Graph + Hadoop = Gradoop

Verteilte Graphanalyse mit Gradoop

Martin Junghanns, André Petermann

Graphen eignen sich für die intuitive Abbildung und Analyse komplexer Beziehungen zwischen beliebigen Datenobjekten. Bekannte Anwendungen reichen von Empfehlungssystemen in sozialen Netzwerken über Wegoptimierungen in Logistiknetzwerken bis hin zur Mustersuche in komplexen organischen Verbindungen. Aber auch in Unternehmensdaten ermöglichen Graphen neuartige Analysen, die mit bisherigen Techniken nicht, oder nur mit hohem Aufwand, möglich waren. Das Forschungsprojekt Gradoop unterstützt die domänenunabhängige Analyse von Graphdaten und setzt dabei auf moderne Big-Data-Technologie.

In den letzten Jahren wurde eine Vielzahl verschiedener Systeme für die Verwaltung und Verarbeitung von Graphen vorgestellt. Während sich Graphdatenbanken, wie zum Beispiel Neo4j, auf operationalen Betrieb und hohe Benutzerfreundlichkeit durch deklarative Anfragesprachen, Tooling und Visualisierung konzentrieren, eignen sich Graph-Processing-Systeme, wie etwa Apache Giraph, für die skalierbare Ausführung analytischer Graphalgorithmen auf umfangreichen Graphen.

Das an der Universität Leipzig entwickelte Gradoop-Projekt fokussiert sich auf die Kombination hoher Benutzerfreundlichkeit und Skalierbarkeit. Es bietet dem Nutzer ein ausdrucksstarkes Graphdatenmodell in Verbindung mit analytischen Operatoren und Algorithmen, die sich zu komplexen analytischen Programmen kombinieren lassen. Gradoop ist eine Third-Party-Bibliothek (GPLv3) auf dem verteilten Dataflow-System Apache Flink und ermöglicht hierdurch die Kombination von Graphanalysen mit bereits vorhandenen Bibliotheken für Machine Learning und SQL.

Architektur

Gradoop (*Gra*ph Analytics on Ha*doop*) verwendet die folgenden im Hadoop-Ökosystem angebotenen Softwareprojekte: das verteilte Dataflow-System *Apache Flink*, die verteilte NoSQL-Datenbank *Apache HBase* und das verteilte Dateisystem *Apache HDFS*. Die Verarbeitung von Graphen erfolgt batch-orientiert, das heißt, Daten werden aus einer Datenquelle gelesen, durch ein analytisches Programm verarbeitet und das Ergebnis wird in eine Datensenke geschrieben. Das Einlesen, Verarbeiten und Schreiben erfolgt verteilt auf den Maschinen eines Rechnerclusters.

Abbildung 1 zeigt eine vereinfachte Darstellung der Gradoop-Architektur auf Apache Flink, einem Dataflow-System, welches die deklarative Definition und verteilte Ausführung analytischer Programme auf Stream- und Batch-Daten anbietet. Gradoop nutzt die Batch-Programmierschnittstelle, deren grundlegende Abstraktion aus *DataSets* und *Transformationen* besteht.

Ein DataSet ist eine unveränderliche Menge beliebiger Datenobjekte, zum Beispiel Pojos oder spezielle Tupel-Typen; Transformationen beschreiben die Konstruktion von DataSets aus existierenden DataSets oder Datenquellen. Anwendungslogik wird in benutzerdefinierten Funktionen (User-Defined Function, UDF) gekapselt und als Parameter den Transformationen übergeben. Diese wenden die UDF auf die Elemente der Eingabemenge an und produzieren eine Ausgabemenge.

