Scalable Business Intelligence with Graph Collections

André Petermann and Martin Junghanns

Abstract: Using graph data models for business intelligence applications is a novel and promising approach. In contrast to traditional data warehouse models, graph models enable the mining of relationship patterns. In our prior work, we introduced an approach to graph-based data integration and analytics called BIIIG (Business Intelligence with Integrated Instance Graphs). In this work, we compare state-of-the-art systems for graph data management and analytics with regard to the support for our approach in Big Data scenarios. To exemplify the analytical value of graph models for business intelligence, we propose an analytical workflow to extract knowledge from graph-integrated business data. Finally, we show how we use Gradoop, a novel framework for distributed graph analytics, to implement our approach.

ACM CCS: Information systems → Business intelligence, Graph-based database models, Data mining, Parallel and distributed DBMSs,

Keywords: business intelligence, property graph model, graph pattern mining

1 Introduction

Graph data models enable flexible and powerful evaluations of domain objects and their relationships. Thus, graph analytics and graph mining has become popular among researchers of different domains, for example, to analyze social networks, the world wide web or chemical and biological structures. Also business data can be abstracted as a graph where vertices represent heterogeneous domain objects like products, sales orders or phone calls and edges represent relationships between those objects. However, the use of graph models for business intelligence has found not much attention yet.

In our prior work [1], we introduced an approach to graph-based data integration and business intelligence. Beside analyzing single graphs, we proposed the use of graph collections to analyze interrelated data objects and their relationships. In a respective graph collection $\mathcal{G} = \langle G_1, G_2, .., G_n \rangle$, every graph G reflects a *case* (e.g., a business process execution) and all its related data objects and relationships. This representation enables the aggregation of business measures (e.g., the financial result) for each case and the comparison to other cases, for example, to reveal correlations between certain measure value ranges (e.g., profit and loss) and meaningful graph patterns. Applying our approach to data of large scale enterprises may lead to millions of graphs with hundreds of vertices and edges each.

Beside scalability, implementing our approach requires a data model supporting graph collections, respective operators and the composition of operators to analytical workflows. We already performed a first evaluation of our approach on top of a state-of-the-art graph database system [2]. Although we thoroughly chose a mature system providing advanced analytical features, it could not fulfill all of our requirements. In particular, the data model was lacking support for graph collections and the system did not scale well in Big Data scenarios.

In this article, we present further findings of our ongoing work on graph-based business intelligence. Our major contributions are as follows:

- We provide a comparison of recent systems for graph data management and analytics with regard to their functionality and scalability.
- We highlight the novel analytical capabilities enabled by graph collections using an example business intelligence workflow.
- We show how to use Gradoop, a novel framework for scalable graph analytics, to implement such workflows based on distributed computing.

The remainder of this article is organized as follows: In Section 2, we provide a comparison of current systems for graph data management and analytics and briefly introduce the Gradoop framework. In Section 3, we review our prior work on business intelligence with integrated graph data. In Section 4, we exemplify analytics with

	Graph Database Systems	Graph Processing Systems	Graph Extensions of Distributed Dataflow Framew.	Gradoop
Examples	Neo4j, Titan	Pregel, Giraph	Gelly, GraphX	
Workload	OLTP/Queries	Analytical	Analytical	Analytical
Data Model	PGM	Generic	Generic	EPGM
Workflows	No	No	Yes	Yes
Graph Collections	No	No	No	Yes

Table 1: Feature comparison of different approaches to graph data management and analytics.

graph collections including the adoption to Gradoop. Finally, in Section 5, we discuss related work and end with conclusions and future work in Section 6.

2 Systems for Graph Data Management and Graph Analytics

In this section, we compare existing approaches to graph data management and analytics. An important criterion is the support for graph collections. Representing data by graph collections is not only suitable for business intelligence but also for other analytics based on the comparison of graphs or subgraphs, for example, to analyze communities in a social network. Beside the abstraction of graph collections, such analytics also require the support for graph attributes, e.g., to store graph measures, analytical collection operators and the composition of such operators to analytical workflows.

2.1 System Comparison

In recent years, different systems for management and analytics of graph data appeared in research and industry. One can group those systems into the three categories of graph database systems, graph processing systems and graph processing extensions for distributed dataflow frameworks. A comparative overview of common and distinctive features of the considered system categories is shown in Table 1. We leave out dedicated RDF stores [10] as expressing complex entities using RDF requires reification and leads to very voluminous representations of attributed vertices and edges [11].

