Universität Leipzig

Institut für Informatik
Fakultät für Mathematik und Informatik
Abteilung Datenbanken

Evaluation des EPGM auf Basis von Apache Spark

Masterarbeit

vorgelegt von:
Timo Adameit

Matrikelnummer:
3757216

Betreuer:
Prof. Dr. Erhard Rahm
Dr. Eric Peukert

Zusammenfassung

Diese Arbeit stellt mit *Gradoop-Spark* eine Implementierung des EPGM auf Basis von Apache Spark vor und untersucht den Einfluss verschiedener interner Darstellungen auf die Laufzeit und Skalierung. Gradoop-Spark basiert auf dem relationalen Spark SQL und verwendet den Catalyst-Optimierer zur Optimierung von Abfragen. Ähnlich zu relationalen Datenbanken werden die Daten in Tabellen dargestellt und mit relationalen Operatoren transformiert. Ein Graphoperator setzt sich dabei aus einer Kette von relationalen Abfragen zusammen. Es wurden zwei Schemata für die Tabellen implementiert und mit anderen Implementierungen des EPGM verglichen. Das GVE-Schema orientiert sich an der Referenzimplementierung und speichert einen Graph bzw. eine Graphmenge in nur drei Tabellen ab. Das TFL-Schema teilt die Daten nach ihrem Label auf, damit Catalyst bei Berechnungen unnötige Tabellen komplett ignorieren kann. Zusätzlich werden die Eigenschaften getrennt von den Graphelementen abgespeichert. Für jedes Schema wurden die Operatoren Subgraph, Grouping und jeweils drei Mengenoperatoren für logische Graphen und Graphmengen implementiert. Das EPGM und die Operatoren konnten mit voller Funktionalität unter Verwendung der relationalen Dataset- und DataFrame-APIs von Spark SQL implementiert werden. Dabei ist aufgefallen, dass die relationalen APIs eine geringere Ausdrucksstärke haben und teils schwer zu optimieren sind. Die Evaluation zeigt besonders für das TFL-Schema schlechte Skalierung mit kleinen Graphen. Auch bei großen Datenmengen sind die beiden Schemata meist langsamer als die auf Flink basierende Referenzimplementierung *Gradoop* und das auf Flink Table basierende *Gradoop-Table*. Das TFL-Schema zeigt in allen Operatoren bis auf Grouping bessere Laufzeiten und Skalierungen als das GVE-Schema. Es konnten verschiedene Gründe für die schlechte Performanz identifiziert und mögliche Optimierungen gefunden werden. Dabei stechen besonders die ungleiche Partitionierung beim GVE-Schema und teils doppelt berechnete Tabellen beim TFL-Schema hervor. Nach Umsetzung der vorgeschlagenen Optimierungen könnte Gradoop-Spark eine Basis für weitere Operatoren und Integrationen mit verschiedenen Frameworks von Spark bilden.

Timo Adameit
3757216

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

Diese Arbeit beschäftigt sich mit der Implementierung und Evaluation des Extended Property Graph Models (EPGM) in Apache Spark. Dabei wird der Prototyp *Gradoop-Spark* entwickelt und mit der Referenzimplementierung Gradoop verglichen.

1.1 Motivation

Viele reale Gegebenheiten weisen Graph-ähnliche Strukturen auf. Neben offensichtlichen Graphen wie Schienennetzen und sozialen Netzwerken lassen sich viele andere Daten als Graph darstellen. Das Internet, Programmcode und Banktransaktionen bestehen aus Entitäten und Beziehungen bzw. Transaktionen, die einen Graph bilden. Für die Analyse dieser Graphen wurden verschiedene Graph-Datenbanken und Analyse-Frameworks entwickelt. Mit diesen Systemen können Besonderheiten in den Graphen identifiziert und Strukturen analysiert werden. Im Big Data-Bereich treten allerdings Datenmengen auf, die zu groß für herkömmliche Systeme sind und verteilt verarbeitet werden müssen. Um diesem Problem zu begegnen, sind eine Reihe von spezialisierten verteilten Graphanalyse-Frameworks entstanden, die mit vielen Maschinen und großen Datenmengen skalieren. Diese eigenen sich meist ausschließlich für Graphalgorithmen und unterstützen keine ETL-Operationen (Extract, Transform, Load) und sonstige Transformationen. Im Big Data-Bereich sind außerdem allgemeine Frameworks entstanden, die beliebige unstrukturierte Daten verarbeiten können. Graphanalyse-Frameworks, die auf diesen allgemeinen Frameworks aufbauen, können von der Vielfältigkeit der Systeme profitieren. ETL-Operationen und das Kombinieren mehrerer Graphen oder Hinzufügen weiterer Daten zu einem Graph werden ermöglicht. Auch Integrationen mit Cloud Anbietern und spezialisierten Systemen wie Machine Learning können den Analyseablauf erleichtern.

In der Abteilung Datenbanken der Universität Leipzig wird die Graphanalyse mit verteilten Frameworks erforscht. Das Forschungsprojekt *Gradoop* [26] ist ein Graphanalyse-Framework auf Basis des verteilten Datenfluss-Frameworks Apache Flink [2]. Gradoop ist die Referenzimplementierung des EPGM-Modells, welches das Property Graph Modell um das Konzept von logischen Graphen erweitert. Eine Vielzahl von Operatoren in Gradoop ermöglichen die ausführliche Analyse realer Graphen.