Prominente Transformationen sind `map` und `reduce`. Während die `map`-Transformation mittels UDF jedem Element der Eingabemenge genau ein Element in der Ausgabemenge zuordnet, kann die `reduce`-Transformation genutzt werden, um alle Elemente der Eingabemenge auf genau ein Element abzubilden. Zusätzlich bietet Apache Flink mit zum Beispiel `join`, `groupBy`, `project` und `filter` weitere Transformationen an, die vor allem aus dem relationalen Bereich bekannt sind. Der Datenfluss innerhalb eines Flink-Programms lässt sich durch die Kombinationen mehrerer Transformationen beschreiben. Bei Ausführung wird das Programm von Flink übersetzt, optimiert und je nach Ausführungsumgebung lokal oder verteilt ausgeführt.

Den Kern der Gradoop-Bibliothek bilden das Extended Property Graph Model (EPGM) und eine Menge von Graphoperatoren. Das EPGM ist ein ausdrucksstarkes Graphdatenmodell, welches sich an Datenmodellen von Graphdatenbanken orientiert. Graphoperatoren dienen der Analyse und Verarbeitung von EPGM-Graphen. Gradoop bildet das Datenmodell auf DataSets und die Graphoperatoren auf die von Flink bereitgestellten Transformationen ab und stellt dem Anwender eine Java-Programmierschnittstelle zur Formulierung analytischer Programme zur Verfügung. Vorhandene Optionen für Datenquellen und -senken sind Apache HBase (GraphStore) sowie im HDFS gespeicherte JSON-Dateien.

Darüber hinaus bietet Gradoop die Möglichkeit, beliebige Graphformate automatisiert in das eigene Datenmodell zu transformieren. Beispiele finden Sie auf [GitHub]. Neben der inhärenten horizontalen

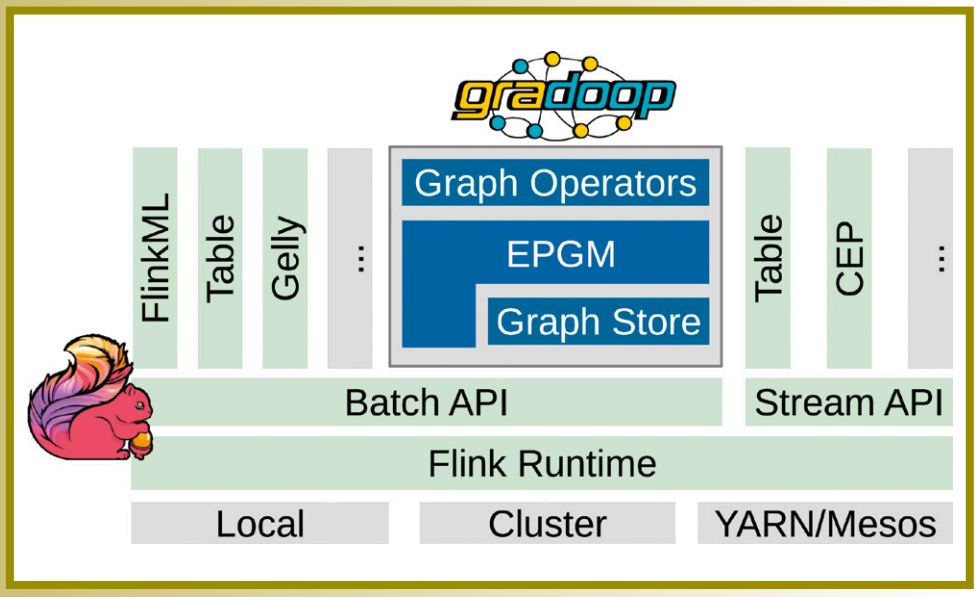

Abb. 1: Gradoop-Bibliothek auf Apache Flink

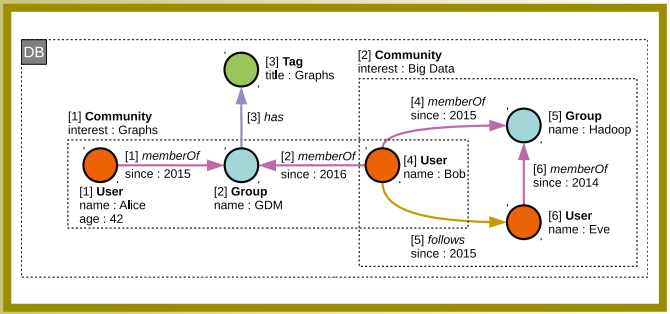

Abb. 2: EPGM-Beispielgraph: ein soziales Netzwerk mit zwei überlappenden logischen Graphen

