValue()` in Listing 1.1). Note, that for associative and commutative operations, such as counting or summation, this can be done in an aggregation tree where the worker node will aggregate the values of all its vertices before sending the aggregated value to the master, therefore reducing communication costs.

Reducing network communication: A technique, similar to the one used for aggregation, can also be used to reduce the number of messages between different worker nodes. If a worker node has multiple messages addressing the same vertex, they can potentially be combined into a single message. In some of the frameworks (Table 2) the user can define a combiner, a dedicated function that takes two messages as input and combines them into one. Listing 1.2 includes the combine function for WCC as implemented in Giraph. In our WCC implementation, we are only interested in the smallest value, so the combiner can discard the larger message. With this combiner, no vertex will receive more messages than the number of worker nodes.

Powergraph [51] further extended the idea of the combiner by introducing the ***G**ather-**A**pply-**S**catter* (GAS) model. Instead of a single vertex compute function, the user has to provide a gather, apply and scatter function. The gather function has the same functionality as the combiner: it aggregates messages addressing the same vertex on the sending worker nodes. The apply function has the incoming messages as input and updates the vertex state. The scatter function has the vertex state as input and produces the outgoing messages. Similar to the gather function, the scatter function can be executed on the worker nodes. Instead of sending multiple messages from one vertex to vertices on the same worker node, only the vertex value is send and the messages are

then created locally. This execution is transparent to the user which only has to provide the three functions. The GAS model is especially effective on graphs with highly skewed degree distributions. It not only reduces the amount of network traffic, but also helps balancing the workload between worker nodes by spreading out the computation. One downside of the GAS model is that all information about messages that should eventually be sent needs to be part of the vertex value. In case of WCC, we need to extend the vertex value by a boolean field that reflects if the vertex value has changed or not to decide if messages should be sent.

The systems Signal/Collect [105] and Chaos [98] introduced the *Scatter-Gather model*. This model requires the user to provide an edge and a vertex function. The vertex function has all incoming messages as input and can modify the vertex value. The edge function takes the vertex value as input and can then generate a message. Compared to the GAS model, in the scatter-gather model, the computation is parallelized across the vertices, which may lead to unbalanced load, if the edge degree distribution is skewed. Depending on the computation, the execution time for high-degree vertices increases as they need to process more messages and thus the synchronization barrier is eventually delayed.

Asynchronous execution: Looking at Figure 6, one can see that worker node A takes longer to compute the vertex function on all of its vertices. In consequence, the faster working nodes B, C, and D have to wait. Not all algorithms require the strict synchronization offered by the BSP execution model. Our WCC implementation in Listing 1.2 tries to find the minimum vertex id in each component. Finding the minimum of a set does not require a specific execution order and can be executed without synchronization. If a worker node is a superstep behind and does not deliver its messages in time, the minimum of each component will eventually be found once the delayed messages are delivered. The overall execution time can be potentially reduced since worker nodes do not spend time waiting for other workers to finish. Furthermore, some algorithms [29, 73] converge much faster on an asynchronous execution model up to the point where running them in a BSP model will not converge in reasonable time. Other graph algorithms such as Ja-be-Ja [93], a peer-to-peer inspired graph partitioning algorithm, can only be implemented using an asynchronous execution model. To address these challenges GraphLab [76], Signal/Collect and GraphChi [74] allow for asynchronous execution. Instead of waiting for a synchronization barrier, in these models, messages produced by a vertex will be delivered to the target vertex directly. Each worker node processes its vertices in order, thus, within a partition, each vertex will be executed with the same frequency. However, in Figure 5a, vertex 6 on worker node C might already have executed ten times while vertex 1 on worker node A has only executed once. GraphLab and GraphChi also allow the user to provide a scheduler function that changes the execution order, for example, to prioritize vertices with a high value. This allows to focus an algorithm on a certain part of the graph, which can lead to faster convergence in some cases.

```
boolean containsVertex(K id);
boolean isInternalVertex(K id);
boolean isBoundaryVertex(K id);
Vertex<K, VV, EV, M> getVertex(K id);
Collection<Vertex<K, VV, EV, M>> internalVertices();
Collection<Vertex<K, VV, EV, M>> activeInternalVertices();
Collection<Vertex<K, VV, EV, M>> boundaryVertices();
Collection<Vertex<K, VV, EV, M>> allVertices();
```

Listing 1.3: Additional functions in the Giraph++ API.

Note, that optimizations such as combiners or the GAS model cannot use their full potential when executed asynchronously since messages are not necessarily batched together. As a result, asynchronous execution generally uses more network resources. Performance gains are hard to quantify since the speedup highly depends on the graph structure and to which degree work is equally distributed between worker nodes. For our WCC example, each superstep might be faster due to the removal of the synchronization barrier but the algorithm might require more steps to terminate. In the BSP execution, the number of required steps is equal to the longest shortest path in the graph since each vertex processes the data from all its neighbors in each superstep. In an asynchronous execution it is possible that the message with the true minimum is delayed so that there are additional steps finding the minimum between larger values before the true minimum is found.

Graph mutation: Transformational algorithms such as graph coarsening or computing the minimum spanning tree need to modify the graph structure during execution. This is a non-trivial task since it may lead to load imbalances, performance loss and memory overflow. Currently, only few frameworks support these operations. For example, while Giraph supports adding and removing vertices and edges, GraphLab only allows addition. Vertices are added or removed from inside the vertex function and the changes to the graph, similar to messages, become visible in the next superstep. Newly created vertices are always marked as active in the superstep they appear in and therefore guaranteed to be executed.

3.3 Think like a Graph

Instead of writing a compute function executed on each vertex, in a graph-/partition-centric model, the user provides a compute function that takes all vertices managed by a worker node as input. These functions are then executed using the BSP model. This approach requires additional support structures when distributing the graph. The input graph is distributed across worker nodes in the same way as for vertex-centric computations. The vertices of worker node n are called internal vertices to n. On each worker node n we then create a copy of each vertex that is not internal to n, but is directly connected to an internal

```
1  void compute() {
2    if (getSuperstep() == 0)
3      sequentialCC();
4      for (bV in boundaryVertices())
5        sendMsg(bV.getVertexId(), bV.getVertexValue())
6    else
7      equiCC = new MultiMap;
8      for (iV in activeInternalVertices())
9        minValue = min(iV.getMessages())
10       if (minValue < iV.getVertexValue())
11         equiCC.add(iV.getVertexValue(), minValue)
12         for (v in allVertices())
13           minValue = equiCC.getMinFor(v.getVertexValue())
14           if (minValue < v.getVertexValue())
15             v.setVertexValue(minValue)
16             if (isBoundaryVertex(v.getId())
17               sendMsg(v.getVertexId(), vertex.getVertexValue())
18   allVoteToHalt()
19 }
20
21 void combine(M message1, M message2) {
22   return min(message1, message2)
23 }
```

Listing 1.4: WCC in Giraph++. First each worker node finds all internal connected components. Then it iteratively shares the information with other worker nodes that have vertices connected.

vertex of n. These vertices are called *boundary vertices* and represent the cached vertex values of copied vertices. Every internal vertex may have up to one of these boundary vertices on each worker node. Figure 5b shows the distributed graph with internal and boundary vertices on the four worker nodes.

Listing 1.3 shows the additional methods of the Giraph++ [109] API. Having a partition compute function instead of a vertex compute function allows direct access to all internal and local boundary vertices and thus computing the entire subgraph. Each worker node executes its user-defined function and afterwards sends messages from all boundary vertices to their internal representation. The partition-centric model can mimic a vertex centric execution by iterating through all active internal nodes once in each superstep. Listing 1.4 shows a partition-centric implementation of WCC. In the initialization step, a sequential connected component algorithm is executed finding all local connected components. The locally computed component label for each boundary vertex is then sent to its corresponding internal vertex. In each of the subsequent supersteps, the algorithm processes all the incoming messages and merges labels representing the same component. Although the implementation of this approach is more complex, it can reduce the amount of iterations and thus improve performance.

Vertex	Step 0	Step 1	Step 2	Step 3
1	1	1	1	1
2	2	1	1	1
3	3	1	1	1
4	4	4	4	4
5	5	4	4	4
6	6	3	1	1
7	7	6	3	1
8	8	4	4	4

Table 3. Vertex states/values in vertex-centric iteration.

Vertex	Step 0	Step 1
1	1	1
2	1	1
3	1	1
4	4	4
5	4	4
6	6	1
7	7	1
8	8	4

Table 4. Vertex state/values in graph-centric iteration.

The number of steps required to converge is smaller or equal to the longest shortest path in the graph. The precise number of saved iteration steps depends on the graph structure, in particular on how vertices are distributed among the worker nodes.

Tables 3 and 4 show the convergence in vertex- and graph-centric iterations respectively. One can see, that it takes four *supersteps* for a vertex-centric iterations whereas using a graph-centric approach, the components can be computed in only two *supersteps*. Notice that the reduction in *supersteps* depends on the partitioning of the input graph. A partitioning where each component resides on a single worker node requires zero *supersteps*, while the worst case partition would require the same amount of *supersteps* as a vertex centric program. The performance gain can be hard to predict and cannot justify the additional complexity of the program in all cases.

In this section we gave an overview about the different dedicated graph processing frameworks available. We summarized the most common programming models and shown their variants. In real-world scenarios, graph processing is often only a single step of a longer pipeline consisting of data transformations. Therefore modern processing frameworks such as Apache Spark and Apache Flink provide graph processing libraries that can be directly integrated into a larger program. These libraries support vertex-centric graph processing with additional graph operations that can be combined with general-purpose data operations on structured and unstructured data.

4 Graph Dataflow Systems

In the previous section, we introduced specialized systems providing tailored programming abstractions for the fast execution of a single iterative graph algorithm on large graphs with billions of vertices and edges. However, complex analytical problems often require the combination of multiple techniques, for example, to create combined graph structures based on unstructured or structured data originated from different sources (e.g., distributed file systems, database systems) or to combine graph algorithms and non-graph algorithms (e.g., for machine learning). In such cases, using dedicated systems for each part of an analytical program increases the overall complexity and leads to unnecessary data movement between systems and respective data duplication [52, 116].

