# Graph Mining for Complex Data Analytics

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

***Abstract*—Complex data analytics that involve data mining often comprise not only a single algorithm but also further data processing steps, for example, to restrict the search space or to filter the result. We demonstrate graph mining with GRADOOP, the first scalable system supporting declarative analytical programs composed from multiple graph operations. We use a business intelligence example including frequent subgraph mining to highlight the analytical capabilities enabled by such programs. The results can be visualized and, to show its ease of use, the program can be modified on visitors request. GRADOOP is built on top of state-of-the-art big data technology and out-of-the-box horizontally scalable. Its source code is publicly available and designed for easy extensibility. We offer to the graph mining community, to apply GRADOOP in large scale use cases and to contribute further algorithms.**

***Index Terms*—Graph Mining, Business Intelligence**

## I. INTRODUCTION

The identification of frequent patterns is an established approach to data mining [1]. However, in real-world applications, the actual mining algorithm is often combined with other operations, e.g., to identify frequent sets of item groups in canceled orders. Such complex analyses require additional data processing steps, e.g., to identify canceled orders and to project items to their groups. For non-graph data (e.g., relational), multi-step analyses are already supported by analytical toolkits of large database vendors or machine learning libraries of big data processing platforms [2].

In comparison to mining frequent itemsets, mining graph patterns additionally provides information about relationships among items that appear together. For example, let a graph collection represent business process executions [3] including master data (e.g., employees, customers, products) and transactional data (e.g., quotations, invoices) as well as their mutual relationships. Here, an analyst might be interested in frequent graph patterns representing the joint occurrence of certain master data objects including their roles, i.e., how were these objects involved during the process execution. Examples for such patterns are shown in Figure 1. To identify such patterns, we do not consider all graphs of the data set, but only such showing financial loss. Therefore, as part of the analysis, we need to calculate the financial result for all graphs and select the lossy ones. Further on, we need to relabel vertices as we want our patterns to include master data identities (e.g., which employee or which product) but only types of transactional data and relationships to understand master data interaction.

Although great research efforts have been made to develop efficient graph mining algorithms, most implementations are stand-alone research prototypes. Thus, applying graph mining algorithms to complex analyses like our example requires the combination of different tools. This quickly becomes a difficult task as they may differ in regard to the underlying platform, graph models, availability (source code, binaries, on request only, etc) or in- and output formats. To support such multi-step graph analytics in a single system, we started developing GRADOOP [4]. The system enables flexible evaluation and modification of graph data by declarative analytical programs composed from different graph operations. The first graph mining algorithm provided by GRADOOP is the extraction of frequent subgraphs from a distributed graph collection. We demonstrate how our motivating example can be implemented in a single program that includes frequent subgraph mining.

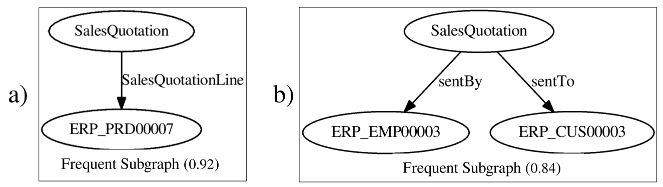


Fig. 1. Example result visualization of the demonstration program using Graphviz. Graph a) represents a quotation containing a specific product. Graph b) shows the common involvement of a specific employee (ERP_EMP..) and a specific customer (ERP_CUS..) in the same quotation.

In the following sections, we provide a brief overview about GRADOOP and its graph mining capabilities (Section II), describe our demonstration scenario in more detail (Section III) and summarize our current research focus (Section IV).

## II. GRAPH MINING WITH GRADOOP

GRADOOP [4] is a system for declarative graph analytics supporting the combination of multiple graph operators and graph mining algorithms in a single program. Graph data is represented within the so-called extended property graph model (EPGM). The EPGM is based on the property graph model [5], i.e., on directed multigraphs supporting identifiers, labels and named attributes (properties) for vertices as well as edges. As an extension, it supports so-called *logical graphs*, which are logical partitions of a base graph. Thus, it is possible to analyze single graphs as well as collections of logical graphs (see Figure 2), for example, to fit the single graph and the transactional setting of frequent subgraph mining [6]. Logical graphs further support labels and properties, for example, to mark a graph to represent a frequent subgraph and to store its support. The EPGM further contains a set of general-purpose operators with either single graphs or graph collections as in- or output. These operators include such extracting a graph collection from a single graph (e.g. graph pattern matching queries [7]) and vice versa.