Skalierbarkeit liegt ein wesentlicher Vorteil dieser Architektur in der Möglichkeit zur Kombination von Gradoop mit den bereits vorhandenen Flink-Bibliotheken, zum Beispiel für Machine Learning (FlinkML), Graph Processing (Gelly) und SQL (Table). Nachfolgend werden wir genauer auf das EPGM und seine Operatoren eingehen.

Extended Property Graph Model

Datenobjekte werden in Graphen durch Knoten und Beziehungen zwischen ihnen durch Kanten dargestellt. Abbildung 2 zeigt dies am Beispiel eines sozialen Netzwerkes. Während hier Knoten Objekte wie Nutzer (`User`), Gruppen (`Group`) oder Interessen (`Tag`) repräsentieren, beschreiben Kanten die Beziehungen zwischen ihnen, zum Beispiel Anhängerschaft (`follows`) oder die Mitgliedschaft in einem Forum (`memberOf`).

Typ-Label ermöglichen es, Knoten und Kanten mit einer Semantik zu versehen (z. B. `User` oder `follows`). Kanten besitzen zusätzlich eine Richtung, welche die Beziehung zwischen den Entitäten weiter konkretisiert (z. B. wer folgt wem: `User-follows->User`). Attribute werden im Datenmodell durch Schlüssel-Wert-Paare an Knoten und Kanten abgebildet. Der Nutzer Alice (Knoten 1) besitzt im Beispiel die Attribute `name : Alice` und `age : 42`, wobei `name` und `age` die Attributschlüssel und `Alice` beziehungsweise `42` die zugeordneten Attributwerte sind. Es sei darauf hingewiesen, dass Knoten und Kanten keinem global definierten Schema folgen, sondern selbstbeschreibend sind. Ein Typ-Label setzt keine festgelegte Menge von Attributen voraus.

Das bisher beschriebene Datenmodell wird in der Literatur als Property Graph Model bezeichnet, es bildet die Grundlage vieler Graphdatenbanksysteme. Gradoop nutzt ebenfalls dieses Modell, erweitert es jedoch unter anderem um das Konzept der logischen Graphen. Diese ermöglichen es, Teilgraphen einer Datenbank explizit zu beschreiben und sie als Eingabe für Graphoperatoren zu verwenden. Das Beispiel enthält zwei logische Graphen, die jeweils eine Interessengruppe (`Community`) umfassen: Während Graph 1 alle Nutzer und Gruppen zum Thema Graphdatenmanagement (GDM) enthält, umfasst Graph 2 alle Nutzer und Gruppen zum Thema Big Data.

Am Beispiel wird deutlich, dass sich logische Graphen überlappen können, Knoten und Kanten sind in diesem Fall in mehr als einem logischen Graphen enthalten. Logische Graphen besitzen ebenfalls ein Typ-Label und beliebig viele Attribute, was es ermöglicht, Graphen mit Informationen anzureichern.

Logische Graphen werden entweder explizit in der Anwendung definiert oder sind das Ergebnis von Graphoperatoren. Letztere verwenden logische Graphen als Ein- und Ausgabe, sie ermöglichen hierdurch die Verkettung mehrerer Operatoren zu analytischen Programmen. Gradoop unterscheidet zwischen unären und binären Graphoperatoren beziehungsweise Graphmengenoperatoren, welche ein oder zwei logische Graphen beziehungsweise Graphmengen als Eingabe erhalten. Das Ergebnis eines Operators ist entweder ein neuer logischer Graph oder eine Graphmenge. Nachfolgend werden wir eine Auswahl verfügbarer Operatoren vorstellen.