Apache Spark [40] ist ein verteiltes Datenfluss-Framework mit vielen Ähnlichkeiten zu Flink. Neben den klassischen APIs haben beide Datenfluss-Frameworks relationale APIs und Unterstützung für Streaming. Allerdings ist Spark populärer und wird besser von anderen Systemen unterstützt. Die bei Spark signifikant größere Community bedeutet auch für Gradoop-Spark eine größere potenzielle Nutzerbasis und bessere Unterstützung bei Problemen. Aufgrund seiner Popularität gibt es für Spark auch viele Integrationen mit Cloud Anbietern und anderen Systemen. GraphX [18], GraphFrames [12], Morpheus [31] und weitere auf Spark basierende Frameworks lassen sich mit nur geringem Aufwand integrieren. Auch Machine Learning kann mit MLlib [29] leicht in den Analyseablauf eingebunden werden. Außerdem kann bei der Unterstützung von sowohl Batch- als auch Streaming-Verarbeitung mit Spark SQL die gleiche relationale API verwendet werden.

1.2 Ziel der Arbeit

Das Ziel dieser Arbeit ist die Implementierung und Evaluation des Extended Property Graph Models (EPGM) in Apache Spark. Dazu sollen zwei relationale Graph-Repräsentationen (Schemata) erarbeitet und implementiert werden. Für jedes dieser Schemata sollen grundlegende Operatoren der Gradoop-Referenzimplementierung implementiert werden. Mit Gradoop-Spark soll eine Grundlage des EPGM-Modells in Spark geschaffen und Spark als mögliches Framework für weitere Implementierungen evaluiert werden. Auf dieser Basis könnten mit der zukünftigen Integration von GraphFrames [12] und Morpheus [31] iterative Graphalgorithmen und Pattern Matching ermöglicht werden.

Die Implementierung basiert auf der Dataset-API von Spark SQL und verwendet die Programmiersprache Scala. Um weitere Arbeiten an Gradoop-Spark zu erleichtern, sollen vorhandene Datenformate und die API von Gradoop wenn möglich übernommen werden. Die Performanz und Funktionalität von Spark soll mit der Gradoop-Referenzimplementierung und der Gradoop-Table Implementierung von Elias Saalmann [35] verglichen werden. Es steht ein Forschungscluster der Universität Leipzig zur Verfügung, mit dem die Skalierung und Performanz für verschiedene Datenmengen auf verschiedenen Cluster-Konfigurationen evaluiert werden kann.

1.3 Aufbau der Arbeit

Das nächste Kapitel führt Grundlagen ein, die zum Verständnis dieser Arbeit benötigt werden. Nach einer Einführung in die Graphentheorie wird das EPGM formal definiert und verteilte Datenfluss-Systeme mit besonderem Fokus auf Spark vorgestellt. Kapitel 3 stellt weitere Arbeiten in den Bereichen der Graphanalyse auf verteilten Systemen vor. Die in Gradoop-Spark implementierten Schemata und Operatoren werden in Kapitel 4 vorgestellt. Kapitel 5 geht auf Details der Implementierung und dabei auftretende Probleme ein. Sowohl die allgemeine Architektur zur Unterstützung mehrerer Schemata als auch die Implementierungen der Operatoren werden erläutert. Die Evaluation in Kapitel 6 untersucht die Vor- und Nachteile der Implementierung in Bezug auf Funktionalität und Performanz. Gradoop-Spark wird dabei mit zwei anderen Implementierungen des EPGM verglichen. Das abschließende Kapitel 7 beurteilt die Ergebnisse der Arbeit und schlägt zukünftige Verbesserungen und Erweiterungen der Implementierung vor.

2 Grundlagen

Dieses Kapitel stellt die Grundlagen der Graphentheorie vor und definiert das EPGM. Mit Map-Reduce werden die Grundlagen verteilter Datenfluss-Systeme erklärt und schließlich wird Apache Spark vorgestellt.

2.1 Graphentheorie

Dieser Abschnitt basiert zu großen Teilen auf dem ersten Kapitel von Diestels „Graph Theory“ [14]. Ein Graph ist definiert als ein Paar $G = (V, E)$, wobei $E \subseteq [V]^2$ gilt. Um Unklarheiten zu vermeiden, wird außerdem angenommen, dass $V \cap E = \emptyset$ gilt. Die Elemente der Menge V heißen *Knoten* des Graphen G und die Elemente der Menge E sind die *Kanten* von G. Die Kanten sind hier jeweils als ungeordnete Menge zweier Knoten definiert, wodurch die Kanten keine inhärente Richtung aufweisen. Ein solcher Graph wird als *ungerichteter Graph* bezeichnet. Die typische Darstellung eines Graphen (siehe Abbildung 2.1) besteht aus Punkten oder Kreisen, die von Linien verbunden werden. Jeder Punkt oder Kreis steht für einen Knoten und jede Linie für eine Kante.

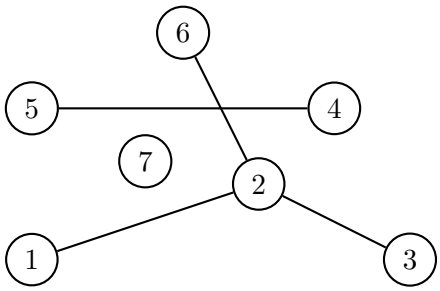


Abbildung 2.1: Ungerichteter Graph mit der Knotenmenge $V = \{1, \ldots, 7\}$ und der Kantenmenge $E = \{\{1, 2\}, \{2, 3\}, \{4, 5\}, \{2, 6\}\}$

Bei einem Graphen G bezieht sich $V(G)$ auf die Knotenmenge von G und $E(G)$ auf die Kantenmenge von G.