The most significant difference between *graph database systems* (GDBS) [6] and *graph processing systems* (GPS) is their application focus. While GDBS like Neo4j[1] or Titan[2] primarily focus on OLTP and query workload, GPS, such as Google Pregel [7] and its open source implementation Apache Giraph[3], are dedicated to graph analytics. With regard to their respective foci, GDBS support CRUD operations for vertices and edges as well as efficient local queries. Here, the term *local* refers to queries relative to one or more fixed entry points, for example, to find paths between two given users in a social network. However, many analytical algorithms, for example, to calculate the page rank of websites, require processing the whole graph iteratively (*global processing*). While GDBS typically show poor scalability for such algorithms on very large graphs, GPS use distributed in-memory execution to enable their efficient execution.

Another fundamental difference between GDBS and GPS are the supported data models. GDBS offer semantically rich data models like the property graph model (PGM) [5], where vertices and edges are labeled and attributed. The use of such a predefined data model allows GDBS to offer query languages and advanced visualization features. In contrast, the data models of GPS are generic, which means only a plain abstraction of vertices and edges is provided but the format of attached data needs to be defined by the user. In consequence, operators need to be user-defined, too. For example, these systems allow simulating the property graph model by attaching strings and key-value maps to vertices and edges but provide no built-in operators involving labels or attributes.

In contrast to dedicated graph systems like GDBS and GPS, *distributed dataflow frameworks* like Apache Spark[4] and Apache Flink[5] were developed to support generic analytical workflows in Big Data processing. Although they are designed for workflows of set-oriented operations on static and streaming data, additional *graph processing extensions* like GraphX [8] on Spark or Gelly on Flink allow them acting as GPS, where their characteristics are the same as those of GPS. However, the support for operator workflows provided by the underlying frameworks additionally allows chaining multiple graph operations as well as mixing graph and set-oriented operations within the same distributed execution environment.

Some GDBS, for example Titan, support the distributed storage of very large graphs and the virtual integration of GPS and distributed dataflow frameworks[6]. However, in this approach data needs to be moved from the database to the GPS and vice versa for each processing step.

[1] http://neo4j.com/
[2] http://thinkaurelius.github.io/titan/
[3] http://giraph.apache.org/
[4] http://spark.apache.org/
[5] http://flink.apache.org/
[6] http://s3.thinkaurelius.com/docs/titan/1.0.0/

None of the systems discussed so far supports graph collections including graph attributes and collection operators. Motivated by that, the Leipzig University started developing a research platform for large-scale graph analytics on heterogeneous graphs named Gradoop [3]. The data model of Gradoop is designed to support single graphs and graph collections including respective operators. Although Gradoop focuses on analytical workload, the framework offers distributed graph processing in combination with a semantically rich data model, a large set of built-in analytical operators and a declarative way to specify workflows, which differentiates Gradoop from graph extensions for distributed dataflow frameworks.

2.2 Introduction to Gradoop

Gradoop is an open source framework for scalable graph analytics. After an initial prototype based on MapReduce and Apache Giraph [3], the current version [4] is implemented on top of the distributed dataflow framework Apache Flink, the successor of the former research project Stratosphere [9]. Gradoop maps graph representation and graph operators to the data model and transformations provided by Flink and benefits from its distributed execution environment to provide scalability for very large data volumes.

The development of Gradoop was motivated by missing functionalities of graph processing systems and the insufficient scalability of graph databases. In particular, Gradoop supports the declaration of analytical workflows including operators for single graphs and graph collections. Graph representation and respective operators are part of the so-called *Extended Property Graph Model* (EPGM) . Like the basic property graph model [5], the EPGM describes directed and attributed multigraphs. To support the definition of graph collections, it additionally provides the concept of logical graphs. In the following, we provide a brief overview of the EPGM and its analytical operators. However, we introduce only selected operators used in the remainder of this paper. Gradoop is work in progress and a complete and constantly updated list of operators can be found online[7].

Graphs are represented within an *EPGM database*, formally an octuple $DB = \langle \mathcal{V}, \mathcal{E}, \mathcal{L}, T, \tau, K, A, \kappa \rangle$, where $\mathcal{V} = \{v_i\}$ is the vertex space, $\mathcal{E} = \{e_k\}$ is the edge space and $\mathcal{L} = \{G_m\}$ is the set of logical graphs. Vertices, edges and (logical) graphs are identified by the respective indices $i, k, m \in \mathbb{N}$. Edges $\langle v_s, v_t \rangle \in \mathcal{E} \mid v_s, v_t \in \mathcal{V}$ direct from v_s to v_t and support loops (i.e., $s = t$). The database further contains the alphabet of type labels T, the alphabet of property keys K and the value set A. Vertices, edges and graphs show type labels, described by the mapping $\tau : (\mathcal{V} \cup \mathcal{E} \cup \mathcal{L}) \to T$, and may have arbitrary properties (key-value pairs), expressed by the function $\kappa : (\mathcal{V} \cup \mathcal{E} \cup \mathcal{L}) \times K \to A$.