By contrast, distributed in-memory dataflow systems such as Apache Spark [52, 116, 115, 118], Apache Flink [21] or Naiad [82, 85] provide general-purpose operators (e.g., map, reduce, filter, join) to load and transform unstructured and structured data as well as specialized operators and libraries for iterative algorithms (e.g., for machine learning and graph analysis). Using such a system for the implementation of complex analytical programs reduces the overall complexity for the user and may lead to performance improvements since the holistic view on the whole program enables optimizations, such as operator reordering or caching of intermediate results.

In this section, we will discuss graph analytics on distributed dataflow systems using Apache Flink as a representative system. We briefly introduce Apache Flink and its concept for iterations and will then focus on Gelly, a graph processing library integrated into Apache Flink. Gelly implements the Scatter-Gather and Gather-Sum-Apply programming abstractions for graph processing and provides additional operators for graph transformation and computation. We will finish the section with a brief comparison to GraphX, a graph library on Apache Spark.

4.1 Apache Flink

Apache Flink is the successor of the former research project Stratosphere [21] and supports the declarative definition and distributed execution of analytical programs on batch and streaming dataflows.[8] The basic abstractions of such programs are *datasets* and *transformations*. A dataset is a collection of arbitrary data objects and transformations describe the transition of one dataset to another one. For example, let X, Y be datasets, then a transformation could be seen as a function $t : X \to Y$. Example transformations are *map*, where for each input object $x_i \in X$ there is exactly one output object $y_i \in Y$, 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.

We give an exemplary introduction to the dataset API using a simple word count program to compute the frequency of each word in an input text (Listing 1.5). We first create a Flink execution environment (Line 1), which abstracts either a local machine (e.g., for developing and testing) or a cluster. In Line 2, we define an input data source, here a file from HDFS, the Hadoop Distributed File System.[9] The resulting dataset contains strings whereas each string represents a line in our input file. In Line 6, we use *flatMap* to declare the first transformation on our input dataset. This transformation allows us to output

[8] In its core, Flink is a distributed streaming system and provides streaming as well as batch APIs. We focus on the batch API, as Gelly is currently implemented on top of that.

[9] Flink supports further systems as data source and sink, e.g., relational and NoSQL databases or queuing systems.

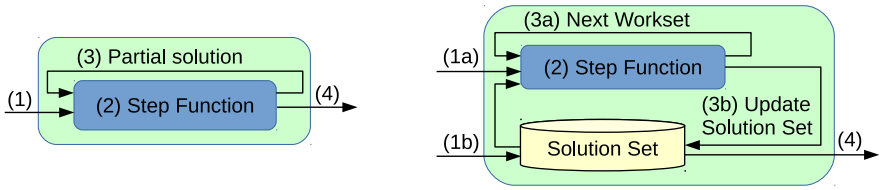
```
1  ExecutionEnvironment env = getExecutionEnvironment();
2  DataSet<String> text = env.readTextFile("hdfs:///text");
3
4  DataSet<Tuple2<String, Integer>> wordCounts = text
5      // splits the line and outputs (word, 1) tuples
6      .flatMap(new LineSplitter())
7      // group tuples by word
8      .groupBy(0)
9      // add together the "1"s in all tuples per group
10     .sum(1);
11
12 wordCounts.print();
```

Listing 1.5: Word Count in Flink

an arbitrary number of objects for each input object. Here, the user-defined function `LineSplitter` is applied on each line in the input dataset and splits it into words. For each word, the function outputs a tuple containing the word and the frequency 1, for example, the line *"graphs are everywhere"* results in the tuples (*"graphs"*, 1), (*"are"*, 1) and (*"everywhere"*, 1). In Line 8, we perform a *group-by* transformation on the output dataset of the previous *flatMap* transformation to gather all tuples that represent the same word. In Line 10, we add together the single frequencies to get the total frequency for each word using *sum*, a predefined aggregation transformation. Flink programs are executed lazily, i.e., program execution needs to be started explicitly. Here, we trigger the execution by printing the dataset to system console (Line 12). When triggered, Flink analyzes the program, optimizes it and executes it in the specific environment. Data lines are read in parallel from the data source and "flow" through the transformations which are scaled-out to all workers in the cluster.

Iterations in Apache Flink Our word count example represents a dataflow whose execution graph is a directed acyclic graph of transformations. However, iterative or recursive graph and machine learning algorithms require cyclic execution graphs. To support cyclic dataflows, Flink offers two specialized operators: Bulk and Delta Iteration [46, 45].

With *Bulk Iteration* (Fig. 7a), each iteration computes a new solution based on the previous iteration result which is then used as input for the next iteration. Conceptually, Flink's Bulk Iteration can be separated into four phases: (1) the *iteration input* is the initial dataset for the first iteration; (2) the *step function* takes the output of the previous iteration as input and executes an acyclic dataflow containing arbitrary transformations on that dataset to create a new dataset; (3) the result of the step function is the *next partial solution*, which is used as input for the next iteration; (4) the *iteration result* is the dataset created by the last iteration and can be used in subsequent dataflows. The convergence criterion for the Bulk Iteration is either a maximum number of iterations or a custom convergence criterion.



(a) Bulk Iterate Operator (b) Delta Iterate Operator

Fig. 7. Iteration Operators in Apache Flink [3].

With *Delta Iteration*, each iteration computes only incremental updates for an evolving global solution set instead of a completely new solution set. The motivation for this approach are algorithms where an update on one element has a direct impact only on a small number of other elements, such that different parts of the solution may converge at different speeds [45]. When applicable, this leads to faster convergence as large parts of the solution are computed in the first iterations so that later iterations compute on much smaller subsets. Figure 7b shows the phases of Flink's Delta Iteration: (1) In contrast to Bulk Iteration, we now have two input datasets: a) the *initial workset* and b) an *initial solution set* which evolves with each iteration; (2) the step function again performs an acyclic dataflow of arbitrary transformations on both the current workset and the solution set; (3) the outputs of the step function are a) the *update solution set*, which contains incremental updates for the initial solution set and b) the *next workset*, which is the input for the next iteration; (4) the solution set after the last iteration is the *iteration result* and can again be used in subsequent dataflows. In contrast to Bulk Iteration, the iteration terminates if the produced next workset is empty or a maximum number of iterations is reached. However, it is also possible to define a custom convergence criterion.

With reference to the introduced programming models for graph processing in Section 3, each iteration in the Bulk and Delta Iteration can be seen as a super step in a synchronous BSP process. Multiple instances of the step function are executed in parallel and synchronized at the end of each iteration. In Section 3, we also showed that for specific graph algorithms, for example, connected components or single-source-shortest-path, not all vertices are necessarily active in each super step.[10] The Delta Iteration is a good foundation for this class of algorithms which is why Gelly uses it to implement vertex-centric programming abstractions, which we will discuss next.

4.2 Apache Flink Gelly

Flink Gelly [8] is a graph library integrated into Apache Flink and implemented on top of its dataset API. Besides dedicated graph processing abstractions, Gelly provides a wide set of additional operators to simplify the definition of graph

[10] When implemented using a synchronous graph-processing system.

analytical programs. The provided data model is a directed, labeled multigraph where vertex and edge labels are generic, i.e., vertices and edges can carry arbitrary user-defined payload ranging from basic data types such as numbers and strings to complex domain objects. In the following, we will discuss the graph representation on Flink's dataset API, transformation methods and how graph processing abstractions are mapped to the Delta Iteration.

Graph Representation Gelly uses two classes to represent the elements of a graph: `Vertex` and `Edge`. A `Vertex` comprises a comparable, unique identifier (id) and a value, an `Edge` consists of a source vertex id, a target vertex id and an edge value. Identifiers and values are generic and need to be declared upon graph creation. Internally, a graph is represented by a dataset of vertices and a dataset of edges as shown below:

```
class Graph<K, VV, EV> {
    DataSet<Vertex<K, VV>> vertices
    DataSet<Edge<K, EV>> edges
}
```

The generic type `K` represents the vertex id type, `VV` the vertex value type and `EV` the edge value type. Since Gelly offers methods to return the vertex and edge datasets, an analytical program can combine those datasets with any other library in Flink (e.g., for machine learning) as well as third-party libraries that are implemented on the dataset API (e.g., Gradoop in Section 5). A Gelly graph provides basic methods for creating graphs and returning simple metrics such as vertex count, edge count or in- and out-degrees of vertices, which result in new datasets for further processing.

Graph Transformations Graph transformation methods are applied on an input graph and return a new, possibly modified graph, hence enabling the composition of multiple graph transformations in an analytical program. Internally, Gelly translates each graph transformation to a series of transformations on the vertex and edge datasets. Similar to other graph dataflow frameworks, e.g., GraphX [52, 116, 115], Gelly offers the following transformation methods:

- **Mutation** methods enable adding and removing of vertices and edges. The result is a new graph with an updated vertex and edge dataset respectively.
- **Map** allows the modification of vertex and edge values by applying user-defined transformation functions on all elements in the corresponding datasets.
- **Subgraph** enables the extraction of a new graph based on user-defined vertex and edge predicates. If an element in the input graph satisfies the predicate, it is contained in the output graph.
- **Join** allows the combination of vertex and edge datasets with additional input datasets. The transformation applies a user-defined function on each matching pair and returns a graph with a updated datasets. This can be useful to attach external data, e.g., from a relational database, to the graph.
- **Undirected** can be used to transform a directed graph into an undirected graph by cloning and reversing all edges.

– **Union/Difference/Intersect** enable merging of two graphs into a new graph based on the respective set-theoretical method applied on vertex and edge datasets.

Neighborhood Methods Neighborhood methods are applied on all incident edges and adjacent vertices of each vertex and can be used to aggregate edge and vertex values (e.g., average/min/max values, vertex degree, etc.). Gelly provides two variants of neighborhood methods:

– **reduceOnEdges/Neighbors** allow the aggregation of edge and vertex values by providing a user-defined, associative and commutative function on pairs of values. The methods result in a new dataset containing exactly one aggregate per vertex.
– **groupReduceOnEdges/Neighbors** allow the aggregation of edge and vertex values by providing a user-defined, non-associative, non-commutative function on all respective values. This is useful, if one needs to have all values available in the function or if more than one aggregate needs to be computed per neighborhood. The methods result in new datasets containing an arbitrary amount of aggregates for each vertex.