Alternativ zu den ungerichteten Graphen gibt es *gerichtete Graphen*. Hier werden Kanten als geordnetes Paar definiert, wodurch diese eine Richtung aufweisen. Diese Richtung wird in Abbildungen oft mit Pfeilspitzen dargestellt (siehe Abbildung 2.2).

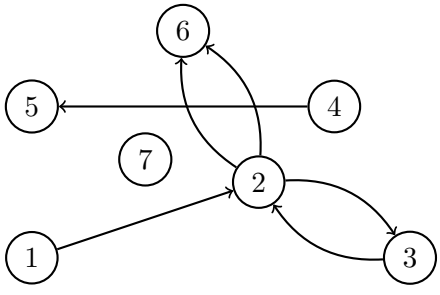


Abbildung 2.2: Gerichteter Multigraph mit der Knotenmenge $V = \{1, \ldots, 7\}$ und der Kantenmultimenge $E = \{(1, 2), (2, 3), (3, 2), (4, 5), (2, 6), (2, 6)\}$

Durch die Definition der Kanten als Menge kann für jedes Paar von Knoten nur eine Kante pro Richtung existieren. Soll ein Graph für ein Paar von Knoten mehr als eine Kante enthalten, muss ein *Multigraph* verwendet werden. Ein gerichteter Multigraph ist definiert als Paar (V, E) von einer Knoten- und einer Kantenmenge und zwei Abbildungen $s, t \colon E \mapsto V$, die jeder Kante einen Start- bzw. Zielknoten zuordnen. Die Kanten haben eine von ihren Start- und Zielknoten unabhängige Identität, wodurch mehrere Kanten mit den gleichen Start- und Zielknoten gleichzeitig existieren können. Alternativ wird die Kantenmenge zur vereinfachten Notation oft als Multimenge von Knotenpaaren definiert, die gleiche Kanten mehrfach enthalten kann (siehe Abbildung 2.2).

Seien $G' = (V', E')$ und $G = (V, E)$ Graphen. G' heißt *Teilgraph* oder *Subgraph* von G mit der Notation $G' \subseteq G$, wenn gilt dass $V' \subseteq V$ und $E' \subseteq E$. Umgekehrt wird G Obergraph oder Supergraph von G' genannt.

2.2 Extended Property Graph Model

Das *Property Graph Model* (PGM) definiert einen gerichteten Multigraphen von Knoten und Kanten, die je ein *Label* und eine Menge von *Eigenschaften* (Properties) haben. Labels ordnen die Elemente einfachen Typen zu, wie „Person“ oder „istFreundVon“. Eigenschaften bestehen aus Key-Value-Paaren und speichern weitere Attribute der Elemente, wie beispielsweise Name und Alter einer Person. Eigenschaften haben kein globales Schema, sondern werden für jedes Element einzeln definiert. Dieses Modell für Graphen ist in Graphdatenbanken weit verbreitet und wird auch in der kommenden ISO-Standard Sprache *GQL* [22] für die Abfrage von Graphen verwendet.

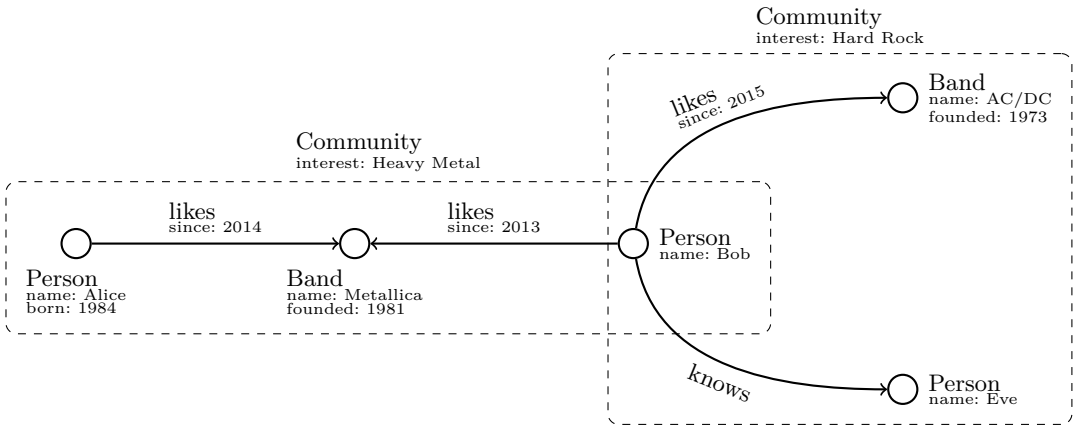


Abbildung 2.3: EPGM-Graph

Das *Extended Property Graph Model* (EPGM) erweitert PGM um das Konzept von *Graphmengen* und *logischen Graphen* (siehe Abbildung 2.3). Eine Graphmenge kann mehrere möglicherweise überlappende Property-Graphen enthalten, die als logischer Graph bezeichnet werden. Jedem der logischen Graphen gehört eine Teilmenge der Knoten und Kanten aus der Graphmenge an. Logische Graphen enthalten ebenfalls ein Label und Eigenschaften, die zusätzliche Informationen über den logischen Graphen beinhalten. Mit diesem Modell können Teilgraphen einer Datenbank als eigene Graphen beschrieben und in Operatoren verwendet werden. Mit *Gradoop* [26, 24] haben Junghanns et al. eine Referenzimplementierung des EPGM vorgestellt.