Graphoperatoren

Bevor Graphen analysiert werden können, müssen sie von einer Datenquelle eingelesen werden. Im nachfolgenden Beispiel gehen wir von einem logischen Graphen aus, dessen Knoten und Kanten in JSON-Dateien im HDFS abgelegt sind:

```
// Knoten inkl. Typ-Label und Attribute
String vertices = "hdfs:///vertices.json";
// Kanten inkl. Typ-Label und Attribute
String edges = "hdfs:///edges.json";
// Erzeugen einer neuen Datenquelle
DataSource dataSource = new JSONDataSource(vertices, edges, config);
// Einlesen des logischen Graphen
LogicalGraph graph = dataSource.getLogicalGraph();
```

Die Variable `graph` referenziert einen logischen Graphen und ist somit Ausgangspunkt für die nachfolgende Analyse. Alternativ kann direkt eine Graphmenge eingelesen werden.

An dieser Stelle sei darauf hingewiesen, dass die Programmausführung verzögert (lazy) erfolgt und explizit getriggert werden muss. Dies erfolgt entweder durch den Aufruf der dedizierten Flink-Methode `ExecutionEnvironment.execute()` oder spezieller DataSet-Methoden wie zum Beispiel `DataSet.count()` oder `DataSet.collect()`. Nachfolgend wollen wir auf einige der verfügbaren Graphoperatoren genauer eingehen.

Aggregation

In analytischen Anwendungen wird die Aggregation allgemein eingesetzt, um eine Menge von Werten auf einen einzelnen Wert mit signifikanter Bedeutung abzubilden. Gradoop unterstützt Aggregation auf der Ebene logischer Graphen und stellt eine Auswahl grundlegender Aggregatfunktionen zur Verfügung. Alternativ kann der Anwender eigene Funktionen definieren und hat dabei Zugriff auf den kompletten logischen Graphen, das heißt alle Knoten und Kanten inklusive ihrer Typ-Label und Attribute. Nach der Ausführung steht das Ergebnis der Aggregatfunktion als neues Attribut am logischen Graphen zur Verfügung.

Im folgenden Beispiel:

```
graph = graph
  .aggregate("vertexCount", new VertexCount())
  .aggregate("minAge", new MinVertexProperty("age"));
```

berechnen wir zwei Aggregate für den Eingabegraphen. Der erste Aufruf berechnet die Anzahl der Knoten innerhalb des logischen Graphen und speichert das Ergebnis als neues Graphattribut unter Verwendung des Attributschlüssels `vertexCount`. Der folgende Aufruf berechnet das Minimum aller Attributwerte, die dem Attributschlüssel `age` zugeordnet sind, und speichert das Ergebnis ebenfalls als neues Graphattribut.

Subgraph

Nicht immer ist die Definition einer eigenen Aggregatfunktion notwendig. Soll zum Beispiel die Anzahl der Nutzer und deren

Beziehungen im Graphen bestimmt werden, lässt sich die Aggregation mit dem Subgraph-Operator verbinden:

```
LogicalGraph subgraph = graph
  .subgraph(new FilterFunction<Vertex>() {
    @Override
    public boolean filter(Vertex v) {
      return v.getLabel().equals("User");
    }
  }, new FilterFunction<Edge>() {
    @Override
    public boolean filter(Edge e) {
      return e.getLabel().equals("follows");
    }
  })
  .aggregate("users", new VertexCount())
  .aggregate("friendships", new EdgeCount());
```

In diesem Beispiel reduzieren wir den logischen Graphen zunächst auf einen Teilgraphen (Subgraph), der ausschließlich Knoten mit dem Typ-Label **`User`** und Kanten mit dem Typ-Label **`follows`** beinhaltet. Das Ergebnis ist ein neuer logischer Graph, den wir anschließend aggregieren, um seine Knoten- und Kantenanzahl zu ermitteln. Alternativ lassen sich Knoten- oder Kantenfilter auch exklusiv definieren, um beispielsweise Beziehungen zwischen Nutzern unabhängig vom Typ-Label zu extrahieren.