Graph Processing In Section 3, we introduced various programming abstractions for graph processing. Gelly currently adopts two variants of vertex-centric iterations: Scatter-Gather and Gather-Sum-Apply. Both are implemented using the Delta Iteration operator and are thus executed in synchronous super steps. In the following, we will discuss both abstractions in further detail.

The Scatter-Gather abstraction is adopted from the Signal/Collect model [105] and divides a super step in two phases. In the Scatter (or messaging) phase, the messages sent to other vertices are being produced, while in the Gather (or update) phase each vertex updates its value using the received messages. The user needs to implement both, a messaging and an update function, which are applied during the computation. Picking up the running example of Section 3, Listing 1.6 shows a WCC implementation using the Scatter-Gather abstraction. While the Scatter function sends the updated vertex value to all neighbors, the Gather function searches for the smallest value among all messages and updates the vertex value if necessary.

Figure 8a illustrates the implementation of the Scatter-Gather abstraction using Delta Iteration. Here, the initial workset and solution set is the vertex dataset. In the step function, Scatter and Gather functions are applied using Flinks *coGroup* transformation.[11] First, an adjacency list is built by grouping each vertex with all of its incident edges. For each row in that adjacency list, Gelly applies the Scatter function to create new messages. That messages are again grouped with the vertex values (solution set) and fed into the Gather function. The output of that transformation is a dataset containing all vertices

[11] The `coGroup` transformation groups each input dataset on one or more fields and then joins the groups.

```
class WCCMessenger extends MessagingFunction {
  void sendMessages(Vertex<K, VV> v) {
    sendMessageToAllNeighbors(vertex.getValue())
  }
}
class WCCUpdater extends VertexUpdateFunction {
  void updateVertex(Vertex<K, VV> v, Iterator<VV> messages) {
    VV current = v.getValue()
    VV min = current
    for (Message message in messages)
      if (message < min) min = m
    if (current != min) v.setValue(min)
  }
}
```

Listing 1.6: Scatter/Messaging and Gather/Update functions for WCC. `MessagingFunction` and `VertexUpdateFunction` are provided by Gelly and need to be extended by the user.

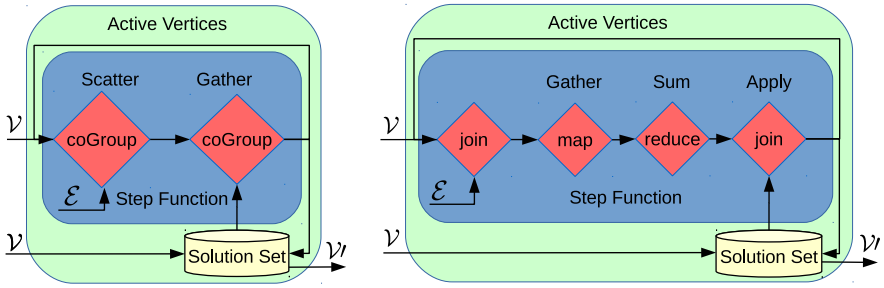
that changed their value. This dataset is then used to update the solution set and also as workset for the next iteration.

In contrast to Scatter-Gather, where information is pushed to a vertex, in the Gather-Sum-Apply (GSA) abstraction, each vertex instead pulls information from its neighbors.[12] One iteration is divided into three phases: In the Gather phase, a user-defined function is applied on the neighborhood of each vertex. Here, each pair of incident edge value and corresponding adjacent vertex value produces a partial value. In the Sum phase, a second user-defined function aggregates the partial values for each neighborhood to a single value. In the final Apply phase, the aggregated value and the current vertex value are used to produce a new vertex value. For a WCC computation, the user-defined functions are presented in Listing 1.7. In the Gather function, we select the value stored at each adjacent vertex.[13] After that, we compute the minimum among those values by reducing them pair-wise in the Sum function. In the Apply function, we finally update the vertex value if the reduced value is smaller than the current vertex value.

Figure 8b illustrates the GSA abstraction implemented using Delta Iteration. In contrast to the Scatter-Gather implementation, vertices are first joined with their incident edges to construct neighbors as input for the Gather function. The latter is applied using a *map* transformation and returns a value for each neighbor. Those values are reduced for each neighborhood by applying the Sum function and finally joined with the vertices to update their values using the Ap-

[12] GSA is a variant of the GAS abstraction introduced by PowerGraph [51] and discussed in Section 3.

[13] The `Neighbor` class allows access to the incident edge value and the adjacent vertex value.



(a) Scatter-Gather (b) Gather-Sum-Apply

Fig. 8. Scatter-Gather and Gather-Sum-Apply abstraction using Delta Iteration [8]. Input for both iterations are the vertex dataset $\mathcal{V}$ (initial working and solution set), the edge dataset $\mathcal{E}$ and the respective user-defined functions. In both cases, the output dataset $\mathcal{V}'$ contains the updated vertex values.

ply function. As with Scatter-Gather, the result is a dataset of updated vertices which is used to evolve the solution set and as workset for the next iteration.

As denoted in Section 3, the main difference between Scatter-Gather and GSA computations is that in the Gather phase of GSA, the computation is parallelized over the edges, while in the Scatter phase, it is parallelized over the vertices. Through this, GSA is advantageous if the Gather phase contains expensive computation or if the graph shows a skewed degree distribution. Also, since the Sum phase of a GSA computation exploits a *reduce* transformation, the results computed on a single worker are internally combined before they are sent to other workers which decreases network traffic and computation times [8]. However, in contrast to Scatter-Gather, the GSA composition prohibits the communication between vertices that are not adjacent in the graph.

4.3 Comparison to other graph dataflow frameworks

Another prominent implementation of a graph dataflow framework is GraphX [52, 116, 115] which is integrated into Apache Spark [118]. GraphX provides a similar API for graph transformation and neighborhood methods that can be composed with other Spark libraries. For iterative graph processing, GraphX implements the Gather-Apply-Scatter abstraction introduced by Powergraph [51] and discussed in Section 3. Like Gelly, GraphX is built on top of the underlying batch API and uses two distributed collections, so-called Resilient Distributed Datasets (RDD), to manage vertices and edges. RDDs are similar to the concept of a dataset in Flink and support transformations (e.g., map, reduce, join) which result in new RDDs. However, in contrast to Gelly, GraphX offers various optimizations tailored for graph analytics. One important optimizations is the partitioning of edges based on vertex-cut algorithms like 2D hash partitioning. Here the edge collection is equally distributed across all workers by minimizing the number of times each vertex is cut. A second optimization is the reduction

```
class GatherNeighborValues extends GatherFunction {
  VV gather(Neighbor n) {
      return n.getVertexValue()
  }
}
class GetMiniumValue extends SumFunction {
  VV sum(VV newValue, VV currentValue) {
    return (newValue < currentValue) ? newValue : currentValue
  }
}
class UpdateComponent extends ApplyFunction {
  void apply(VV sumValue, VV originalValue) {
    if (sumValue < originalValue) setResult(sumValue)
  }
}
```

Listing 1.7: Gather, Sum and Apply functions for WCC in Gelly.

of network traffic between workers by introducing so called mirror vertices in combination with multicast joins [52]. Here, a join operation between vertex and edge RDD transfers only those vertices to edge partitions that are incident to the contained edges.

5 Gradoop

The distributed graph processing and graph dataflow approaches presented in the preceding sections are well suited for scalable graph analytics, especially to execute iterative graph algorithms on large graphs. The graph dataflow approaches also support a flexible combination of graph processing with general data transformation operators provided by the underlying frameworks. However, the implemented graph data models are largely generic and do not meet the requirements posed in the introduction, in particular schema-flexible support for semantic graph data with vertices and edges of different types and varying attributes. Without this support, graph operators such as evaluations on vertex or edge attributes need to be user-defined making the analysis of heterogeneous real-world data a laborious programming task. Moreover, none of the graph systems discussed so far has built-in support to manage collections of graphs, e.g., application-specific subgraphs such as communities in social networks. Finally, the graph data model should provide a set of declarative operators on graphs and graph collections that can be used for the simplified development of advanced graph analysis programs.

The Gradoop framework (*Gra*ph data management and analytics with Ha*doop*) [64, 65] aims at meeting these requirements and improving current graph dataflow systems. It is built on the so-called **E**xtended **P**roperty **G**raph **M**odel [65] supporting semantically rich, schema-free graph data within many

distinct graphs. A set of high-level operators is provided for analyzing both single graphs and collections of graphs. These operators fulfill the closure property[14] as they take single graphs or graph collections as input and result in single graphs or graph collections thus enabling their composition to complex analytical programs. Gradoop is GPLv3-licensed and publicly available.[15] In the following subsections, we will first introduce the architecture of Gradoop and then focus on the data model including its operators. Finally, we illustrate the capabilities of Gradoop with an exemplary analytical dataflow program.

5.1 Architecture

Gradoop aims at providing a framework for scalable graph data management and analytics on large, semantically expressive graphs. To achieve horizontal scalability of storage and processing capacity, Gradoop runs on shared nothing clusters and utilizes existing Hadoop-based software for distributed data storage and processing.

Figure 9 shows the high-level architecture of Gradoop. Analysts declare graph analytical programs using a domain specific language, called Graph Analytical Language (GrALa). The language contains analytical operators for single graphs and graph collections as well as general operators to read and write graph data from and to data stores. GrALa has been developed on top of the Extended Property Graph Model (EPGM) that will be discussed in the next section.

To execute analytical programs in a distributed environment, the EPGM and GrALa are implemented on top of Apache Flink. This way, Gradoop provides new features for graph analytics while benefitting from existing Flink capabilities for large-scale data and graph processing. Flink handles program optimization as well as data distribution and parallel execution across a cluster of machines. Furthermore, Gradoop can be easily integrated with other Flink libraries, like Gelly or Machine Learning.