Junghanns et al. [25] definieren einen *Extended Property Graph* als Tupel:

$$\mathbb{G} = (L, V, E, l, s, t, T, \tau, K, A, \kappa)$$

L ist eine Menge von Graphköpfen, die logische Graphen repräsentieren, V eine Menge von Knoten und E eine Menge von Kanten. Die Kanten und Knoten gehören beliebig vielen logischen Graphen an. Diese Graphzugehörigkeit wird von der Abbildung $l\colon V \cup E \mapsto \mathcal{P}(L) \setminus \emptyset$ definiert. Die Start- und Zielknoten der Kanten sind durch die Abbildungen $s\colon E \mapsto V$ und $t\colon E \mapsto V$ definiert, wobei die Kanten eine Richtung von Start- zu Zielknoten aufweisen. T ist eine Menge von *Bezeichnern* (Label) und $\tau\colon L \cup V \cup E \mapsto T$ weist jedem Graph, jedem Knoten und jeder Kante ein Label zu. *Eigenschaften* sind durch eine Menge von Eigenschafts-Schlüsseln K, einer Menge von Eigenschafts-Werten A und der Abbildung $\kappa\colon (L \cup V \cup E) \times K \mapsto A \cup \{\epsilon\}$ definiert. ϵ wird verwendet, wenn ein Element für einen gegebenen Eigenschafts-Schlüssel keinen Wert hat.

Ein *logischer Graph* $G_i = (V_i, E_i)(i \in L)$ besteht aus einer Teilmenge von Knoten $V_i \subseteq V$ und einer Teilmenge von Kanten $E_i \subseteq E$, sodass $\forall v \in V_i\colon i \in l(v)$ und $\forall e \in E_i\colon s(e), t(e) \in V \wedge i \in l(e)$ gelten. Eine *Graphmenge* ist eine Menge von logischen Graphen $\mathcal{G} = \{G_1, G_2, \ldots, G_n\}$.

2.3 Verteilte Datenfluss-Systeme

Da für die Verarbeitung großer Datenmengen einzelne Rechner nicht ausreichen, wird die Berechnung auf mehrere Rechner verteilt. Das Ziel von verteilten Datenfluss-Systemen ist die Verarbeitung auf einem Cluster von vielen günstigen Maschinen anstatt auf teuren Großrechnern. Verteilte Datenfluss-Frameworks sind für die Abstraktion der Parallelisierung und Ausfallsicherheit dieser Systeme zuständig. Dabei werden sogenannte *shared nothing*-Cluster verwendet, die keinen gemeinsamen Zugriff auf Speicher haben.

2.3.1 MapReduce

Das *MapReduce* Programmiermodell wurde 2004 von Google für die Indexierung von Webseiten entwickelt [13]. Es eignet sich für die Verarbeitung von strukturierten und unstrukturierten Datensätzen bis zu mehreren Petabytes. Die Berechnung kann auf viele tausend Maschinen aufgeteilt und parallel ausgeführt werden. Die Daten liegen verteilt gespeichert vor und werden in den drei Phasen Map, Shuffle und Reduce verarbeitet. Ein MapReduce-Programm besteht aus einer Map-Funktion und einer Reduce-Funktion, die sich an den gleichnamigen Funktionen der funktionalen Programmierung orientieren. Die Eingabedaten können zum Beispiel Zeilen von Dokumenten oder Knoten eines Graphen sein. Sie liegen in Form von Schlüssel-Wert-Paaren vor und werden in der Map-Phase zu neuen Schlüssel-Wert-Paaren verarbeitet. In der Shuffle-Phase werden die von der Map-Funktion erzeugten Paare zu Gruppen kombiniert, die jeweils aus allen Paaren mit dem gleichen Schlüssel bestehen. Dabei werden die Daten über das Netzwerk zwischen den Maschinen übertragen, um alle Daten einer Gruppe auf einer Maschine verarbeiten zu können. In der Reduce-Phase wird jede Gruppe zu einem einzelnen Schlüssel-Wert-Paar reduziert. Eine einfache Reduce-Funktion würde zum Beispiel die Werte summieren und als Ergebnis ausgeben.

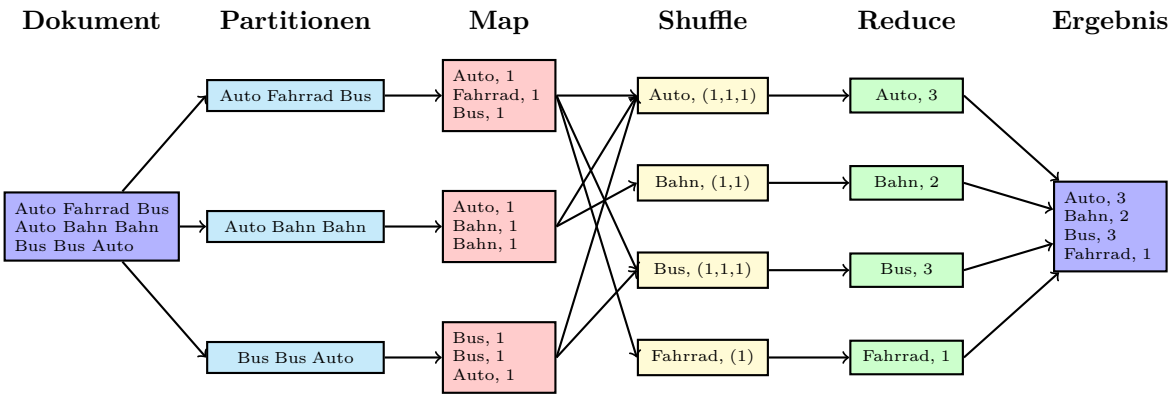


Abbildung 2.4: WordCount-Beispiel mit MapReduce

Ein bekanntes Beispiel ist das Zählen der Häufigkeit jedes Wortes in einer Menge von Dokumenten (siehe Abbildung 2.4). Die Anfangsdaten sind hierbei die eingelesenen Zeilen der Dokumente, die in der Map-Phase in einzelne Wörter aufgeteilt werden. Dabei ist das Wort der Schlüssel und der Wert ist eine Eins. Die Shuffle-Phase gruppiert diese Paare nun nach den Wörtern, wodurch für jedes Wort eine Menge von Einsen vorliegt. In der Reduce Phase müssen diese Einsen nur noch summiert werden, um die Häufigkeit des Wortes in den ursprünglichen Dokumenten zu erhalten.