Transformation

Dieser unäre Operator bietet dem Anwender die Möglichkeit, Typ-Labels und Attribute des Graphen und seiner Knoten und Kanten zu manipulieren, ohne dabei die Graphtopologie zu verändern. Dies ist zum Beispiel bei der Datenintegration oder beim Preprocessing für nachfolgende Operatoren sinnvoll.

Folgendes Listing transformiert die Knoten des Subgraphen aus dem vorhergehenden Beispiel:

```
LogicalGraph transformed = subgraph
 .transformVertices(new TransformationFunction<Vertex>() {
  @Override
  public Vertex execute(Vertex current, Vertex transformed) {
  transformed.setLabel(„Person“);
   // berechne das Geburtsjahr
   int yob = CURRENT_YEAR - current.getPropertyValue("age").getInt();
   // bestimme Dekade und erzeuge neues Attribut
   transformed.setProperty("decade", yob - yob % 10);
   return transformed;
  }
 });
```

Die Transformationsfunktion erhält den aktuellen Knoten (**`current`**) sowie einen neuen Knoten (**`transformed`**) als Eingabe. Letzterer übernimmt die Id und das Typ-Label des aktuellen Knotens, jedoch nicht dessen Attribute. Der Anwender kann nun entscheiden, welche Attribute er übernehmen will (white listing). Alternativ kann auch der aktuelle Knoten manipuliert und zurückgegeben werden. Im Beispiel setzen wir ein neues Typ-Label und berechnen ausgehend vom **`age`**-Attribut des aktuellen Knotens die Dekade, in welcher die Person geboren wurde, und hinterlegen diese als neues Attribut.

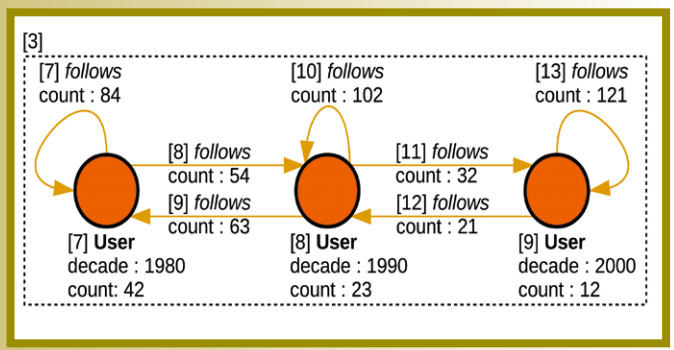

Abb. 3: Mögliches Ergebnis des Grouping-Operators

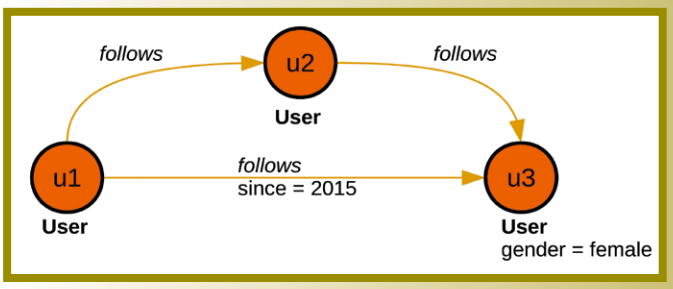

Abb. 4: Mustergraph als Eingabe für den Pattern-Matching-Operator

Grouping

Gradoop ermöglicht die strukturelle Gruppierung logischer Graphen basierend auf Typ-Labels und Attributen. Nachdem wir den Graphen transformiert haben, sollen die Nutzer anhand ihrer Dekade gruppiert und die Beziehungen zwischen den entstehenden Gruppen berechnet werden.

Abbildung 3 zeigt ein mögliches Ergebnis dieser Operation. Jeder Knoten im resultierenden Graphen repräsentiert alle Knoten mit identischem Attributwert im Eingabegraphen. Kanten zwischen Knoten im Eingabegraphen werden zusammen mit ihren jeweiligen Start- und Zielknoten gruppiert, können aber auch durch Angabe von Attributschlüsseln weiter unterteilt werden. Ebenfalls optional ist die Angabe von Aggregatfunktionen, wie **`count`** oder **`min`**, welche auf die erzeugten Gruppen angewendet werden.