The distributed graph store offers the possibility to manage a persistent graph database structured according to the EPGM and is implemented in Apache HBase[16], a distributed, non-relational database running on the Apache HDFS (Hadoop Distributed File System). The graph store offers basic methods to read and write a database and therefore serves as data source and sink for graph analytical programs. Additionally, Gradoop allows reading from and writing to any data store which is supported by Apache Flink (e.g., HDFS files, relational databases, NoSQL databases).

5.2 Extended Property Graph Model

The EPGM extends the popular property graph model [97] (Section 2.2) by supporting graph collections and composable analytical operators. Graph collections

[14] An operator fulfills the closure property if the execution of that operator on members of an input domain results in members of the same domain.

[15] `http://www.gradoop.com`

[16] `http://hbase.apache.org`

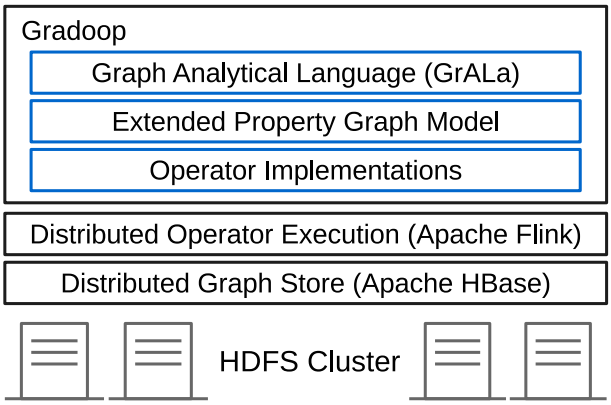


Fig. 9. Gradoop High-Level Architecture.

are a natural way to represent logical partitions of a graph, e.g., communities in a social network [47] or business process executions [88]. Further on, graph collections are the result of certain graph algorithms, e.g., embeddings found by graph pattern matching [48] or frequent subgraph mining [63]. Using GrALa, the EPGM operators for graphs and graph collections can be used together within analytical programs. In the following, we present the EPGM graph representation and operators in more detail.

Graph Representation 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 an *extended* property graph, a database consists of multiple property graphs which are called *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.

Figure 10 shows an example EPGM database DB of a simple social network. Formally, DB consists of the vertex set $\mathcal{V} = \{v_1, .., v_6\}$ and the edge set $\mathcal{E} = \{e_0, .., e_6\}$ where each element has a unique identifier (e.g., `[1]`). Vertices represent users, groups and interest tags, denoted by corresponding type labels (e.g., `User`) and are further described by their properties (e.g., `name:Alice`). Edges describe the relationships between vertices and also have type labels (e.g., `memberOf`) and properties. Type labels do not determine a schema, as elements with the same type label may have different property keys, e.g., v_1 and v_4.

The sample database contains the set of logical graphs $\mathcal{L} = \{G_1, G_2\}$, where each graph represents a community inside the social network, in particular specific interest groups (e.g., `Graphs`). Each logical graph has a dedicated subset of vertices and edges, for example, $V(G_1) = \{v_1, v_2, v_4\}$ and $E(G_0) = \{e_1, e_2\}$.

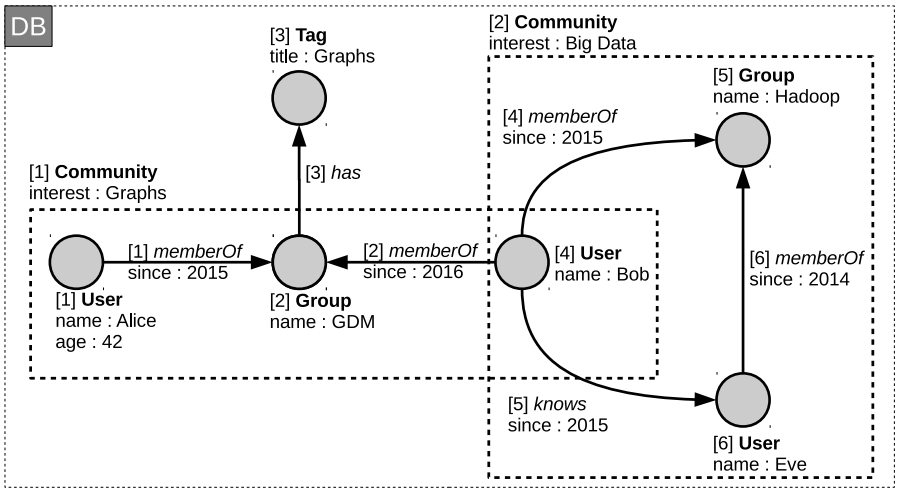


Fig. 10. Example EPGM database representing a simple social network containing two logical graphs. Each logical graph describes a community inside the social network, for example, people that are member of a group related to graphs form the *Graphs* community.

One can see that vertex (and also edge sets) of logical graphs may overlap since $V(G_1) \cap V(G_2) = \{v_4\}$. Note that also logical graphs have type labels (e.g., `Community`) and may have properties to annotate the graph with specific metrics or descriptive information (e.g., `interest:Big Data`). 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.

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 $\mathcal{G} \in \mathcal{L}^n$ is a n-tuple of logical graphs and thus 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 5 lists the analytical operators together with their corresponding pseudocode syntax for calling them in GrALa. 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, GrALa distinguishes between *graph operators* and *collection operators* as well as *unary* and *binary operators* (single graph/collection vs. two graphs/collections as input). There are also *auxiliary operators* to apply graph operators on collections or to call specific graph algorithms. In addition to the listed ones GrALa provides operators to create graphs, vertices and edges including respective labels and properties. In the following, we will present a subset of available

	Graph Analytical Language		
	Operator	Operator Signature	Output
Unary	Aggregation	`Graph.`**`aggregate`**`(propertyKey, aggregateFunction)`	`Graph`
	Transformation	`Graph.`**`transform`**`(graphFunction, vertexFunction, edgeFunction)`	`Graph`
	Pattern Matching	`Graph.`**`match`**`(patternGraph)`	`Collection`
	Subgraph	`Graph.`**`subgraph`**`(vertexPredicateFunction, edgePredicateFunction)`	`Graph`
	Grouping	`Graph.`**`groupBy`**`(vertexGroupingKeys, vertexAggregateFunction, edgeGroupingKeys, edgeAggregateFunction)`	`Graph`
	Selection	`Collection.`**`select`**`(predicateFunction)`	`Collection`
	Distinct	`Collection.`**`distinct`**`()`	`Collection`
	Limit	`Collection.`**`limit`**`(n)`	`Collection`
	Sorting	`Collection.`**`sortBy`**`(propertyKey, [:asc|:desc])`	`Collection`
Binary	Equality	`Graph.`**`equals`**`(otherGraph, [:identity|:data])`	`Boolean`
	Combination	`Graph.`**`combine`**`(otherGraph)`	`Graph`
	Exclusion	`Graph.`**`exclude`**`(otherGraph)`	`Graph`
	Overlap	`Graph.`**`overlap`**`(otherGraph)`	`Graph`
	Equality	`Collection.`**`equals`**`(otherCollection, [:identity|:data])`	`Boolean`
	Difference	`Collection.`**`difference`**`(otherCollection)`	`Collection`
	Intersect	`Collection.`**`intersect`**`(otherCollection)`	`Collection`
	Union	`Collection.`**`union`**`(otherCollection)`	`Collection`
Aux.	Apply	`Collection.`**`apply`**`(unaryGraphOperator)`	`Graph`
	Reduce	`Collection.`**`reduce`**`(binaryGraphOperator)`	`Graph`
	Call	`[Graph|Collection].`**`callForGraph`**`(algorithm, parameters)`	`Graph`
		`[Graph|Collection].`**`callForCollection`**`(algorithm, parameters)`	`Collection`

Table 5. Overview of operators provided by the domain specific language GrALa.

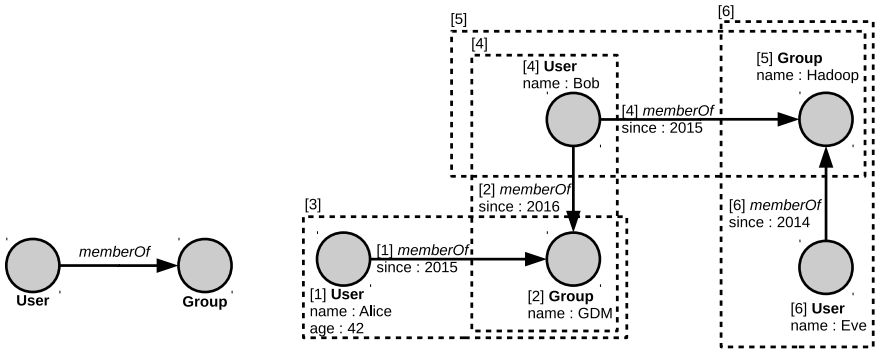
operators, a detailed discussion of all operators and their implementation can be found in [64].

Aggregation An operator often used in analytical applications is aggregation, where a set of values is mapped to a single value of significant meaning. The EPGM supports aggregation at the graph level. Formally, the operator maps an input graph G to an output graph G' and applies the user-defined aggregate function $\alpha : \mathcal{L} \rightarrow A$. Thus, the resulting graph is a modified version of the input graph with an additional property k. In the following, we show a simple vertex count example:

```
alpha = (g => g.V.count())
outGraph = inGraph.aggregate('vertexCount', alpha)
```

Here, a user-defined aggregate function `alpha` computes the cardinality of the vertex set `g.V` of an input graph `g`. The aggregation operator is called on the logical graph referred to by the variable `inGraph`. The operator takes property key `vertexCount` and aggregate function `alpha` as arguments. The resulting logical graph is assigned to the variable `outGraph` and provides a property `vertexCount` storing the result of `alpha`. Basic aggregate functions such as *count*, *sum*, *min* and *max* are predefined in GrALa and can be applied to vertex and edge collections.