A *logical graph* $\langle V, E \rangle \in \mathcal{L}$ is a pair of a subset of vertices $V \subseteq \mathcal{V}$ and a subset of edges $E \subseteq \mathcal{E}$. The vertices and edges of two logical graphs may overlap without redundant storage, for example, to represent different communities in a social network containing common users. A *graph collection* is defined to be a *n-tuple* of logical graphs $\mathcal{G} \subseteq \mathcal{L}^n$ and, thus, allows sorting graphs, for example, by graph measure values.

To aggregate such measures, Gradoop provides an *aggregation* operator. The operator is called for a logical graph $G' = G.\texttt{aggregate}(k_\alpha, \alpha)$, where an aggregate function $\alpha : \mathcal{L} \to A$ is applied to an input graph G and the result is stored in a property with key k_α, such that $\kappa(G', k_\alpha) = \alpha(G)$. Aggregation functions may evaluate structural information (e.g., vertex count) as well as vertex and edge properties (e.g., summation of numerical values).

Some analytics only need to consider a part of a graph collection, for example graphs showing an aggregated measure within a specific interval. Therefore, Gradoop provides a *selection* operator. The operator is called for a graph collection $\mathcal{G}' = \mathcal{G}.\texttt{select}(\varphi(G))$ with a given predicate function φ: $\mathcal{L} \to \{true, false\}$ and returns a collection $\mathcal{G}'$ where $\forall G \in \mathcal{G} : G \in \mathcal{G}' \vee \varphi(G) = false$.

Removing and manipulating properties with regard to the requirements of specific evaluations is advantageous, especially for very large data volumes. Therefore, Gradoop offers the *transformation* operator. The operator is called for a logical graph $G' = G.\texttt{transform}(\gamma, \nu, \epsilon)$, where three functions define transformations of the graph itself $\gamma : \mathcal{L} \to \mathcal{L}$ as well as transformations of its vertices $\nu : \mathcal{V} \to \mathcal{V}$ and edges $\epsilon : \mathcal{E} \to \mathcal{E}$. Those functions allow the modification of labels and properties, but preserve identifiers and the graph structure.

Graph operators like aggregation or transformation will not only be useful for single graphs, but also if executed for all graphs of a collection. Thus, Gradoop provides the auxiliary operator *apply*, which is called for a graph collection $\mathcal{G}' = \mathcal{G}.\texttt{apply}(op(G))$ with an arbitrary graph operator $op : \mathcal{L} \to \mathcal{L}$ as argument. Let $\mathcal{G} = \langle G_1, G_2, .., G_n \rangle$ be an input collection, then the output is $\mathcal{G}' = \langle op(G_1), op(G_2), .., op(G_n) \rangle$ under preservation of cardinality and order.

As Gradoop is a framework for general-purpose graph analytics, there may be use cases requiring the execution of specific algorithms which are not covered by the provided operators. For such cases, Gradoop offers the generic *call* operator. There are four variants of the call operator to cover all possible in- and output combinations $\mathcal{L} \to \mathcal{L}$, $\mathcal{L} \to \mathcal{L}^n$, $\mathcal{L}^n \to \mathcal{L}$ and $\mathcal{L}^n \to \mathcal{L}^n$. For example, $\mathcal{G}' = G.\texttt{callForCollection}(alg(G), P)$ is called for a single graph G, executes a fitting algorithm $alg : \mathcal{L} \to \mathcal{L}^n$ under given parameters $P = \{p_0, .., p_n\}$ and outputs a graph collection $\mathcal{G}'$. To use custom algorithms with one of the call operators, they need to be implemented using the Java programming language and match one of four respective interfaces.

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

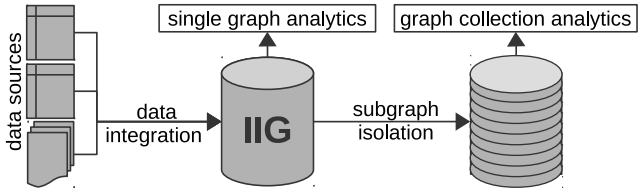

Figure 1: Overview of the BIIIG approach.

Process	TD types	MD types	Measure
Trade	Phone call Sales order	Customer Product	Financial result
Car assembly	Assembly step Car (instance)	Worker Car model	Assembly time
Medical treatment	Examination Prescription	Doctor Drug	Recovery rate

Table 2: Example transactional (TD) types, master data (MD) types and measures of different high-level business processes.