### *A. Relation to other systems*

GRADOOP is implemented on top of the big data processing platform Apache Flink [8]. The fundamental concepts of platforms like Apache Flink or Apache Spark [9] are distributed collections of data objects (*datasets*) and *transformations* of these. A *complex program* is represented by a directed acyclic graph (DAG) where vertices represent datasets and edges represent transformations. In comparison to MapReduce [10], these frameworks offer a wider range of operators as well as the possibility to hold data in distributed main memory between single processing steps. There already exist graph processing libraries based on such systems (e.g., Apache Spark GraphX [11], Apache Flink Gelly [8]). However, these libraries neither include a rich data model like the property graph model nor operators based thereon. Thus, answering complex analytical questions that involve multiple graph operations still requires notable programming effort. Further on, there is no abstraction of graph collections to fit the transactional setting of graph mining algorithms. In contrast, GRADOOP supports analytical programs in form of DAGs where vertices represent either graphs or graph collections and edges represent either built-in operators or custom algorithms, e.g., such for graph mining. Additionally, GRADOOP supports different data sources and sinks (e.g., files, HBase). Programs are declared using a Java API representing our domain specific language GrALa (**Gr**aph **A**nalytical **La**nguage). Below the user-facing API, operators and algorithms are mapped to Apache Flink datasets and transformations and, thus, are horizontally scalable by default. The source code of GRADOOP is available online under the GPL license[1].

### *B. Algorithm integration*

We distinguish between general-purpose operators that are part of our data model (e.g., the union of two graph collections) and specific algorithms (e.g. frequent subgraph mining). To support custom graph mining algorithms, GRADOOP offers the generic *call* operator and fitting Java interfaces whose implementations can be included in analytical programs. The provided interfaces cover all algorithms with one or two graphs or graph collections as input and a graph or a graph collection as output in arbitrary constellation. For example, a graph partitioning algorithm takes a single graph as input and results into a collection of partitions. To be compatible with the EPGM, all algorithm implementations must be capable to handle multigraphs. For example, in a social network the same user might be member and leader of the same group which is usually represented by two parallel edges. Further on, it must be considered that EPGM edges are always directed. However, to also support undirected graph data (e.g., chemical compounds) algorithm implementations may provide a parameter to optionally ignore edge directions.

[1]www.gradoop.com

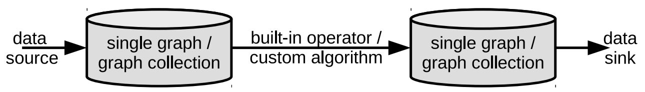

data source
single graph / graph collection
built-in operator / custom algorithm
single graph / graph collection
data sink

Fig. 2. Basic GRADOOP analytical program.

### *C. Frequent Subgraph Mining*

The first graph mining algorithm provided by GRADOOP addresses the problem of frequent subgraph mining (FSM) in the transactional setting, i.e., the input is a collection of vertex- and edge-labeled graphs $\mathcal{G} = \{G_0, .., G_n\}$ and the output is the complete set of frequent connected graph patterns $\mathcal{F} = \{P_0, .., P_m\}$. A graph will be considered to support a pattern, if at least one subgraph exists that is isomorphic to the pattern. A pattern will be considered to be frequent, if its *support* $s(P)$ (share of graphs supporting a pattern) is above a given minimum support $s_{min}$ such that $s(P) \geq s_{min} \Leftrightarrow P \in \mathcal{F}$. While existing work [6] including distributed approaches based on MapReduce [12], [13], [14] mostly support only undirected graphs without parallel edges, we extended the well known gSpan algorithm [15] to support directed multigraphs. It should be noted that the algorithm only evaluates labels of vertices and edges but relevant property values can be added to the label, if required (see Section III).

The gSpan algorithm follows the *pattern growth* approach: First, starting from $k = 1$, every graph reports supported $k$-edge graph patterns. Second, $s(P)$ is counted and frequent graph patterns are determined using $s_{min}$. Third, every graph grows $k+1$-edge children of frequent patterns. This procedure is repeated until no more frequent patterns can be found. Among similar algorithms, gSpan is an efficient representative as it applies frequency-based relabeling of vertices and edges, a systematic pattern growth avoiding multiple discovery of isomorphic patterns and has no need for subgraph isomorphism testing through canonical labeling of graph patterns [16].