Pattern Matching A fundamental operation of graph analytics is the retrieval of subgraphs isomorphic to a user-defined pattern graph [48]. The operator results



(a) Pattern Graph (b) Result Graph Collection

Fig. 11. Example of a pattern matching execution where Figure (a) represents the pattern which is applied on the graph of Figure 10 and Figure (b) shows the resulting graph collection containing all subgraphs that match the pattern.

in a graph collection containing all embeddings of that pattern graph in the input graph. For example, in Figure 11a, a simple pattern graph describes the membership relation between an arbitrary user and an arbitrary group. Applied on our example graph in Figure 10, the operator returns the collection shown in Figure 11b. Each logical graph in that collection represents an embedding of the pattern graph. To support such queries, GrALa provides the pattern matching operator, where a pattern graph G^* and a predicate $\varphi : \mathcal{L} \rightarrow \{true, false\}$ are the operator arguments. Pattern matching is applied to a graph G and returns a graph collection $\mathcal{G}' = \{G' \subseteq G \mid G' \simeq G^* \wedge \varphi(G') = true\}$ containing all matches, for example:

```
embeddings = db.G.match("(a:User)-[e:memberOf]-(b:Group))")
```

The shown pattern graph reflects our membership query. GrALa adopts the basic concept of describing graph patterns using ASCII characters from Neo4j Cypher [7], where `(a)-[e]->(b)` denotes an edge `e` from vertex `a` to vertex `b`. The predicate function φ is embedded into the pattern by defining type labels and properties. In the example, we describe a pattern of two vertices and one edge, which are assigned to variables (`a,b` for vertices; `e` for the edge). Variables are optionally followed by a label (e.g., `a:User`) and properties (e.g., {`name = 'Alice'`}). The operator is called for the logical graph representing the whole database DB (`db.G`) of Figure 10 and returns a collection assigned to variable `embeddings` and containing four new logical graphs.

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 $V(G)$. Vertices are grouped based

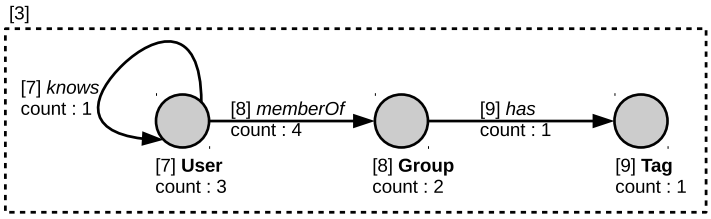


Fig. 12. Example of the grouping operator applied on the graph of Figure 10. The graph is grouped by vertex and edge label and a count aggregate is used to compute the number of elements represented by each resulting super vertex/edge.

on selected property values (including their type label) while edges are grouped along their incident vertices and optionally by selected property values. Vertices and edges in the grouped graph are called *super vertices* and *super edges*, respectively. Additionally, the vertex and edge aggregate functions can be used to compute aggregated property values for super vertices and edges, e.g., the average age of users 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 the grouping operator:

```
1  outGraph = db.G.groupBy(
2    [:label],
3    (suVertex, vertices => suVertex['count'] = vertices.count()),
4    [:label],
5    (suEdge, edges => suEdge['count'] = edges.count()))
```

The goal of this example is to group vertices and edges in the graph of Figure 10 by their corresponding type label. Furthermore, we want to count the number of vertices and edges represented by each label. In line 2, we define the vertex grouping keys. Here, we want to group vertices by their type label. However, it is also possible to define property keys which are used to select property values for grouping (e.g., to group users by their age). Edges are also grouped by type label (line 4). In lines 3 and 5, we define the vertex and edge aggregate functions. Both receive the super entity (i.e., `suVertex`, `suEdge`) 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 12 shows the resulting logical graph of the grouping example.

Analytical Example Finally, we illustratete the capabilities of Gradoop using an exemplary analytical program based on social network data. We assume a heterogeneous network including various vertex and edge types including users and their mutual friendship relations similar to Figure 10. Vertices and edges have properties, for example, user vertices store the corresponding name, gender and the city the user lives in.

```
1  Graph socialNetwork = EPGMDatabase.fromHBase().getDBGraph()
2  Graph result = socialNetwork
3   .subgraph(
4     (vertex => vertex[:label] == 'User'),
5     (edge => edge[:label] == 'knows'))
6   .transformVertices(currentVertex, transformedVertex => {
7       transformedVertex['city']   = currentVertex['city']
8       transformedVertex['gender'] = currentVertex['gender'])
9   .callForCollection(:LabelPropagation, [:id, 5]))
10  .apply(g => g.aggregate('vertexCount', (h => h.V.count())))
11  .select(g => g['vertexCount'] > 50_000)
12  .reduce(g, h => g.combine(h))
13  .groupBy(['city','gender'],
14    (suVertex, vertices => suVertex['count'] = vertices.count()),
15    [], (suEdge, edges => suEdge['count'] = edges.count()))
16  .aggregate('vertexCount', (g => g.V.count()))
17  .aggregate('edgeCount', (g => g.E.count()))
18 result.writeAsJSON('hdfs:///output/')
```

Listing 1.8: Analytical program which shows the combination of EPGM operators.

The graph analytical program used for our example is shown in Listing 1.8 and includes several operators from Table 5 not discussed before. The input is an entire social network represented as a single logical graph. Here, the graph is stored in HBase and distributed across a cluster of machines. In line 1, we load the graph and refer to it using the variable `socialNetwork`. Starting from Line 2, we define our analytical program as a composition of GrALa operators. First, we extract the *subgraph* containing only users and their mutual friendship relationships by applying user-defined vertex and edge predicate functions. The vertices of the resulting graph are then *transformed* to a representation which is limited to information necessary for subsequent operators. The user-defined transformation function takes the current vertex and a copy of that vertex with omitted label and properties as input and determines, which data gets transferred from the current to the copied vertex. Here, we adopt only the `gender` and `city` properties. The transformed subgraph is then used as input for the *call* operator in line 9. That operator allows us to call specific graph algorithms on logical graphs (e.g., pagerank) or graph collections (e.g., common subgraph detection). Here, we use Label Propagation [92], a community detection algorithm that is already implemented in Flink Gelly. The algorithm propagates the value associated with a given property key (we use the vertex id) through the graph in five iterations. The result is a graph collection containing all found communities. In line 10, we *apply* the *aggregate* operator on each of these communities to compute their respective vertex counts. Then, we use the *selection* operator to filter communities that have more than 50K users. The filtered graphs are then combined to a single logical graph by applying the *reduce* operator on the

filtered collection. The result is a single logical graph containing all vertices and edges from all graphs in the collection. We further *group* this 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 16 and 17, we use *aggregation* to compute how many super entities are contained in the resulting logical graph. As Gradoop is build on top of Apache Flink, program execution needs to be triggered explicitly. In the last line, we start the program by writing the resulting logical graph to HDFS using a dedicated JSON output format.

The example illustrates that Gradoop allows the combined application of graph queries and transformations as well as the execution of graph algorithms such as for community detection within a compact dataflow program. The entire program can be automatically executed in parallel on distributed clusters since all operators are implemented using Flink operators.

6 Comparison

In our introduction we stated various requirements for flexible and efficient management and analysis of big graph data. In the previous sections, we discussed three system categories in detail: graph database, graph processing and graph dataflow systems. We now want to compare these categories based on the stated requirements. Table 6 highlights the features of the respective categories.

- *Powerful graph data model:* The need to process graphs with heterogeneous vertices and edges of varying types and with different attributes is currently addressed best by graph database systems that offer schema-free, flexible data models like the PGM or RDF. From the considered distributed frameworks for graph analytics, only Gradoop supports such a graph data model. Its EPGM is the only data model with versatile support for graph collections.
- *Powerful query and analysis capabilities:* We saw that each system category has its own approach for querying and analyzing the graph. While declarative languages, like Cypher or SPARQL, are unique for graph database systems, graph processing systems provide vertex- and graph-centric programming abstractions that simplify the implementation of distributed graph algorithms. In contrast, graph dataflow systems combine vertex-centric computation with additional libraries and general-purpose data operators for pre- and post-processing. While Gelly and GraphX provide transformation and aggregation methods for single graphs, Gradoop in addition offers operators that exploit the expressiveness of the underlying graph data model.
- *High performance and scalability:* The main focus of graph database systems are OLTP applications with demand for very low query execution times. To achieve that, those systems focus on query optimization, indexing, efficient physical storage and data replication. Graph processing and dataflow systems on the other hand, focus on analytical programs involving graphs that

	Graph Database Systems	Graph Processing Systems	Graph Dataflow Systems	
Examples	Neo4j, Marklogic	Pregel, Giraph	Gelly, GraphX	Gradoop
Data Model	PGM/RDF	Generic Graph	Generic Graph, Datasets	EPGM, Datasets
Graph Collections	No	No	No	Yes
Query approach	Query Languages	Vertex-/Graph-centric Computation models	Vertex-centric Computation, Dataflow Programs	
Scope	OLTP/Queries	Analytics	Analytics	Analytics
Scalability	Up/(Out)	Out	Out	Out
Persistency	Yes	No	No	Yes
Transactions	Yes	No	No	No
Graph Visualization	Interactive Traversal	No	No	No

Table 6. Feature comparison of different approaches to graph data management and analytics.

span an entire cluster of machines. Here, the focus is on balanced load distribution, reducing network traffic and fault-tolerance in case of system-failures during long running programs. While the architecture of graph processing as well as dataflow systems is built with data distribution in mind, only a subset of available graph databases provides that feature. The actual system performance depends on many implementation decisions as well as on the data and workload characteristics - an extensive benchmarking could help clarify differences between the different approaches (see Section 7.2).

- *Persistent graph storage and transaction support:* As stated before, graph databases focus on OLTP applications, hereby offering support for ACID compliant transactions on persistent data. Graph processing and dataflow systems solely focus on reading the graph from data sources, process it in a distributed manner and write the results back to an arbitrary data sink. Gradoop offers rudimentary support for managing graphs in a persistent database. Those graphs can be either used in further analytical programs or be queried directly in the graph store. However, some graph databases, for example Titan [17], already provide APIs to execute ACID compliant graph processing algorithms.
- *Ease of use / graph visualization:* The growing interest in graph-based data systems indicates that the use of graphs is intuitive for many use cases. However, if it comes to meaningful visualization of graphs, there is only

limited support in some graph database systems for navigating through the graph. Hence, support for versatile visualization and interactive exploration for large graphs or the results of graph analytics is still missing and a topic for future research and development (see Section 7.5).