3 The BIIIG Approach

In our prior work [1], we proposed an approach to graph-based business intelligence called BIIIG (**B**usiness **I**ntelligence with **I**ntegrated **I**nstance **G**raphs). An overview of its components is shown in Figure 1. After a semi-automated data integration process, data from one or more sources is represented within a single *integrated instance graph* (IIG). In this graph, every domain data object is represented by a vertex and every relationship by an edge. Both, vertices and edges are self-descriptive by providing a type label and arbitrary named properties. Details about our data integration strategy can be found in [1].

The integrated instance graph can be used as the base for *single graph analytics*, for example, creating summaries of a business data network by grouping vertices and edges based on types and properties [4]. However, one can also apply a specific algorithm to isolate a collection of subgraphs representing logical partitions of this graph, for example, to represent communities in a social network or to outline interrelated domain objects. In this way, BIIIG also enables *graph collection analytics*, where the remainder of this paper will focus on.

In this section, we first discuss the analytical foundation of our approach, where some domain knowledge is introduced to provide comprehensibility. Additionally, we review our representation of interrelated business data by a graph collection, which is the base for the analytics discussed in the next section.

3.1 Analytical Foundation

We observed that typical data recorded by business information systems can be categorized into master and transactional data. Handling both kinds of data according to their particular characteristics is fundamental to the BIIIG approach. In the following, we describe these characteristics, their origin in source data and the respective treatment in analytics. For better understanding, we first introduce some domain specific basics:

Independent from the field of business, the daily operations of enterprises can be abstracted by high-level business processes like trading items, manufacturing goods or providing services. We denote a single execution of a business process by the term *case*. Every case triggers the creation of transactional data objects (e.g., phone calls and sales orders) with references to master data objects (e.g., customers and products). For business analytics, the outcome of a case can be qualified by business measures (e.g., financial result). Table 2 shows examples of transactional data types, master data types and business measures for different business processes.

First, we define *transactional data* (TD) to be data, that is created exclusively for a single case. The main characteristic of TD is their documenting nature, in particular TD objects document the facts of a case. For example, phone call logs document customer interaction or sales orders document sold goods. Further on, TD provides the source values of business measures like monetary values or time durations.

Second, we define *master data* (MD) to be data, which is referenced by many cases. While TD reflects the facts of a case, referenced MD reveals the context of those facts, for example, which customer sent an order, which products were ordered or which employee processed the order. MD typically provides dimensional attributes (e.g., product group or customer name).

In a typical data warehouse model (e.g., star schema), selected transactional data and master data is represented type-wise by fact and dimension tables. However, those models are designed to evaluate facts of a single type by a set of predefined dimensional attributes, which makes them not suitable for evaluations on the case-level. For example, certain cases of car assembly may result in a failed final audit and the analyst wants to identify the reasons. However, there are potentially hundreds of TD objects of tens of types that may have led to this result. Additionally, those TD objects refer to each other and to MD objects.

One can abstract a case as an instance network of domain objects with mostly m:n relationships. With BIIIG, we implement this natural abstraction using a schema-less graph model and graph collections. Thus, we are able to:

- aggregate base values of transactional data to measures on the case level.
- compute occurrence and quantities of transactional data types to represent the facts happened in a case.
- consider relationships within transactional data to evaluate mutual inferences between facts.
- consider relationships between transactional and master data as well as selected dimensional attributes to reveal the context of facts.

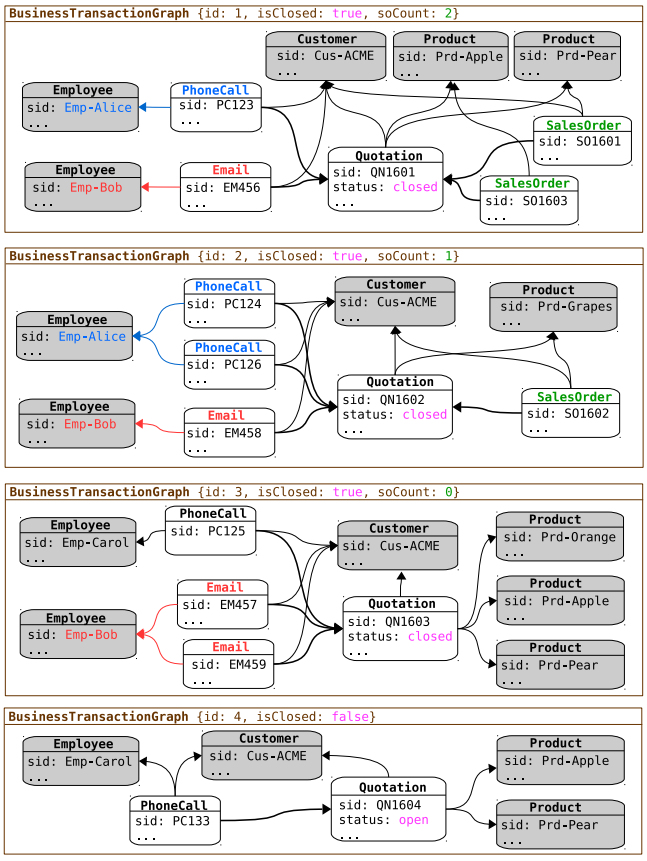

Figure 2: Example cutouts of business transaction graphs with aggregated graph measures 'isClosed' and 'soCount'.