Folgendes Listing zeigt die Verwendung des Grouping-Operators, welchen wir zunächst mittels Builder initialisieren und dann auf dem Eingabegraphen aufrufen:

```
Grouping operator = new Grouping.GroupingBuilder()
  .useVertexLabel(true)
  .useEdgeLabel(true)
  .addVertexGroupingKey("decade")
  .addVertexAggregator(new CountAggregator("count"))
  .addEdgeAggregator(new CountAggregator("count"))
  .build();
LogicalGraph grouped = graph.callForGraph(operator);
```

Pattern Matching

Dieser Operator ermöglicht die Suche nach beliebigen, benutzerdefinierten Mustern innerhalb des Eingabegraphen. Im Gegensatz zu den bisher vorgestellten Operatoren ist das Ergebnis kein einzelner logischer Graph, sondern eine Menge logischer Graphen, welche jeweils einen Teilgraphen des Eingabegraphen darstellen und mit dem Muster übereinstimmen.

Für die Definition von Mustern orientiert sich Gradoop an der Neo4j-Anfragesprache Cypher. In Cypher werden Mustergraphen durch Pfade ausgedrückt, die jeweils visuell, sogenannte ASCII-Art, beschrieben werden. Sollen zum Beispiel alle Dreiecksbeziehungen zwischen Nutzern gefunden werden (s. Abb. 4), können wir dies durch drei Pfade mit jeweils einer Kante formulieren:

```
GraphCollection triangles = graph.match(
  "(u1:User)-[:follows]->(u2:User);" +
```

```
    "(u2)-[:follows]->(u3:User {gender=female});" +
    "(u3)<-[:follows {since=2015}]-(u1)");
```

Zur mehrfachen Referenzierung auf Knoten und Kanten im Mustergraphen lassen sich Variablen (z. B. **`u1`**) vergeben. Ist dies der Fall, kann in nachfolgenden EPGM-Operatoren auf Knoten und Kanten mittels Variablen zugegriffen werden. Ebenfalls optional ist die Einschränkung der Knoten- und Kantenkandidaten durch die Angabe von Typ-Labeln (z. B. **`:User`**) oder Attributen (z. B. **`{since=2015}`**). Kantenrichtungen werden durch die Verwendung von Pfeilen (z. B. **`(a)<-[:follows]-(b)`**) festgelegt.

Mengenoperatoren

Eine wesentliche Eigenschaft des EPGM ist die Unterstützung von Graphmengen zur Analyse vieler, sich möglicherweise überlappender Graphen. Hierfür stellt Gradoop mehrere Operatoren zur Verfügung, welche sich zum Teil an relationalen Operatoren orientieren.

Apply

Der Apply-Operator ist ein Hilfsoperator, der es ermöglicht, unäre Operatoren, wie zum Beispiel Aggregation oder Subgraph, auf jedes Element einer Graphmenge anzuwenden. Folgendes Listing zeigt die Kombination von Apply und Aggregation zur Ermittlung der Knotenanzahl für jeden logischen Graphen:

```
collection = collection
  .apply(new ApplyAggregation("vertexCount", new VertexCount()));
```

Anschließend besitzt jeder Graph der Eingabemenge ein neues Attribut **`vertexCount`**, welches das Ergebnis der Aggregatfunktion speichert.