In contrast to the existing MapReduce based gSpan derivate [13], we even adapt the relabeling to our parallel implementation. We first determine frequent vertex labels and drop vertices showing infrequent labels including their incident edges. Second, we determine frequent labels of the remaining edges and drop the ones with infrequent labels. For both, vertices and edges, input labels are encoded based on their support, whereas a high support leads to a short label. After the mining process that includes the gSpan systematic pattern growth, labels of frequent patterns are decoded using previously stored label dictionaries.

To support directed graphs, we extended canonical labeling and its lexicographic order. The gSpan algorithm is using *minimum DFS codes* as canonical labels. A DFS code $C = \langle s_0, s_1, .., s_k \rangle$ is a sequence of edge traversals describing a depth first search. A single traversal step is represented by a pentuple $s = \langle t_a, t_b, l_a, l_e, l_b \rangle$ expressing the traversal of an edge with label $l_e$ from a vertex with label $l_a$ to a vertex with label $l_b$. Additionally, it contains initial discovery times of both vertices $t_a, t_b$. For example, if $t_b = k$, then the vertex was first visited in the last extension to a $k$-edge pattern. As a graph can be traversed in multiple ways, a lexicographic order is used to determine a minimum among all possible DFS codes. The original order [16] is a combination of linear orders over vertex times, traversal start vertex label, edge label and traversal end vertex label. To support directed graphs, we extended traversal steps to hextuples $s = \langle t_a, t_b, l_a, d, l_e, l_b \rangle$ by an additional direction indicator $d$ expressing wether an outgoing or incoming edge was traversed and extended the lexicographic order respectively with $outgoing < incoming$.

Our multigraph extension affects the representation of *embeddings* (pattern instances). In a simple graph without parallel edges, an edge is identified by its incident vertices. Thus, a mapping between vertices and initial discovery times is sufficient to transitively map edges to DFS traversal steps. In a multigraph, two or more edges may connect the same pair of vertices but edges provide an identifier. In consequence, an embedding needs to contain a mapping between extension number $k$ and edge identifier. For directed graphs, an explicit vertex mapping is not required as it can be derived transitively from edge direction and traversal direction. For undirected multigraphs, both mappings are required.

### D. Selected operators

In the following, we sketch a subset of EPGM operators used in our demonstration example. A detailed overview about all operators can be found in [4].

An important preprocessing operator is *transformation*. The operator preserves the graph structure but allows to modify labels and properties of vertices, edges and graphs by three respective user-defined functions. An example application is the combination of multiple properties, for example, first and last name of vertices representing persons. A further primitive operator enabling the quantitative evaluation of graphs is *aggregation*. It stores the scalar result of a built-in or user-defined aggregate function in a graph property. Within the scope of an aggregate function, all vertices and edges of a graph are accessible including their labels and properties. This allows, for example, to count the number of edges or to sum all values of a certain vertex property. An operator specific to graph collections is *selection*. The operator returns the subset of an input graph collection for which a user-defined predicate function evaluates to true. The scope of a predicate function is limited to graph label and graph properties, for example, to select graphs by a previously aggregated value (e.g., edge count, business measure).

### E. Data sources and sinks

Every analytical program has to contain one or more data sources and one data sink. Besides direct HBase storage, GRADOOP supports multiple in- and output file formats, for example, JSON, DOT[2] and a graph list format often used by FSM prototypes. Similar to algorithms, GRADOOP provides interfaces to easily implement additional sources and sinks for arbitrary databases or further file formats.

[2]http://www.graphviz.org/doc/info/lang.html

## III. DEMONSTRATION DESCRIPTION

During the demonstration, we will use a business intelligence scenario to exemplify a complex analytical program including frequent subgraph mining. The program aims to identify characteristic patterns for bad business process outcomes. Therefore, it extracts frequent subgraphs from a graph collection representing lossy business process executions. Although all included operations are designed to be executed on shared-nothing clusters, we execute the program on a single machine to demonstrate functionality and ease of use. In this setup, the program is executed in a so-called local cluster, i.e., is parallelized among threads without shared memory. On site, the program itself as well as operator parameters can be modified on visitors request. Results will be visualized using the tool Graphviz [17]. In the following, we will describe the used data set and the demonstration program in more detail.