3.2 Business Transaction Graphs

According to our analytical foundation, information about a case can be retrieved from the transactional data created during its execution, from the referenced master data and from all relationships in between. A graph spanned by those data objects and relationships is a natural representation of a case. We denote such graphs by *business transaction graphs*.

Figure 2 shows an example collection of four business transaction graphs in EPGM representation, where single graphs are bounded by rectangles. Vertices and graphs show type labels (e.g., `Employee` and `BusinessTransactionGraph`) as well as properties (e.g., `status:closed` and `soCount:1`). For simplification, we show only selected vertex properties and leave out properties and types of edges. Graph properties represent aggregated measures of a graph, where the text color indicates the connection to their base data. For example, the property `soCount` displays the number of `SalesOrder` vertices contained in a graph.

The type labels of vertices represent class names. A gray background of vertices indicates them to be instances of master data classes and a white background to be transactional data. The distinction of classes in master and transactional data happens before data integration on the metadata level (see [1]).

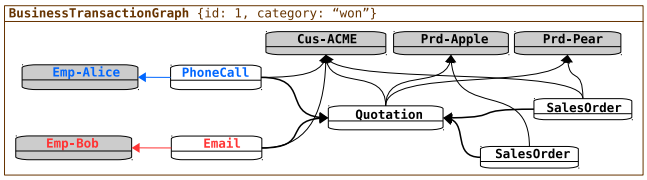

Figure 3: business transaction graph (id=1) after transformation for pattern mining.

The property `sid` is shown for all vertices and marks their mandatory *source identifier*. Source identifiers are globally unique, required for data integration (see [1]) and provide provenance information. They are often meaningful to domain experts as they include source-specific business identifiers (e.g., customer numbers). They further act as the finest granularity of dimensional hierarchies (e.g., customer number ← customer city ← customer country). For better understanding, we use natural language to represent business identifiers in our example graphs (e.g., Alice, ACME or Apple).

In Figure 2, every graph represents a case of a trade process. For simplification, they cover only the initial steps from quotation to order. In particular, they show quoted products, the related customer, sales activities (emails and phone calls), the employees executing these activities and sales orders based on the shown quotations. It should be mentioned, that graphs covering a full trade process are much larger and more heterogeneous as they also include other subprocesses such as purchasing, logistics or accounting. Further on, although not shown in our example, also edge types and properties provide useful information, for example, the amount of quoted products.

In [1], we proposed an algorithm to extract business transaction graphs from the integrated instance graph. Our algorithm is based on the observation, that transactional data objects of the same case and their relationships form connected components. Relationships between transactional data objects (represented by bold edges in Figure 2) exist in source systems to provide traceability of interrelated data. For example, every sales order is related to a prior quotation. Such relationships also exist across data sources. For example, sales phone calls can be logged in a telemarketing system and every log entry provides a reference to a specific quotation stored in a dedicated system for customer relationship management. The proposed algorithm identifies such components and extends them by referenced master data vertices and the connecting edges. In the result, one MD vertex may be part of multiple business transaction graphs. For example, in Figure 2, all graphs contain the same customer. However, shared master data is no hint for a common case. For example, two sales orders containing the same products may have been processed completely independently.

4 Graph Collections Analytics

Potential analytics with graph collections are manifold. For example, one can compare metrics of different communities in a social network or mine frequent substructures of chemical compounds. In connection with the BIIIG approach, graph collections enable analyzing the k-neighborhoods of customers or evaluating business process executions using the abstraction of business transaction graphs.

In this section, we describe an example analytical scenario and propose an approach to identify correlations between aggregated graph measures and the occurrence of certain graph patterns. We also show, how such graph collection analytics can be implemented on top of the Gradoop framework.

4.1 Analytical Scenario

In our example scenario, an analyst wants to identify reasons for won and lost quotations in sales cases. Therefore, closed cases should be evaluated to give recommendations for sales activities in open cases. A case is considered to be *closed*, if there are no open quotations. In Figure 2, the first three graphs represent closed cases and the fourth graph an open one. Further on, a closed case is considered to be *won*, if at least one sales order was created, and to be *lost*, otherwise. Consequently, the first two graphs of Figure 2 represent won cases and the third graph a lost one.