Selection

Nachdem wir für jeden Graphen einen Aggregatwert berechnet haben, möchten wir die Menge auf Grundlage dieses Wertes filtern. Hierfür steht der Selection-Operator zur Verfügung. Dieser ermöglicht es, unter Angabe einer Prädikatfunktion, nur die Graphen in die Ergebnismenge zu übernehmen, für welche das Prädikat erfüllt ist. Wollen wir beispielsweise nur die Graphen weiter betrachten, die mehr als zehn Knoten beinhalten, so lässt sich dies wie folgt ausdrücken:

```
GraphCollection filtered = collection
  .select(new FilterFunction<GraphHead>() {
    @Override
    public boolean filter(GraphHead g) {
      return g.getPropertyValue("vertexCount").getLong() > 10;
    }
  });
```

Die Filterfunktion hat Zugriff auf Typ-Label und Attribute der Graphen, der Anwender kann davon ausgehend beliebige Bedingungen formulieren. Die Ergebnismenge enthält nur die Graphen, für welche die Funktion **`true`** zurückliefert.

Bevor wir die Einbindung externer Graphalgorithmen diskutieren, soll kurz gezeigt werden, wie logische Graphen und Graphmengen in eine Datensenke geschrieben werden. Wie eingangs erwähnt, stellt Gradoop verschiedene Implementierungen für Datenquellen und -senken bereit. Nachfolgend schreiben wir eine Graphmenge in JSON-Dateien ins HDFS:

```
String graphHeads = "hdfs:///output/graphHeads.json";
String vertices = "hdfs:///output/vertices.json";
String edges = "hdfs:///output/edges.json";
// Erzeugen einer neuen Datensenke
DataSink dataSink = new JSONDataSink(graphHeads, vertices, edges, config);
// Schreiben der Graphmenge
collection.writeTo(dataSink);
// Ausfuehren des Programms
flinkExecutionEnvironment.execute();
```

In der letzten Zeile triggern wir explizit die Ausführung des erzeugten Programms. Apache Flink übernimmt nun die Übersetzung in einen Ausführungsplan, optimiert diesen entsprechend der verwendeten Transformationen und führt das Programm in der bereitgestellten Umgebung, zum Beispiel lokal oder auf einem Cluster, aus.

Algorithmen

Neben den bereits vorgestellten Operatoren bietet Gradoop die Möglichkeit, komplexe Graphalgorithmen nahtlos in analytische Programme einzubinden. Beispiele für solche Algorithmen sind etwa das Auffinden häufiger Muster (FSM, Frequent Subgraph Mining) oder Page Rank (ein ursprünglich von Google vorgestellter Algorithmus zur Wichtung von Webseiten in Abhängigkeit der Netzwerktopologie). Neben einigen bereits in Gradoop enthaltenen Algorithmen werden auch benutzerdefinierte Algorithmen unterstützt.

Einbindung

Für den Aufruf von Graphalgorithmen bietet Gradoop den generischen Call-Operator und entsprechende Interfaces, deren Implementierungen mit dem Operator verwendet werden können. Es gibt sechs Interfaces, welche alle Kombinationen aus unärer oder binärer Eingabe von Graphen oder Graphmengen sowie deren jeweilige Ausgabe abdecken. So erfordert etwa der Aufruf **`graph.callForCollection(algorithm)`** für den Parameter **`algorithm`** eine Implementierung des Interfaces **`UnaryGraphToCollectionOperator`**, folglich einen Algorithmus, der einen Graphen in eine Graphmenge überführt.

Frequent Subgraph Mining

Ein Beispiel für die bereits in Gradoop enthaltenen Algorithmen ist Frequent Subgraph Mining (FSM). Der Algorithmus findet in einer Graphmenge alle zusammenhängenden Teilgraphen, die in einer Mindestanzahl von Graphen enthalten sind. FSM ist das Graph-Pendant zur Warenkorbanalyse (Frequent Itemsets) im relationalen Kontext. Allerdings ist für FSM nicht nur das gemeinsame Auftreten gewisser Knoten- und Kantentypen relevant, sondern auch deren Topologie. So kann die Abstraktion der Graphmenge dazu genutzt werden, einzelne Ausführungen eines Geschäftsprozesses abzubilden.