7 Current research and open challenges

The development of systems for graph analytics has made great progress in the past decade but there are still several areas requiring significant further improvement and research. In the following, we discuss some of these areas together with a brief outline of initial results that have already been achieved.

7.1 Graph data allocation and partitioning

The efficiency of distributed graph processing substantially depends on a suitable data allocation (partitioning) of the graph data among all nodes of the processing system. This data allocation should enable graph processing with a minimum of inter-node communication and data transfer while at the same time ensure a good load balancing such that all nodes can be effectively utilized. The associated optimization objective is to find a balanced distributed of vertices and their edges such that the each partition includes about the same number of vertices while the sum of edges crossing partitions is minimized ("vertex cut"). The graph partitioning problem is known to be NP-hard and has attracted a large amount of research, in particular in graph theory [32]. The most promising approximate solutions are multilevel approaches such as METIS [67] that include steps for coarsening graphs to find partitions for condensed graphs and uncoarsening to the detailed graph while keeping the partitioning from the coarse graph [32]. Although these approaches can be run in parallel they are still expensive and thus likely of limited scalability to very large graphs [81]. A further problem is that even a near-perfect static data allocation is not sufficient since during the execution of long-running analysis, e.g., in a Pregel-like system, the processing of some partitions may have already terminated while others still have to be processed. Furthermore, some graphs may quickly change, e.g., in social networks, so that the data allocations needs to be quickly adapted without causing a completely new static data allocation [60].

Current distributed graph data systems mostly follow a simple hash-based partitioning of the vertices across all nodes. This approach achieves an even distribution of vertices and thus good load balancing and does not require a data structure to locate vertices. On the other hand, it frequently assigns neighboring vertices to different partitions leading to poor locality of processing and high communication overhead for many algorithms. A number of proposals has been made to address these limitations and also support adaptive data allocation to deal with changing graphs or load imbalances during analytical processing. Stanton and Kliot [104] propose a locality-aware data allocation that incrementally assigns vertices to the partition where most of the already assigned neighbors

have been placed without causing significant load imbalances. This approach is also suitable for changing graphs to allocate new vertices. PowerGraph [51] as well as Spark GraphX [52] support an *edge-based partitioning* rather than a vertex-driven approach so that the number of edges is balanced across partitions and vertices are replicated along with their edges ("vertex cut"). Such an approach is especially valuable for graphs with a highly uneven distribution of vertex degrees such that a large fraction of the edges is associated with few vertices that could easily lead to load imbalances with a vertex-based data partitioning. The replication of vertices has also been proposed for an adaptive data allocation for dynamically evolving graphs [84, 60]. Furthermore, graph repartitioning approaches based on the dynamic migration and replication of vertices have been proposed to deal with load imbalances during analytical processing in graph processing systems such as GPS and XPregel, but the associated overhead has not always resulted in significantly improved execution time [81].

The discussion shows that graph data allocation is a challenging problem that is not sufficiently solved with a single solution such as hash partitioning. This is because a good data allocation depends on the characteristics of the graph data as well on the intended kinds of graph analytics. It would be desirable to have support of a spectrum of allocation approaches from which the system can automatically choose depending on the graph and workload characteristics (similar as for data allocation in parallel database systems [95, 86]). Furthermore, it would be desirable to find a data allocation that can deal with mixed workloads including different kinds of complex analytical tasks as well as diverse kinds of interactive queries.

7.2 Benchmarking and evaluation of graph data systems

The large number of existing systems for analyzing graph data poses the question for potential users of such systems which of the systems performs best for which kind of analysis tasks, datasets and platforms. This asks for a comprehensive and comparative performance evaluation or benchmarking of the different implementations under comparable conditions. Such evaluations are also expected to help identify existing bottlenecks that may be addressed in the further development of systems. A large number of studies has addressed these issues by comparing the performance of selected systems, in particular for graph database and graph processing systems, e.g., [42, 53, 56, 77, 80, 102, 121]. While these studies have been insightful, their results are mostly not comparable as each study has chosen a different set of systems, different sets of real or synthetically generated graph datasets, different sets of queries and analytical tasks (ranking from pagerank to collaborative filtering and graph coloring) as well as different environments in terms of number of worker nodes and their characteristics such as memory size and number of cores. The different studies thus have only few observations in common such that graph processing systems significantly outperform MapReduce-based implementations [42, 53] and that Giraph is mostly slower than other graph processing systems such as GraphLab or GPS [53, 56, 77, 102, 121].

A few studies also considered Apache Spark [121] and Stratosphere (the predecessor of Flink) [42, 53], but comparative performance evaluations for GraphX and Gelly are still missing. Here, it is of great interest, how the underlying dataflow systems behave for graph analytical workflows involving multiple graph and non-graph transformations as well as iterative algorithms. For example, Flink optimizes dataflow programs with cost-based query-optimization techniques that are similar to the ones used in database systems, e.g., to reorder transformations. As the optimizer assumes independent datasets, such generic program optimizations may, however, cause problems for the processing of graph data with strongly interrelated vertex and edge datasets. It would thus be interesting to evaluate the performance of different kinds of graph dataflow programs in more detail and develop optimization techniques customizable for graph processing and analysis, e.g., to automatically repartition the graph during program execution.

In addition to the individual performance studies there are also several recent attempts for the definition of graph analysis benchmarks, namely Linkbench [25], LDBC [44, 24] and gMark [26]. These benchmarks specify the synthetic generation of datasets of different sizes, the workloads and performance metrics together with rules on how to perform the evaluation in order to achieve comparable results between different systems. Furthermore, there are proposals for the synthetic generation of graph datasets, e.g., for business intelligence [89]. From the mentioned benchmarks, the LDBC (Linked Data Benchmark Council) effort is the most ambitious as it consists of two benchmarks for semantic publishing and social network benchmarking (SNB) and different sets of query and analysis workloads exhibiting several "choke points" to stress-test the systems. Unfortunately, there are only few evaluations so far for all benchmarks so that they could not yet demonstrate their usefulness.

7.3 Analysis of dynamic graphs

Previous approaches for graph analytics focus on static graphs that remain stable. Most graphs, e.g., social networks, however are constantly changing so that the results of analytical processes, e.g., for community detection or on metrics such as pagerank or centrality, need to be updated or refreshed. Furthermore, there is a need for fast, one-pass graph analysis in data streams, e.g., to quickly identify new topics and correlations in Twitter data, to determine online recommendation for users based on their current website usage (clickstream) or to identify potentially criminal acts such as credit card misuse or planned terror attacks.

According to [19], dynamic graphs fall into two categories: slowly evolving graphs (e.g., co-authorship networks) and streaming networks. In the first case, it is possible to maintain different snapshots of the graph as the basis for an offline analysis while in the second case a near real-time analysis is necessary. The analysis can further focus on understanding the evolution, e.g., by comparing different snapshots, or on refreshing previous analysis results for the new graph data. A large amount of research has already dealt with these topics as

surveyed in [19]. Typical observations show that the number of edges grows stronger than the number of vertices leading to increasingly denser networks (reduced distances between vertices). Many studies focused on analyzing the evolution of communities, e.g., by applying a clustering-based community detection on different snapshots and analyzing the cluster changes. Graph analysis for streamed data has also found interest already, e.g., to detect outliers such as a new co-author link between authors of different communities (linkage anomaly). There is also some work to incrementally update complex graph metrics such as betweenness centrality[17] for streamed data, e.g., using approximation techniques and specific index structures [59]. The Kineograph system [37] supports the dynamic graph-based analysis of Twitter data (correlations between users and hashtags) by continuously creating new in-memory graph snapshots that can then be evaluated by conventional mining approaches for static graphs, e.g., for ranking or community detection.

Despite the relatively large body of previous theoretical and experimental work on dynamic networks, little work has been done for big graph data utilizing current distributed graph data platforms as discussed in this chapter. Analyzing massive amounts of changing graph data in a distributed way poses many new algorithmic and data management challenges including the need for adaptive data allocation (as discussed in Section 7.1 above). Data management and graph analytics is challenging on a sequence of large graph snapshots as well as for streaming data and needs much further research. Most studies for graph evolution and dynamic graph analytics focused on structural changes such as the addition of new vertices and edges; more work is needed for considering both changes in structure and content, e.g., new publication topics or changing interests of users in social networks. Furthermore, the graph changes may have to be associated with information in different data sources, e.g., to better understand certain changes or identify potential criminal acts. The latter aspects might imply the need to develop application-specific approaches to take the specific kinds of changes and additional information to correlate with into account.

7.4 Graph-based data integration and knowledge graphs

Before graph data can be analyzed it is necessary to construct and store the graphs for further processing. As for big data analysis in general, the graph data typically needs to be extracted from the original data sources (e.g., from social networks, web pages, tweets, relational databases, etc.), transformed and cleaned. Furthermore, it is often necessary to combine and interrelate data from multiple sources into the combined graph. These steps are typically carried out within so-called ETL (extract-transform-load) workflows that may be performed in parallel on Hadoop platforms, e.g., using MapReduce or other frameworks such as Hive, Spark or Flink [28, 54, 69]. A particularly important and expensive step

[17] The betweenness centrality of a vertex is defined as the number of shortest paths in a network pathing through the vertex. A high value thus indicates that a vertex is centrally located so that it plays an important role in a network.

is the matching of equivalent entities (users, products, etc.) from different sources so that they can be fused together, e.g., within one graph vertex. Map-reduce-based tools such as Dedoop [72] have been developed for scalable entity matching. So far, relatively little work has focused on ETL for graph data, although there are new challenges in all steps of a typical ETL pipeline, e.g., to extract graphs from certain data sources such as relational databases, for data cleaning and for data integration. GraphBuilder is one of the few tools for graph ETL [62]. It utilizes MapReduce jobs to extract data from sources based on user-defined parsers and to generate vertices and edges. It also provides different options for distributed storage of the resulting graph data. The BIIIG system supports the extraction of graph data from several relational databases to support a graph-based business intelligence [88, 90].