### A. Input Data

The demonstration input data is a business data network containing transactional data, master data and their relationships. The data was generated using FoodBroker [18], a data generator based on business process simulation, and integrated into a single instance graph using the BIIIG approach [3]. The data integration process was already demonstrated earlier [19] which is why we just summarize its result to understand the starting point of the analytical program:

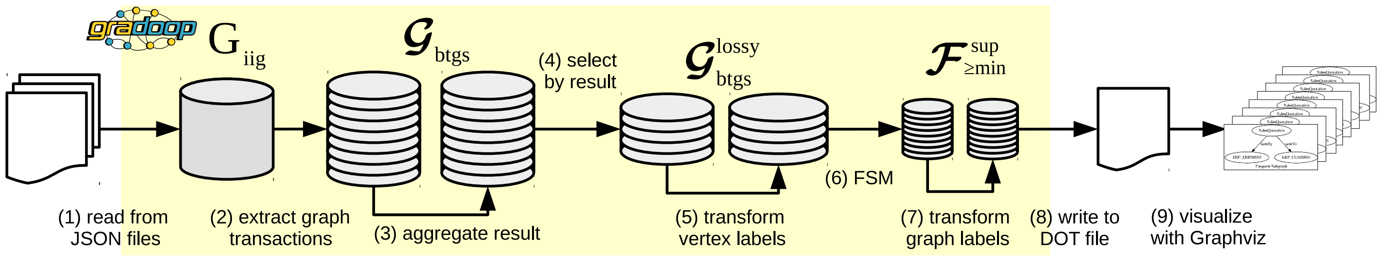
In the integrated instance graph, every vertex represents a domain data object and every edge represents a relationship. Vertices are labeled by the class of their respective data objects (e.g., SalesQuotation, Employee) and edges are labeled by their semantic relationship type (e.g. processedBy). Further on, vertices and edges contain properties representing either dimensional attributes (e.g. name, group) or facts about the process execution (e.g. revenue, expense). Additionally, every vertex includes two system properties. First, sourceId, an identifier concatenated from source system, class and local id and, second, superClass, associating every vertex to represent either master or transactional data. Both properties result from data integration [3].

### B. Program description

The demonstration program including a data sample is available online[3]. The program can be executed by cloning the GRADOOP repository and running the example program's main method. Figure 3 illustrates the program's DAG, where varying shapes are used to represent different data formats and arrows in between indicate applied operators and algorithms. Following, we will describe its single steps in more detail:

1) The integrated instance graph is read from a data source. We choose the JSON format for the demonstration because of its good human readability.
2) An algorithm is applied to the integrated instance graph to extract a collection of so-called business transaction

[3]https://github.com/dbs-leipzig/gradoop/blob/master/gradoop-examples/src/main/java/org/gradoop/examples/biiig/FrequentLossPatterns.java



gradoop
$G_{iig}$
$\mathcal{G}_{btgs}$
(4) select by result
$\mathcal{G}^{lossy}_{btgs}$
$\mathcal{F}^{sup}_{\geq min}$
(6) FSM
(1) read from JSON files
(2) extract graph transactions
(3) aggregate result
(5) transform vertex labels
(7) transform graph labels
(8) write to DOT file
(9) visualize with Graphviz

Fig. 3. Example analytical program used in the demonstration.

graphs, where every graph represents a single business process execution (more details can be found in [3]).

3) To calculate the financial result, a domain-specific aggregate function is applied to every graph and the aggregate value is stored in a graph property.
4) Lossy graphs are selected (financial result $< 0$).
5) The remaining graphs are transformed in preparation for pattern mining. In particular, labels of master data vertices are modified to show their source identifier instead of their class, transactional vertices as well as edges keep their labels and all properties are dropped.
6) The frequent subgraph mining algorithm is applied to the prepared collection. Every frequent subgraph stores its support in a graph property.
7) As Graphviz provides no support for graph properties, the support property value is added to the graph label in a result transformation.
8) The frequent subgraphs are written to a data sink creating a file in the DOT format.
9) Graphviz is used to generate a postscript file containing a single page for every frequent subgraph. Two example graphs are shown by Figure 1.