Für die Extraktion solcher Prozessgraphen aus einem umfangreichen Graphen mit integrierten Geschäftsdaten bietet Gradoop einen weiteren Algorithmus an: Business Transaction Graphs. Innerhalb der Prozessgraphen repräsentieren Knoten beliebige Geschäftsobjekte, wie Mitarbeiter, Produkte, E-Mails oder Rechnungen, und Kanten deren Beziehungen. In diesen Graphen können dann sowohl einfache Muster, wie etwa **`(E-Mail)-[gesendetVon]->(Alice)`**, als auch komplexe Beziehungsstrukturen analysiert werden. Mit der Kombination von Aggregation, Selection und FSM lassen sich dann zum Beispiel häufige Graphmuster in besonders gewinnträchtigen Prozessausführungen ermitteln:

```
GraphCollection goodProcessRuns = enterpriseGraph
  .callForCollection(new BusinessTransactionGraphs())
  .apply("result", new FinancialResult())
  .select(new FilterFunction<GraphHead>() {
    @Override
    public boolean filter(GraphHead g) {
      return g.getPropertyValue("result").getDouble() > 10000;
    }
  });
BinaryGraphToCollectionOperator fsm =
  new TransactionalFSM.TransactionalFSMBuilder()
    .minSupport(0.7)
    .minEdgeCount(3);
GraphCollection frequentPatterns = goodProcessRuns
  .callForCollection(fsm);
```

Fazit und Ausblick

Gradoop ist ein Framework zur deklarativen und horizontal skalierbaren Analyse von Graphdaten. Auch wenn der Einsatz von Graphanalysen noch nicht im Business-Intelligence-Alltag angekommen ist, lässt die steigende Anzahl verschiedener Softwareprojekte rund um Graphdaten eine wachsende Popularität von Graphanalysen zur Wissensextraktion erkennen. Während Graph-Processing-Systeme jedoch für jede analytische Fragestellung die Entwicklung eines dedizierten Programms erfordern, können in Gradoop, ähnlich einer SQL-Anfrage, selbst komplexe mehrstufige Analysen einfach deklariert werden. Darüber hinaus erlaubt Gradoop die Kombination mit weiteren Flink-Bibliotheken und grenzt sich hierdurch von spezialisierten Graphdatenbanken und Graph-Processing-Systemen ab.

Aufgrund des festgelegten Datenmodells ist es zudem nicht erforderlich, analysespezifische Datenstrukturen zu entwerfen. Und sollten die enthaltenen Operatoren einer Anforderung nicht gerecht werden, lassen sich mit den `call`-Interfaces anwendungsspezifische Komponenten ergänzen. Idealerweise können die in diesem Rahmen von Anwendern entwickelten Algorithmen als Contribution in das Open-Source-Projekt einfließen. Hierdurch erhoffen wir uns eine ständig wachsende Anzahl analytischer Bausteine, von denen wiederum andere Nutzer profitieren können.

Links

[Cypher] `https://neo4j.com/developer/cypher-query-language/`
[EPGM] `http://dbs.uni-leipzig.de/file/EPGM.pdf`
`[Flink] http://flink.apache.org/`
[GitHub] `https://git.io/vKg9I`
[Giraph] `http://giraph.apache.org/`
[Gradoop] `http://gradoop.com/`
[HBase] `https://hbase.apache.org/`
[HDFS] `http://hadoop.apache.org/`
[Neo4j] `https://neo4j.com/`

Martin Junghanns forscht an der Universität Leipzig an Methoden und Techniken zur verteilten Analyse stark vernetzter, heterogener Daten im Rahmen des Gradoop-Projektes. Seine praktische Erfahrung aus der Tätigkeit als Softwareentwickler für den Graphdatenbank-Hersteller sones GmbH und SAP bringt er in die Forschung ein. E-Mail: junghanns@informatik.uni-leipzig.de

André Petermann forscht am ScaDS Dresden/Leipzig und der Universität Leipzig an der Nutzung von Graphmodellen für Business Intelligence im Rahmen des Gradoop-Projektes. Die Motivation hierfür entstand während seiner mehrjährigen Tätigkeit als Softwareentwickler für Data Warehousing und Reporting. E-Mail: petermann@informatik.uni-leipzig.de