A particularly challenging kind of graph-based data integration becomes necessary for the generation and continuous maintenance of so-called *knowledge graphs* [41, 87, 94] providing a large amount of interrelated information about many real-world entities (persons, locations, ...) and their describing metadata concepts, typically extracted and combined from several other sources. Non-commercial knowledge graph projects include YAGO[18], DBpedia[19], Freebase and its successor Wikidata[20]. Companies such as Yahoo! [28], Google, Microsoft or Facebook utilize even larger knowledge graphs [87] combining information from more resources including web pages and search queries. Most of the systems make use of the RDF data model to express the contained knowledge.

A massive problem is the typically low data quality, high diversity and large volume of the automatically extracted information to be integrated into knowledge graphs. Dealing with these issues requires scalable and largely automatic (learning-based) approaches for information extraction, cleaning, classification and matching [28, 41, 94].

Low data quality including incomplete and contradicting information from the information to be integrated into a knowledge graphs is a huge challenge to deal with requiring scalable and largely automatic (learning-based) approaches for information extraction, cleaning, classification and matching [28, 41, 94].

7.5 Interactive graph analytics

Interactive graph analytics supported by suitable visualizations is highly desirable to put the human in the loop for exploring and analyzing graph data. However, interactive graph analysis is currently only supported for query processing with graph databases (Section 2) while graph analytics with the discussed distributed frameworks is largely batch-oriented. For example, Neo4j allows such an interactive and visual exploration of the immediate neighborhood of selected vertices[21]. Screen size and human recognition capabilities limit this approach to

[18] www.mpi-inf.mpg.de/yago-naga/yago/
[19] http://dbpedia.org/
[20] www.wikidata.org
[21] http://neo4j.com/graph-visualization-neo4j/

inspecting only tens to a few hundreds of vertices at a time. More promising is the exploration and visualization of summarizing graph data, similar to multidimensional OLAP queries for data warehouses. Several approaches for such graph summaries [108, 119], graph OLAP [36, 50, 113, 120] and grouping (Section 5.2) have already been proposed and can potentially be applied for large graphs. For example, k-SNAP [108] automatically creates summarized graphs with k vertices, where the change of parameter k enables an OLAP-like *roll-up* and *drill-down* within a dimension hierarchy [36]. However, the approach is not yet fully interactive as it depends on a pre-determined parameter.

To improve ease-of-use there is a strong need for extending interactive and visual analysis to more kinds of graph analysis, from OLAP-style aggregations for single large graphs and graph collections to exploring evolution in dynamic graphs. Furthermore, it should be possible to interactively evaluate the results of expensive graph analytics, e.g., to inspect parts of the graphs with a high centrality, certain communities of interest, etc. The currently existing separation between interactive query processing with graph databases and batch-oriented graph analytics should thus be overcome by providing all kinds of analysis in a unified, distributed platform with support for interactive and visual analysis. Some of the graph databases of Section 2, e.g., Blazegraph, System G and Titan, try to go into this direction, but there are still many open issues in finding suitable visualizations and interaction forms for the different kinds of analysis. Furthermore, the combined processing of mixed workloads with queries and heavy-weight graph algorithms should also be possible with the graph processing frameworks for Hadoop-based clusters.

8 Conclusions and outlook

The analysis of graph data has become of great interest in many applications and a major focus of big data platforms. We have posed major requirements for big data graph analytics and surveyed current systems in three categories: graph database systems, distributed graph processing systems and distributed graph dataflow systems. The summarizing comparison of these system categories with respect to the posed requirements in Section 6 showed that there are still big differences between the query-focused graph database systems and the distributed platforms focusing on large-scale iterative graph analysis. While distributed graph analysis platforms generally lack an expressive graph data model, the distributed dataflow approach Gradoop provides an extended property graph model with powerful support for analyzing collections of graphs.

Despite the significant advances made in the last few years, the development and use of distributed graph data systems are still in an early stage. Hence, the posed requirements are not yet fully achieved and there are many opportunities for improvement and future research. As discussed in Section 7, this is especially the case for evaluating and improving the performance and scalability of graph data systems, for graph data partitioning and load balancing, for the analysis of dynamic graph data, for graph-based data integration, and for interactive and visual graph analytics.

References

1. Akka. http://www.akka.io, Accessed: 2016-03-10
2. AllegroGraph. http://franz.com/agraph/allegrograph/, Accessed: 2016-03-10
3. Apache Flink Iteration Operators. https://ci.apache.org/projects/flink/flink-docs-master/apis/batch/index.html#iteration-operators, Accessed: 2016-03-09
4. Apache Giraph. http://www.giraph.apache.org, Accessed: 2016-03-10
5. Apache Jena - TBD. https://jena.apache.org/documentation/tdb/, Accessed: 2016-03-09
6. Big Data Spatial and Graph User's Guide and Reference. http://docs.oracle.com/cd/E69290_01/doc.44/e67958/toc.htm, accessed: 2016-03-16
7. Cypher Query Language. http://neo4j.com/docs/stable/cypher-query-lang.html, Accessed: 2016-03-16
8. Gelly: Flink Graph API. https://ci.apache.org/projects/flink/flink-docs-master/apis/batch/libs/gelly.html, Accessed: 2016-03-15
9. GraphDB: At Last, the Meaningful Database. http://ontotext.com/documents/reports/PW_Ontotext.pdf, Whitepaper July 2014
10. InfiniteGraph: The Distributed Graph Database. http://www.objectivity.com/wp-content/uploads/Objectivity_WP_IG_Distr_Benchmark.pdf, Whitepaper 2012
11. Key Features - ArangoDB. https://www.arangodb.com/key-features/, Accessed: 2016-03-10
12. MarkLogic Semantics. http://www.marklogic.com/resources/marklogic-semantics-datasheet/, Datasheet March 2016
13. Oracle Spatial and Graph: Advanced Data Management. http://www.oracle.com/technetwork/database/options/spatialandgraph/spatial-and-graph-wp-12c-1896143.pdf, Whitepaper September 2014
14. quasar. http://www.paralleluniverse.co/quasar, Accessed: 2016-03-10
15. Stardog 4 - The Manual. http://docs.stardog.com/, Accessed: 2016-03-10
16. The bigdata RDF Database. https://www.blazegraph.com/whitepapers/bigdata_architecture_whitepaper.pdf, Whitepaper May 2013
17. TITAN: Distributed Graph Database. http://thinkaurelius.github.io/titan/, Accessed: 2016-03-10
18. Why OrientDB? http://orientdb.com/why-orientdb/, Accessed: 2016-03-10
19. Aggarwal, C., Subbian, K.: Evolutionary network analysis: A survey. ACM Computing Surveys (CSUR) 47(1), 10 (2014)
20. Agha, G.A.: Actors: A model of concurrent computation in distributed systems. Tech. rep., DTIC Document (1985)
21. Alexandrov A. et al.: The Stratosphere Platform for Big Data Analytics. VLDB Journal 23(6) (2014)
22. Angles, R.: A comparison of current graph database models. In: Proc. ICDEW (2012)
23. Angles, R., Gutierrez, C.: Survey of graph database models. ACM Computing Surveys (CSUR) 40(1) (2008)
24. Angles R. et al.: The linked data benchmark council: a graph and RDF industry benchmarking effort. Proc. SIGMOD 43(1) (2014)
25. Armstrong T. G. et al.: Linkbench: a database benchmark based on the facebook social graph (2013)
26. Bagan G. et al.: gMark: Controlling Diversity in Benchmarking Graph Databases. CoRR abs/1511.08386 (2015)

27. Batarfi O. et al.: Large scale graph processing systems: survey and an experimental evaluation. Cluster Computing 18(3) (2015)
28. Bellare K. et al.: Woo: A scalable and multi-tenant platform for continuous knowledge base synthesis. PVLDB 6(11) (2013)
29. Bertsekas, D.P., Tsitsiklis, J.N.: Parallel and distributed computation: numerical methods, vol. 23 (1989)
30. Bolouri, H.: Modeling genomic regulatory networks with big data. Trends in Genetics 30(5) (2014)
31. Brickley, D., Miller, L.: Foaf vocabulary specification 0.98. Namespace document 9 (2012)
32. Buluç A. et al.: Recent advances in graph partitioning. CoRR (2013)
33. Canim, M., Chang, Y.C.: System G Data Store: Big, Rich Graph Data Analytics in the Cloud. In: IEEE Cloud Engineering (IC2E) (March 2013)
34. Carothers, G.: RDF 1.1 N-Quads: A line-based syntax for RDF datasets. W3C Recommendation (2014)
35. Cattell, R.: Scalable SQL and NoSQL data stores. Proc. SIGMOD 39(4) (2011)
36. Chen C. et al.: Graph OLAP: Towards online analytical processing on graphs. In: IEEE Data Mining (ICDM) (2008)
37. Cheng R. et al.: Kineograph: taking the pulse of a fast-changing and connected world. In: Proc. EuroSys (2012)
38. Das S. et al.: A Tale of Two Graphs: Property Graphs as RDF in Oracle. In: EDBT (2014)
39. Diestel, R.: Graph Theory, 4th Edition, Graduate texts in mathematics, vol. 173 (2012)
40. Ding, Y.: Scientific collaboration and endorsement: Network analysis of coauthorship and citation networks. Journal of informetrics 5(1) (2011)
41. Dong X. et al.: Knowledge Vault: A Web-Scale Approach to Probabilistic Knowledge Fusion. In: Proc. SIGKDD (2014)
42. Elser, B., Montresor, A.: An evaluation study of bigdata frameworks for graph processing. In: IEEE Big Data (2013)
43. Erling, O., Mikhailov, I.: RDF support in the Virtuoso DBMS. In: Networked Knowledge-Networked Media (2009)
44. Erling O. et al.: The ldbc social network benchmark: interactive workload. In: Proc. SIGMOD (2015)
45. Ewen S. et al.: Spinning fast iterative data flows. PVLDB 5(11) (Jul 2012)
46. Ewen S. et al.: Iterative Parallel Data Processing with Stratosphere: An Inside Look. In: Proc. SIGMOD (2013)
47. Fortunato, S.: Community detection in graphs. Physics Reports 486(3-5) (2010)
48. Gallagher, B.: Matching structure and semantics: A survey on graph-based pattern matching. AAAI FS 6 (2006)
49. Gao J. et al.: Glog: A high level graph analysis system using mapreduce. In: Proc. ICDE (2014)
50. Ghrab A. et al.: A Framework for Building OLAP Cubes on Graphs. In: Advances in Databases and Information Systems (2015)
51. Gonzalez J. E. et al.: Powergraph: Distributed graph-parallel computation on natural graphs. In: Proc. OSDI (2012)
52. Gonzalez J. E. et al.: GraphX: Graph Processing in a Distributed Dataflow Framework. In: Proc. OSDI (2014)
53. Guo Y. et al.: How well do graph-processing platforms perform? an empirical performance evaluation and analysis. In: Proc. Parallel and Distributed Processing Symp. (2014)