Hadoop [37] ist eine Open Source Implementierung des MapReduce-Modells. Hadoop besteht aus dem Hadoop Distributed File System (HDFS) und einem MapReduce-Framework. Das HDFS koordiniert die verteilte Speicherung von Daten auf vielen Maschinen und schützt gegen Datenverlust bei Ausfällen einzelner Maschinen. Es setzt auf Standard-Linux-Dateisystemen auf, bietet ein virtuelles Dateisystem über das Netzwerk und eine Schnittstelle für MapReduce. Die Partitionierung der Daten bestimmt, welche Daten auf welchen Maschinen gespeichert werden. Dabei sollen die Daten möglichst gleichmäßig verteilt sein, um Engstellen (engl. Bottlenecks) bei einzelnen Maschinen zu vermeiden. Beim Ausfall von Maschinen oder neu hinzugefügten Maschinen soll die Partitionierung mit möglichst wenig Umverteilung angepasst werden. Die Daten werden auf mehrere Maschinen repliziert, um bei Ausfall gegen Datenverlust zu schützen.

MapReduce hat einige Einschränkungen, die zu der Entwicklung von neuen Modellen wie Apache Spark und Flink führten. Um die Fehlertoleranz zu gewährleisten, muss das Ergebnis der Map-Phase auf die Festplatten materialisiert werden. Der Ausfall einer Maschine führt so zwar zu keinem Datenverlust, aber der Festplattenzugriff verlangsamt die Ausführung. Die Map-Phase muss erst auf allen Maschinen beendet werden, bevor die Reduce-Phase beginnen kann. Somit ist keine Stream-Verarbeitung möglich, bei der einzelne Daten nach Komplettierung einer Phase sofort zu der nächsten Phase weitergeleitet werden. Durch diese strenge Trennung der Phasen sind besonders iterative Algorithmen wie PageRank und k-Means benachteiligt, da bei denen die drei Phasen mehrfach ausgeführt werden müssen. Außerdem lassen sich viele Algorithmen ohne die Einschränkung auf die Map- und Reduce-Funktionen effizienter berechnen.

2.3.2 Apache Spark

Apache Spark [40] wurde von MapReduce inspiriert und soll um etwa eine Größenordnung schneller sein [6]. Es verallgemeinert das Modell von MapReduce durch einen Ausführungsplan, der durch Aneinanderreihung von Funktionen wie Filter, Map, Reduce und Join erzeugt wird. Übertragungen über das Netzwerk wie bei Shuffle werden je nach Plan automatisch eingefügt. Die Daten werden während der Berechnung immer im Hauptspeicher behalten, wodurch der langsame Festplattenzugriff entfällt. Für die Ausfallsicherheit werden die Daten nicht bei jedem Schritt gespeichert, sondern für jede Partition vermerkt, wie genau sie berechnet wird. Bei Ausfall einer Partition werden die Daten neu berechnet, während die Berechnung der anderen Partitionen ungestört fortgesetzt werden kann. Spark unterstützt außerdem Streaming-Aufgaben, bei denen nicht ein großer Datensatz verarbeitet wird, sondern dauerhaft neue Daten ankommen und verarbeitet werden. Mit Integrationen wie GraphX [18] zur Graphanalyse und MLlib [29] zum Machine Learning lassen sich komplexe Analyseabläufe komplett in Spark durchführen.

Resilient Distributed Dataset Spark führt als Hauptabstraktion das *resilient distributed dataset* (RDD) ein. Es ist eine Sammlung von Daten, die verteilt auf den Knoten des Clusters liegen. Anstatt mit festen Map und Reduce Schritten wird mit RDDs ein Datenfluss als gerichteter azyklischer Graph von Transformationen definiert und ausgeführt. Dieser beginnt immer mit einer Datenquelle und endet mit einer finalen Aktion (engl. Action). Dazwischen können auf die Daten verschiedene Transformationen angewandt werden, die aus einem existierenden RDD ein oder mehrere neue RDDs erzeugen. Dazu gehören unter anderem *Filter*, *Map*, *Group* und *ReduceByKey*. Transformationen sind lazy und werden erst dann ausgeführt, wenn für das RDD eine Action ausgelöst wurde. Eine Action erzeugt aus einem RDD ein Ergebnis, das kein RDD ist. Dazu zählt das Speichern der Daten auf dem HDFS oder Funktionen wie *Reduce* und *Count*, die einen Wert an das Hauptprogramm (*driver program*) des Clusters zurückgeben. Um Daten bei Shuffle-Schritten über das Netzwerk zu übertragen, werden sie zu einem Binärformat serialisiert und so komprimiert. Der Default ist dabei Java-Serialisierung, alternativ kann aber auch der Kryo Serialisierer verwendet werden. Kryo ist schneller, erfordert für eine effiziente Komprimierung allerdings die vorherige Registrierung der verwendeten Datentypen.

```
1  val rdd = sc.textFile("hdfs://...")
2  val counts = rdd.flatMap(line => line.split(" "))
3      .map(word => (word, 1))
4      .reduceByKey((a, b) => a + b)
5  counts.saveAsTextFile("hdfs://...")
```