Besides changing the program, visitors can modify steps 3 to 7 (the aggregate function, predicates and parameters) and visually trace the changing result.

## IV. Ongoing Research

The demonstration shows the novel analytical capabilities enabled by the seamless combination of different approaches to graph mining and graph analytics. To the best of our knowledge, GRADOOP is the first system supporting such analytics within a single declarative program. The foundation of GRADOOP is ongoing research in the fields of graph data management, graph analytics and graph mining. Regarding the latter, we investigate in methods using more meaningful interestingness measures than minimum support, for example, to identify patterns correlating with certain business measure values [20]. Due to the huge data volume in many applications and the high computational complexity of graph pattern mining, our second focus is the efficient parallelization of graph algorithms using shared-nothing clusters. Due to limited space, we only gave a brief introduction to operator and algorithm implementations. More details, for example, about our approach to distributed frequent subgraph mining, will follow in future publications.

## Acknowledgment

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

## References

[1] C. C. Aggarwal and J. Han, *Frequent Pattern Mining*. Springer, 2014.
[2] X. Meng, J. Bradley, B. Yuvaz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen *et al.*, "Mllib: Machine learning in apache spark," *JMLR*, vol. 17, no. 34, pp. 1–7, 2016.
[3] A. Petermann, M. Junghanns, R. Müller, and E. Rahm, "BIIIG: Enabling Business Intelligence with Integrated Instance Graphs," in *Data Engineering Workshops (ICDEW)*, 2014, pp. 4–11.
[4] M. Junghanns, A. Petermann, N. Teichmann, K. Gómez, and E. Rahm, "Analyzing extended property graphs with apache flink," in *Proc. ACM SIGMOD Workshop on Network Data Analytics*, 2016, pp. 3:1–3:8.
[5] M. A. Rodriguez and P. Neubauer, "Constructions from dots and lines," *Bulletin of the American Society for Information Science and Technology*, vol. 36, no. 6, pp. 35–41, 2010.
[6] C. Jiang, F. Coenen, and M. Zito, "A survey of frequent subgraph mining algorithms," *The Knowledge Engineering Review*, vol. 28, no. 01, pp. 75–105, 2013.
[7] R. Angles, "A comparison of current graph database models," in *Data Engineering Workshops (ICDEW)*, 2012, pp. 171–177.
[8] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache flink: Stream and batch processing in a single engine," *Data Engineering*, p. 28, 2015.
[9] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *USENIX Conf. on Networked Systems Design and Implementation*, 2012.
[10] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Comm. of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.
[11] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica, "GraphX: A Resilient Distributed Graph System on Spark," in *Proc. GRADES*, 2013.
[12] S. Hill, B. Srichandan, and R. Sunderraman, "An iterative mapreduce approach to frequent subgraph mining in biological datasets," in *Proc. ACM Conf. on Bioinf., Computational Biology and Biomedicine*, 2012.
[13] W. Lu, G. Chen, A. K. Tung, and F. Zhao, "Efficiently extracting frequent subgraphs using mapreduce," in *Big Data, 2013 IEEE International Conference on*. IEEE, 2013, pp. 639–647.
[14] W. Lin, X. Xiao, and G. Ghinita, "Large-scale frequent subgraph mining in mapreduce," in *Data Engineering (ICDE), Int. Conf. on*, 2014, pp. 844–855.
[15] X. Yan and J. Han, "gspan: Graph-based substructure pattern mining," in *Data Mining (ICDM), Int. Conf. on*, 2002, pp. 721–724.
[16] ——, "gspan: Graph-based substructure pattern mining," in *Technical Report UIUCDCS-R-2002.2296*, 2002.
[17] E. R. Gansner, "Drawing graphs with graphviz," Technical report, AT&T Bell Laboratories, Murray, Tech. Rep, Tech. Rep., 2009.
[18] A. Petermann, M. Junghanns, R. Müller, and E. Rahm, "FoodBroker - Generating Synthetic Datasets for Graph-Based Business Analytics," in *Proc. WBDB*, 2014.
[19] ——, "Graph-based Data Integration and Business Intelligence with BIIIG," *PVLDB*, vol. 7, no. 13, 2014.
[20] A. Petermann and M. Junghanns, "Scalable business intelligence with graph collections," *it-Information Technology*, vol. 58, no. 4, 2016.