Reasons for won and lost cases should be represented by meaningful patterns like *'a phone call made by Alice'*. However, patterns should not be limited to such simple statements, but also reflect compositions like *'Alice made a phone call and Bob sent an email regarding the same quotation'*. Additionally, patterns must not be trivial. For example, if a pattern occurs frequently in won cases but also in lost cases, it cannot be considered to be characteristic for either of the outcome categories.

4.2 Graph Pattern Mining

From our analytical scenario, we can derive the following generalized problem definition: Let $\mathcal{G} = \langle G_1, .., G_n \rangle$ be a collection of business transaction graphs, $C = \{c_1, .., c_m\}$ be a set of categories (e.g., $C = \{won, lost\}$) and $\zeta : \mathcal{G} \rightarrow C$ be a mapping associating every graph to a category, then we want to identify a graph collection representing patterns $\mathcal{P} = \langle P_1, .., P_k \rangle$, where each pattern is considered to be characteristic for a category, formally another mapping $\pi : \mathcal{P} \rightarrow C$.

To identify $\mathcal{P}$ and π, we use an approach based on *frequent subgraph mining* (FSM), in particular the *transactional setting*, which aims at identifying subgraphs *supported* by a minimum number of graphs (*threshold*) in a collection [15]. However, our problem definition differs from typical FSM scenarios. In the following, we discuss the differences and our adaption.

First, existing FSM algorithms are based on label equality [15] but we also need to consider selected dimensional attributes (e.g., source identifiers of master data) to achieve meaningful patterns. To represent relevant features of business transaction graphs within vertex and edge labels, we apply a transformation (see Section 2.2) to generate our search space $\mathcal{G}'$ and consider *patterns* to be subgraphs supported by image graphs of the search space. More precisely, let G be a business transaction graph and let G' be its transformed image, then G and G' belong to the same measure category $\zeta(G) = \zeta(G')$ and every image subgraph $P \subseteq G'$ is considered to be a *pattern* of the original graph G.

In Figure 2, we highlighted two example patterns. The pattern in blue color represents *'a phone call made by Alice'* and the one in red color *'an email sent by Bob'*. Figure 3 shows the first graph of Figure 2 after the transformation. The transformation removes all vertex properties and replaces type labels by source identifiers for master data vertices. One can see, that pattern information previously embedded in properties is now retrievable by mining of vertex labels.

Our problem further differs from typical FSM scenarios, as we require patterns not just to be frequent but to be characteristic for a measure category. For example, the blue pattern is interesting, as it occurs in all of the won cases but not in the lost one. By contrast, the red pattern occurs in all graphs of both categories. Therefore, we use an *interestingness measure* comparing the frequency of a subgraph in different categories. We define the interestingness of a pattern P in a category $c \in C$ as follows:

$$interestingness(P, c) \mapsto \frac{relSupport(P, c)}{avgRelSupport(P, C)}$$

The function evaluates the relative support of a pattern within a category in relation to its average relative support in all categories. Values ≥ 1 imply an exceptional frequency in the corresponding category. Based on this measure, the analyst sets an *interestingness threshold* to prune patterns by minimum interestingness.

However, we further need to avoid unnecessary computation, as FSM includes the subgraph isomorphism problem, which is known to be NP-complete. Therefore, we additionally require a *candidate threshold* defining the minimum relative support of a pattern inside a category to be reported as a candidate for characterization. Setting this threshold is a trade-off between result completeness and computing time. A high value leads to fewer candidates and in consequence a low value to more results. Choosing the right value also depends on the analytical goal. For example, the identification of rare interesting patterns (e.g., fraud detection) requires a nearly complete result but the identification of patterns occurring systematically (e.g., to extract process models) may not require the computation of outliers.

4.3 Implementation as Gradoop Workflow

The extended property graph model used by Gradoop well suits the abstraction of business transaction graphs, as shown in Figure 2. We just require two additional properties with reserved keys mandatory for all vertices, in particular `superType` to express the association to either master or transactional data and `sid` to mark the source identifier.

Business intelligence often combines different techniques. For example, to answer our analytical question, we need to identify closed cases, categorize them by either lost or won cases and identify characteristic patterns for each group. In Script 1, we illustrate how such workflows can be implemented using Gradoop. In future releases, Gradoop will provide a domain specific language with a syntax similar to the shown pseudocode. Currently, such workflows are expressed using the Java programming language, where operators provide a declarative API. The aim of this example is not only the workflow itself but also the general analytical value of graph collections and the composition of workflows including collection operators.