Listing 2.1: WordCount-Beispiel mit RDDs

Listing 2.1 zeigt, wie man mit RDDs die Häufigkeit jedes Wortes in einem Dokument zählt. Ähnlich wie MapReduce benötigen die meisten Transformationen die Übergabe einer Lambda-Funktion. Eine Map-Funktion erzeugt aus jedem Eingabeobjekt genau ein Ausgabeobjekt, während FlatMap beliebig viele Ausgabeobjekte erzeugt. In Zeile 2 resultiert `line.split(" ")` in einem Array und FlatMap erzeugt aus jedem Wert des Arrays einen Eintrag im neuen RDD. Reduce aggregiert alle Werte zu einem einzelnen, indem Werte paarweise kombiniert werden. ReduceByKey aggregiert

Gruppen von Werten, die durch einen Key identifiziert werden. Die Einträge liegen als Key-Value-Paare vor und werden pro Key zu einem Key-Value-Paar kombiniert. In Zeile 3 wird das Wort als Key und 1 als Wert definiert, die in Zeile 4 durch Addition der Werte pro Key aggregiert werden. Das Ergebnis enthält die Häufigkeit für jedes Wort.

Spark SQL Spark SQL baut auf RDDs auf und fügt Unterstützung für relationale Abfragen hinzu, die mit SQL oder mit der DataFrame-API verwendet werden können. Die Daten liegen in Tabellen vor, die durch DataFrames repräsentiert werden. Relationale Funktionen wie Select, Join, Filter und GroupBy erzeugen aus DataFrames neue DataFrames. Die Transformationen benötigen im Gegensatz zu RDDs keine Lambda-Funktionen, sondern Spaltenausdrücke. Ein elementarer Spaltenausdruck ist eine Referenz auf eine Spalte oder eine Konstante. Diese lassen sich zu komplexeren Ausdrücken transformieren und kombinieren. Dafür stehen in SQL typische Operatoren und Funktionen wie Addition, Count und Substring zur Verfügung. Die Ausdrücke ergeben je nach Zusammensetzung Anweisungen zur Berechnung von Werten, Aggregationen oder Bedingungen. Für jede der Funktionen gibt es ein Äquivalent, das in SQL-Abfragen verwendet werden kann. Ein DataFrame kann als Tabelle registriert werden, aus der mit einer SQL-Abfrage ein neues DataFrame erzeugt werden kann. Ähnlich zu relationalen Datenbanken werden die Funktionen und SQL-Abfragen vom Spark Optimierer *Catalyst* optimiert, während bei RDDs keine weitere Optimierung stattfindet.

```
1  val dataFrame = sc.textFile("hdfs://...").toDF("line")
2  val counts = dataFrame.select(split(Column("line"), " ").alias("words"))
3      .select(explode(Column("words")).alias("word"))
4      .groupBy("word").count()
5  counts.rdd.saveAsTextFile("hdfs://...")
```

Listing 2.2: WordCount-Beispiel mit DataFrames

Listing 2.2 zeigt, wie mit DataFrames die Häufigkeit jedes Wortes in einem Dokument gezählt werden. In Zeile 2 wird in der `select`-Funktion ein Spaltenausdruck zum Aufteilen von Zeilen in Wörter verwendet. Die „line“-Spalte wird mit der relationalen Funktion `split` aufgeteilt und in der neuen Spalte „words“ gespeichert. Diese Arrays werden mit `explode` in einzelne Einträge aufgeteilt und danach mit `groupBy` und der Aggregation `count` gezählt.

Ein DataFrame stellt einen logischen Plan zur Berechnung der Daten dar. Catalyst erzeugt aus diesem Plan einen physischen Plan, der direkt auf dem Cluster ausgeführt werden kann. Der logische Plan wird erst durch Anwendung von Optimierungsregeln oder Kostenfunktionen optimiert, danach wird aus dem optimierten Plan der physische Plan erzeugt. Ursprünglich wurde ausschließlich regelbasierte Optimierung verwendet, bei der durch vordefinierte Regeln unnötige Transformationen eliminiert werden und die Reihenfolge durch Heuristiken optimiert wird. Mittlerweile können anstatt von simplen Regeln auch Statistiken über die Daten verwendet werden, um die Kosten einzelner Schritte zu schätzen und den besten Plan zu wählen.

Dataset-API In Spark 2.0 wurde Spark SQL um die neue Dataset-API erweitert. Abbildung 2.5 zeigt den Aufbau von Spark SQL und die verfügbaren APIs. Ähnlich zu RDDs stellen Datasets

Mengen von Objekten dar, die verteilt vorliegen und mit Transformationen verändert werden können. Allerdings basiert ein Dataset intern auf einer Tabellendarstellung ähnlich zu DataFrames und hat relationale Operatoren. Datasets haben im Gegensatz zu DataFrames von außen sichtbare Typen und bieten neue typsichere relationale Operatoren an. Zusätzlich gibt es wie bei RDDs Funktionen, die mit Lambda-Funktionen direkt auf die Objekte zugreifen. Die Optimierung durch Catalyst ist bei diesen allerdings beschränkt, da die Lambdas nur teilweise analysiert werden können. Das WordCount-Beispiel für Datasets kann bis auf kleine Unterschiede in der Syntax sowohl ähnlich zu RDDs (siehe Listing 2.1) als auch zu DataFrames (siehe Listing 2.2) aussehen. Wegen der besseren Optimierung sollten wenn möglich relationale Funktionen bevorzugt werden.