54. Haas D. et al.: Wisteria: Nurturing scalable data cleaning infrastructure. PVLDB 8(12) (2015)
55. Haerder, T., Reuter, A.: Principles of transaction-oriented database recovery. ACM Computing Surveys 15(4) (1983)
56. Han M. et al.: An experimental comparison of pregel-like graph processing systems. PVLDB 7(12) (2014)
57. Harris, S., Seaborne, A., Prudhommeaux, E.: SPARQL 1.1 query language. W3C Recommendation 21 (2013)
58. Hartig, O., Thompson, B.: Foundations of an alternative approach to reification in RDF. Tech. Rep. arXiv:1406.3399 (2014)
59. Hayashi, T., Akiba, T., Yoshida, Y.: Fully dynamic betweenness centrality maintenance on massive networks. PVLDB 9(2) (2015)
60. Huang, J., Abadi, D.J.: LEOPARD: Lightweight Edge-Oriented Partitioning and Replication for Dynamic Graphs. PVLDB 9(7) (2016)
61. Iordanov, B.: HyperGraphDB: a generalized graph database. In: Web-Age Information Management (2010)
62. Jain, N., Liao, G., Willke, T.L.: Graphbuilder: scalable graph ETL framework. In: Int. Workshop on Graph Data Management Experiences and Systems (2013)
63. Jiang C. et al.: A survey of Frequent Subgraph Mining algorithms. Knowledge Eng. Review 28(1) (2013)
64. Junghanns M. et al.: GRADOOP: Scalable Graph Data Management and Analytics with Hadoop. Tech. Rep. arXiv:1506.00548 (2015)
65. Junghanns M. et al.: Analyzing Extended Property Graphs with Apache Flink. In: Proc. SIGMOD Workshop on Network Data Analytics (2016)
66. Kaoudi, Z., Manolescu, I.: RDF in the Clouds: A Survey. VLDB Journal 24(1) (2015)
67. Karypis, G., Kumar, V.: Multilevelk-way partitioning scheme for irregular graphs. Journal of Parallel and Distributed computing 48(1) (1998)
68. Khayyat Z. et al.: Mizan: A System for Dynamic Load Balancing in Large-scale Graph Processing. In: Proc. EuroSys (2013)
69. Khayyat Z. et al.: Bigdansing: A system for big data cleansing. In: Proc. SIGMOD (2015)
70. Klyne, G., Carroll, J.J.: Resource description framework (RDF): Concepts and abstract syntax (2006)
71. Kolb, L., Sehili, Z., Rahm, E.: Iterative Computation of Connected Graph Components with MapReduce. Datenbank-Spektrum 14(2) (2014)
72. Kolb, L., Thor, A., Rahm, E.: Dedoop: efficient deduplication with Hadoop. PVLDB 5(12) (2012)
73. Koller, D., Friedman, N.: Probabilistic graphical models: principles and techniques (2009)
74. Kyrola, A., Blelloch, G., Guestrin, C.: GraphChi: Large-Scale Graph Computation on Just a PC. In: Proc. OSDI (2012)
75. Lin, J., Schatz, M.: Design Patterns for Efficient Graph Algorithms in MapReduce. In: Proc. 8th Workshop on Mining and Learning with Graphs (2010)
76. Low Y. et al.: Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud. PVLDB 5(8) (2012)
77. Lu, Y., Cheng, J., Yan, D., Wu, H.: Large-scale distributed graph computing systems: An experimental evaluation. PVLDB 8(3) (2014)
78. Malewicz G. et al.: Pregel: A System for Large-scale Graph Processing. In: Proc. SIGMOD (2010)

79. Martinez-Bazan, N., Gomez-Villamor, S., Escale-Claveras, F.: DEX: A high-performance graph database management system. In: Proc. ICDEW (2011)
80. McColl R. et al.: A Performance Evaluation of Open Source Graph Databases. In: Proc. PPAAW (2014)
81. McCune, R.R., Weninger, T., Madey, G.: Thinking like a vertex: a survey of vertex-centric frameworks for large-scale distributed graph processing. ACM Computing Surveys (CSUR) 48(2) (2015)
82. McSherry F. et al.: Composable incremental and iterative data-parallel computation with naiad. Tech. Rep. MSR-TR-2012-105 (October 2012)
83. Miller, J.J.: Graph database applications and concepts with Neo4j. In: Proc. Southern Association for Information Systems Conf. vol. 2324 (2013)
84. Mondal, J., Deshpande, A.: Managing large dynamic graphs efficiently. In: Proc. SIGMOD (2012)
85. Murray D. G. et al.: Naiad: A Timely Dataflow System. In: Proc. 24th ACM Symposium on Operating Systems Principles. SOSP '13 (2013)
86. Nehme, R., Bruno, N.: Automated partitioning design in parallel database systems. In: Proc. SIGMOD (2011)
87. Nickel, M., Murphy, K., Tresp, V., Gabrilovich, E.: A review of relational machine learning for knowledge graphs. Proc. IEEE 104(1) (2016)
88. Petermann A. et al.: BIIIG: Enabling Business Intelligence with Integrated Instance Graphs. In: Proc. ICDEW (2014)
89. Petermann A. et al.: FoodBroker-Generating Synthetic Datasets for Graph-Based Business Analytics. In: Big Data Benchmarking (2014)
90. Petermann A. et al.: Graph-based Data Integration and Business Intelligence with BIIIG. PVLDB 7(13) (2014)
91. Poulovassilis, A., Levene, M.: A nested-graph model for the representation and manipulation of complex objects. ACM Transactions on Information Systems (TOIS) 12(1) (1994)
92. Raghavan U. N. et al.: Near linear time algorithm to detect community structures in large-scale networks. Phys. Rev. E 76, 036106 (2007)
93. Rahimian F. et al.: Distributed vertex-cut partitioning. In: Distributed Applications and Interoperable Systems (2014)
94. Rahm, E.: The case for holistic data integration. In: Advances in Databases and Information Systems (2016)
95. Rao J. et al.: Automating physical database design in a parallel database. In: Proc. SIGMOD (2002)
96. Rodriguez, M.A.: The gremlin graph traversal machine and language (invited talk). In: Proc. 15th Symposium on Database Programming Languages (2015)
97. Rodriguez, M.A., Neubauer, P.: Constructions from dots and lines. Bulletin of the American Society for Information Science and Technology 36(6) (2010)
98. Roy A. et al.: Chaos: Scale-out graph processing from secondary storage. In: Proc. 25th Symposium on Operating Systems Principles (2015)
99. Rudolf M. et al.: The graph story of the SAP HANA database. In: Proc. BTW (2013)
100. Sakr, S., Liu, A., Fayoumi, A.G.: The family of mapreduce and large-scale data processing systems. ACM Computing Surveys (CSUR) 46(1) (2013)
101. Salihoglu, S., Widom, J.: GPS: A Graph Processing System. In: Proc. 25th International Conference on Scientific and Statistical Database Management. SSDBM (2013)
102. Satish N. et al.: Navigating the maze of graph analytics frameworks using massive graph datasets. In: Proc. SIGMOD (2014)

103. Shim, K.: MapReduce Algorithms for Big Data Analysis. PVLDB 5(12) (2012)
104. Stanton, I., Kliot, G.: Streaming graph partitioning for large distributed graphs. In: Proc. SIGKDD
105. Stutz, P., Bernstein, A., Cohen, W.: Signal/collect: Graph algorithms for the (semantic) web. In: ISWC (2010)
106. Sun W. et al.: SQLGraph: an efficient relational-based property graph store. In: Proc. SIGMOD (2015)
107. Teixeira C. et al.: Arabesque: a system for distributed graph mining. In: Proc. 25th Symposium on Operating Systems Principles (2015)
108. Tian, Y., Hankins, R.A., Patel, J.M.: Efficient aggregation for graph summarization. In: Proc. SIGMOD (2008)
109. Tian Y. et al.: From "Think Like a Vertex" to "Think Like a Graph". PVLDB 7(3) (Nov 2013)
110. Turk-Browne, N.B.: Functional interactions as big data in the human brain. Science 342(6158) (2013)
111. Valiant, L.G.: A bridging model for parallel computation. CACM 33(8) (1990)
112. Wang X.H. et al.: Ontology based context modeling and reasoning using owl. In: Pervasive Computing and Communications Workshops (2004)
113. Wang Z. et al.: Pagrol: parallel graph olap over large-scale attributed graphs. In: Proc. ICDE (2014)
114. Xia Y. et al.: Graph analytics and storage. In: IEEE Big Data (2014)
115. Xin R.S. et al.: GraphX: A Resilient Distributed Graph System on Spark. In: First International Workshop on Graph Data Management Experiences and Systems. GRADES '13 (2013)
116. Xin R.S. et al.: GraphX: Unifying Data-Parallel and Graph-Parallel Analytics. Tech. Rep. arxiv/1402.2394 (2014)
117. Yuan P. et al.: Triplebit: a fast and compact system for large scale rdf data. PVLDB 6(7) (2013)
118. Zaharia M. et al.: Spark: Cluster Computing with Working Sets. In: Proc. 2Nd USENIX Conference on Hot Topics in Cloud Computing. HotCloud'10 (2010)
119. Zhang, N., Tian, Y., Patel, J.M.: Discovery-driven graph summarization. In: Proc. ICDE (2010)
120. Zhao P. et al.: Graph cube: on warehousing and OLAP multidimensional networks. In: Proc. SIGMOD (2011)
121. Zhao Y. et al.: Evaluation and Analysis of Distributed Graph-Parallel Processing Frameworks. Journal of Cyber Security and Mobility 3(3) (2014)