The input of our workflow is a BIIIG integrated instance graph (`iig`). In line 4, we embed a custom algorithm [1] to extract the collection of business transaction graphs (`btgs`) using the call operator. After that, we mark closed cases using aggregation. Line 6 shows the definition of an according aggregate function. In the function body, the vertices of the input graph `g.V` are filtered to those of type `Quotation` providing a property with key `status` and value `open`. The function will return true, if the filtered vertex set is empty, i.e., the graph contains no such vertex. In line 10, the function is executed for all graphs using the apply operator and the aggregation result is stored into a graph property with key `isClosed`. This property is used within a predicate in line 13 to select graphs representing closed cases.

In the second step, we categorize closed cases. The categorization of business transaction graphs requires a *categorical property*, which associates a graph to a single value within a finite range. Such ranges may reflect natural ranges like customer countries or describe intervals of numerical measures, for example to categorize the financial result of a case into profit and loss. In line 15 to 19, we apply a further aggregate function to count the number of sales orders per case. Based on that measure, we categorize cases as either *won* or *lost* within a graph transformation function in line 21.

To form a search space for meaningful patterns, we also define transformation functions for vertices in line 26 and edges in line 31. The vertex function keeps only the label for transactional vertices but uses the source identifier as master data vertex label and the edge function keeps only edge labels. The functions are applied to all graphs in line 33 to generate the search space for our custom pattern mining algorithm, which is called under specific parameters in line 36.

```
1  Input:  LogicalGraph iig,
2  Output: GraphCollection

4  btgs = iig.callForCollection(:BTGIsolation, {})

6  isClosedFunc = (g => g.V.filter(
7    v => v.label == "Quotation" &&
8    v["status"] == "open").isEmpty())

10 btgs = btgs.apply(
11   g => g.aggregate("isClosed", isClosedFunc))

13 btgs = btgs.select(g => g["isClosed"])

15 soCountFunc = (g => g.V.filter(
16   v => v.label == "SalesOrder").size())

18 btgs = btgs.apply(
19   g => g.aggregate("soCount", soCountFunc))

21 gFunc = (gIn, gOut =>
22   if gIn["soCount"] > 0
23   then gOut["category"]="won"
24   else gOut["category"]="lost")

26 vFunc = (vIn, vOut =>
27   if vIn["superType"] == "MasterData"
28   then vOut.label = vIn["sid"]
29   else vOut.label = vIn.label)

31 eFunc = (eIn, eOut => eOut.label = eIn.label)

33 btgs = btgs.apply(
34   g => g.transform(gFunc, vFunc, eFunc))

36 interestingPatterns = btgs
37   .callForCollection(
38     :CategoryCharacteristicSubgraphs, {
39       categoryKey: "isClosed",
40       candidateThreshold: 0.5,
41       interestingnessThreshold: 1.2 })

43 return interestingPatterns
```

Script 1: BIIIG workflow on Gradoop.

The workflow results in a new graph collection, where every graph represents a characteristic pattern including a property reflecting its category association. This collection can be presented to the analyst, for example using an appropriate graph visualization.

The shown workflow is currently executable, except the mining algorithm. The implementation progress can be traced on GitHub[8].

5 Related Work

To the best of our knowledge, there is no other approach utilizing graph collections similarly to BIIIG.

Recent work on graph OLAP focuses on single graphs. Existing approaches already use distributed computation [12], provide support for semantically rich graph models [13] and adopt the concepts of facts, measures and dimensions to graphs [14].

[8] https://github.com/dbs-leipzig/gradoop/tree/master/gradoop-examples/src/main/java/org/gradoop/examples/biiig/CategoryCharacteristicPatterns.java

A core technique of our approach, *frequent subgraph mining* (FSM), is a well studied problem for single machine scenarios [15] and recent work focuses on the distribution of FSM algorithms using MapReduce [16, 17].

The evaluation of business process executions is also subject to the research field of *process mining* (PM) [18]. PM relies on the existence of well-structured *event logs*. The authors of [19] provide a semi-automated approach to extract event logs from information systems. However, while PM solely focuses on the process execution itself, e.g., model conformity, we proposed to evaluate business measures in the context of relationship patterns including data types and dimensional attributes.

6 Conclusions and Future Work

In this work, we demonstrated the analytical value of graph collections for business intelligence, a domain that is until now dominated by the use of relational and multidimensional models. We compared recent approaches to graph data management and analytics with regard to complex analytical workflows and their suitability in Big Data scenarios. We observed that the Gradoop framework is the most fitting solution as it provides a semantically rich data model including the support of graph collections and respective operators. Further on, Gradoop allows the declaration of operator workflows and uses distributed processing to provide scalability for very large data volumes.

In future publications we will provide details about distributed graph pattern mining on Gradoop in general and in the context of the BIIIG approach. We plan to extend our mining technique to consider not only single dimensional properties of master data but also include dimension hierarchies. We also want to investigate approaches to predictive graph analytics, for example, to forecast case development and to enable what-if analyses. To examine such approaches, we aim at comprehensive evaluations using real-world data and involving domain experts.