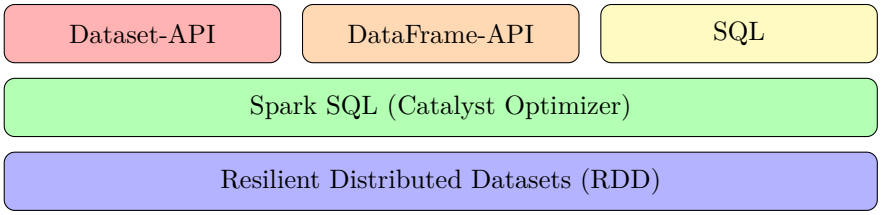


Abbildung 2.5: Spark SQL

DataFrame ist ab Spark 2.0 ein Alias für Dataset, das *Row*-Objekte enthält und wie gewohnt verwendet werden kann. *Row* lässt sich mit einem typlosen Tupel vergleichen und enthält alle Felder eines Tabelleneintrages. Transformationen zwischen Datasets und DataFrames sind damit trivial und zusätzlich sind sämtliche DataFrame-Operatoren für Datasets verfügbar. Diese Operatoren geben DataFrames zurück, da Spark den resultierenden Typ nicht kennt. Um das resultierende DataFrame zurück in ein Dataset umzuwandeln, muss lediglich der Typ in Kombination mit dem passenden Encoder bereitgestellt werden. Da es zu den DataFrame-Operatoren oft keine äquivalenten typsicheren Operatoren gibt oder diese umständlich zu verwenden sind, werden beide APIs oft gemischt.

Im Gegensatz zu RDDs verwenden Datasets keine normale Java-Serialisierung oder Kryo, sondern *Encoder*. Encoder sind ähnlich wie Kryo für die Umwandlung von Objekten zu Bytes und zurück zuständig, werden aber dynamisch generiert und resultieren in einem besonderen Binärformat. Das Format ermöglicht es, Operationen wie Filtern, Sortieren und Hashing ohne Deserialisierung der Objekte durchzuführen, wodurch häufige Serialisierung und Deserialisierung vermieden werden kann. Spark bietet Encoder für verschiedene Typen an, die zu komplexeren Typen kombiniert werden können. Primitive Typen wie Integer, String und Boolean können unter anderem zu Arrays, Tupeln, und Scala-Case-Klassen kombiniert werden. Case-Klassen sind ähnlich zu normalen Klassen mit kleinen Unterschieden, wodurch sie besser zur Modellierung von unveränderbaren Daten und Pattern Matching geeignet sind. Zum Beispiel werden Funktionen zum Vergleichen und Kopieren der Case-Klasse automatisch generiert. Mit implizit verfügbaren Funktionen werden bei der Erzeugung von Datasets die Encoder der verwendeten Typen wenn möglich automatisch generiert. Um viele nicht unterstützte Typen verwenden zu können, ist es möglich, Encoder aus normalen Serialisierern zu erstellen. Das Binärformat ist dann allerdings nicht optimiert und die (De-)Serialisierung ist langsamer als mit Sparks Encodern, da sie für die jeweiligen Typen entsprechend generiert und optimiert werden. Es ist derzeit nicht möglich, beide Varianten in einem komplexen Typ zu kombinieren.

Streaming Die Verarbeitung einer festen Menge von Daten wird auch *Batch-Verarbeitung* genannt. Eine Menge von Daten wird hier in einem abgeschlossenen Arbeitsablauf verarbeitet. Sparks RDDs fallen in diese Kategorie und auch die relationalen DataFrames bzw. Datasets werden meist als Batch verarbeitet. Das EPGM und seine Referenzimplementierung unterstützen derzeit ebenfalls nur Batch-Verarbeitung. Alternativ gibt es die *Streaming-Verarbeitung*, bei der laufend neue Daten ankommen und verarbeitet werden. Das Ziel ist meist, Daten nach ihrer Erzeugung direkt zu verarbeiten und Echtzeitanalysen auszuführen. In Spark wird dies mit Mikro-Batches realisiert. Anstatt einzelne Daten sofort nach der Berechnung einer Funktion weiterzureichen, wird ein Stream in viele kleine Batches aufgeteilt. Diese werden großteils wie normale Batches verarbeitet, allerdings gibt es zusätzliche Schwierigkeiten wie die Wahl eines Sichtfensters oder die Behandlung von verspäteten Daten. Außerdem steigt die in vielen Anwendungen kritische Latenz durch die Verwendung von Mikro-Batches im Vergleich zu kontinuierlichen Streams.

In Spark gibt es zwei verschiedene Streaming-APIs, die nacheinander entstanden sind. Die erste API definiert als Abstraktion *Discretized Streams* (DStreams) [36], die eine fortlaufende Reihe von RDDs darstellen. Da DStreams auf RDDs basieren, profitieren sie nicht von Catalyst. Die zweite API *Structured Streaming* [7] führt keine neue Abstraktion ein, sondern erweitert Spark SQL. Die Streaming-Aufgaben können genau wie Batch-Aufgaben mit DataFrames bzw. Datasets definiert werden, verwenden aber Streaming-Daten. Intern werden Discretized Streams und Spark SQL verwendet, wodurch von dem Catalyst-Optimierer profitiert wird. Für Structured Streaming gibt es seit einiger Zeit den experimentellen kontinuierlichen Modus, der die Discretized Streams durch einen kontinuierlichen Datenfluss ersetzt. Die verwendbaren Operatoren sind dabei eingeschränkt, aber geringere Latenzen können erreicht werden.