7 Acknowledgments

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

Literature

[1] A. Petermann, M. Junghanns, R. Müller and E. Rahm. *BIIIG: Enabling Business Intelligence with Integrated Instance graphs.*. ICDE Workshops, pp. 4-11, 2014.

[2] A. Petermann, M. Junghanns, R. Müller and E. Rahm. *Graph-based Data Integration and Business Intelligence with BIIIG*. PVLDB 7(13), pp. 1577-1580, 2014.

[3] M. Junghanns, A. Petermann, K. Gómez and E. Rahm. *GRADOOP: Scalable Graph Data Management and Analytics with Hadoop*. arXiv:1506.00548, 2015.

[4] M. Junghanns, A. Petermann, N. Teichmann, K. Gomez and E. Rahm. *Analyzing Extended Property Graphs with Apache Flink*. Workshop on Network Data Analytics in conjunction with ACM SIGMOD/PODS, 2016.

[5] M. A. Rodriguez and P. Neubauer. *Constructions from dots and lines*. Bulletin of the American Society for Information Science and Technology 36(6), pp. 35-41, 2010.

[6] R. Angles. *A comparison of current graph database models*. ICDE Workshops, pp. 171-177, 2012.

[7] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser and G. Czajkowski. *Pregel: a system for large-scale graph processing*. SIGMOD, pp. 135-146, 2010.

[8] R. Xin et al. *GraphX: A Resilient Distributed Graph System on Spark* Int. Workshop on Graph Data Management Experiences and Systems, pp. 2-7, 2013.

[9] A. Alexandrov et al. *The Stratosphere platform for big data analytics*. VLDB Journal, 23(6), pp. 939-964, 2014.

[10] Z. Kaoudi and I. Manolescu. *RDF in the Clouds: A Survey*. VLDB Journal, 24(01), pp. 67-91, 2015.

[11] J. Futrelle. *Harvesting RDF triples.*. Provenance and Annotation of Data, 24(01), pp. 64-72, 2006.

[12] Z. Wang et al. *Pagrol: PArallel GRaph OLap over large-scale attributed graphs*. ICDE, pp. 496-507, 2014.

[13] A. Ghrab et al. *A Framework for Building OLAP Cubes on Graphs*. Advances in Databases and Information Systems, pp. 92-105, 2015.

[14] M. Rudolf, H. Voigt, C. Bornhövd and W. Lehner. *SynopSys: Foundations for Multidimensional Graph Analytics*. Enabling Real-Time Business Int., pp. 159-166, 2015.

[15] C. Jiang, F. Coenen and M. Zito. *A survey of frequent subgraph mining algorithms*. The Knowledge Engineering Review, 28(01), pp. 75-105, 2013.

[16] W. Lu, G. Chen, A.K. Tung and F. Zhao. *Efficiently extracting frequent subgraphs using MapReduce*. IEEE Int. Conf. on Big Data, pp. 639-647, 2013.

[17] W. Lin, X. Xiao and G. Ghinita. *Large-scale frequent subgraph mining in MapReduce*. ICDE, pp. 844-855, 2014.

[18] W. van der Aalst et al. *Change your history: Learning from event logs to improve processes*. IEEE Int. Conf. on Comp. Supported Coop. Work in Design, pp. 7-12, 2015.

[19] X. Lu, M. Nagelkerke, D. van de Wiel and D. Fahland, *Discovering Interacting Artifacts from ERP Systems*. IEEE Trans. on Services Comput., 8(06), pp.861-873, 2015.

André Petermann studied Information Technology at the University of the West of Scotland (BSc 2009) and Multimedia Technology at the Leipzig University of Applied Sciences (Dipl.-Ing.(FH) 2011). He gained in-depth experience in design and development of business intelligence solutions with COMPAREX AG (2009-2012). In 2013, Petermann became a researcher in the field of graph analytics for business intelligence at Leipzig University Database Research Group.

Address: Universität Leipzig, Institut für Informatik, D-04109 Leipzig, E-Mail: petermann@informatik.uni-leipzig.de

Martin Junghanns studied Computer Science at the Leipzig University (MSc 2014). While working for the NoSQL database vendor sones GmbH (2009-2011) and SAP (2013-2014) he gained in-depth knowledge about database systems, software development and graph data management. Since 2014, Junghanns is working as a researcher in the field of distributed graph analytics and management at Leipzig University Database Research Group. He is an active contributor in Big Data related Open Source projects.

Address: Universität Leipzig, Institut für Informatik, D-04109 Leipzig, E-Mail: junghanns@informatik.uni-leipzig.de