2.3.3 Apache Flink

Apache Flink [9] ist wie Spark von MapReduce inspiriert und nutzt ähnliche Verbesserungen wie einen flexiblen Ausführungsplan und vollständige Verarbeitung im Hauptspeicher. Vergleicht man Flink mit Spark, dann finden sich viele Gemeinsamkeiten bei den APIs und den verfügbaren Transformationen. Die Flink *DataSet-API* entspricht in etwa den RDDs von Spark. Daten werden als Batches mit vom Nutzer definierten Funktionen verarbeitet und die Optimierungsmöglichkeiten sind eingeschränkt. Für Streams gibt es die *DataStream-API*, die im Gegensatz zu Sparks DStreams von der DataSet-API getrennt ist. *Flink Table* ist eine relationale API, die sich mit Spark SQL vergleichen lässt. Daten werden in relationalen Tabellen gespeichert und mit relationalen Ausdrücken oder direkt mit SQL manipuliert. Flink Table kann wie Spark SQL ebenfalls Streams verarbeiten. Vom internen Aufbau unterscheiden sich Flink und Spark allerdings. Flink ist im Gegensatz zu Spark von Grund auf auf Streaming ausgelegt und behandelt Batch-Daten nur als Sonderfall von Streams, indem das Sichtfenster alle Daten abdeckt. Dies spiegelt sich in teils besserer Streamingleistung und besonders in geringeren Latenzen wider [11].

3 Related Work

Mit Gradoop haben Junghanns et al. [26, 24] ein Framework zur verteilten Graphanalyse veröffentlicht. Es führt das Extended Property Graph Modell (EPGM) ein, welches das Property Graph Modell um logische Graphen und Graphmengen erweitert.

3.1 EPGM auf Flink Table

Saalmann entwirft in seiner Masterarbeit [35] verschiedene relationale Schemata für das EPGM und implementiert diese in dem Prototyp *Gradoop-Table*. Dabei wird die Table-API von Apache Flink verwendet, die dem Nutzer eine relationale Abstraktion von Flink bietet. Das EPGM wird in drei verschiedenen relationalen Schemata implementiert und diese Implementierungen mit Gradoop verglichen. Gradoop-Spark und Gradoop-Table verfolgen mit der Implementierung des EPGM unter Verwendung einer relationalen API ein ähnliches Ziel. Dieser Abschnitt basiert zu großen Teilen auf Saalmanns Masterarbeit [35].

Grundschema Die entwickelten Schemata bauen alle auf einem Grundschema auf, das von den verschiedenen Schemata vervollständigt wird (siehe Abbildung 3.1). Knoten und Kanten werden in getrennten Datensätzen gespeichert, wobei Kanten mit *sourceId* und *targetId* Referenzen auf ihre Start- und Zielknoten beinhalten. Für die im EPGM beschriebene Erweiterung um logische Graphen wird pro logischem Graph ein Element gespeichert, auf das die zugehörigen Knoten und Kanten sich beziehen können. Alle Elemente enthalten eine Id und ein Label, welche direkt am Element gespeichert werden. Somit enthält jedes Schema mindestens drei Tabellen, je eine für Knoten, Kanten und Graphen. Für die fertigen Schemata fehlt noch die Zugehörigkeit der Knoten und Kanten zu logischen Graphen und Eigenschaftswerte der Elemente.

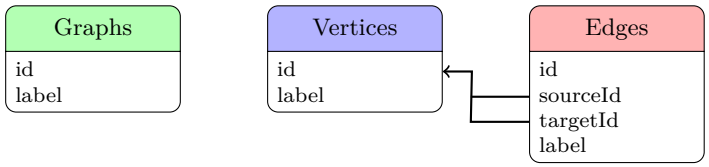


Abbildung 3.1: Grundschema

GVE-Schema Das *Graphs Vertices Edges*-Schema basiert auf der Referenzimplementierung Gradoop, in der alle Information direkt am Element gespeichert werden. Zu jeder der drei Tabellen des Grundschemas kommen weitere Felder dazu, aber weitere Tabellen werden nicht benötigt (siehe Abbildung 3.2). Die Graphzugehörigkeit wird als Menge von Graph-Ids in der Spalte *graphIds* gespeichert. Die Eigenschaftswerte *properties* bestehen aus einer Abbildung von Schlüssel zu Wert. Pro Element wird ein *properties*-Eintrag gespeichert, der alle Eigenschaftswerte des Elements beinhaltet.

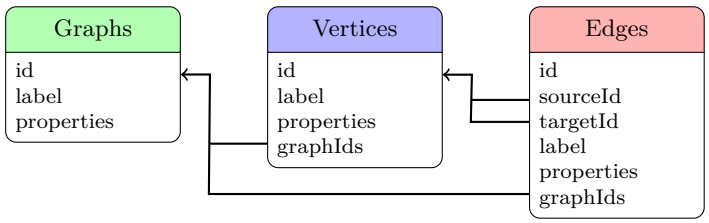


Abbildung 3.2: GVE-Schema

Von Vorteil ist, dass die Elemente alle Informationen direkt vorliegen haben. Um auf die Graphzugehörigkeit oder Eigenschaften zuzugreifen ist kein Verbund mit anderen Tabellen notwendig. Nachteilhaft ist, dass alle Tabellen vollständig eingelesen werden müssen. Selbst wenn zum Beispiel nur die Ids benötigt werden, müssen trotzdem die Eigenschaften und Labels in den Hauptspeicher geladen werden. Da Flink Table keine mengenwertigen Typen unterstützt, müssen für die Graphzugehörigkeit und die Eigenschaftswerte benutzerdefinierte Typen verwendet werden. Diese können von dem Optimierer schlechter analysiert und optimiert werden als native Typen.

Vertikales Schema Das vertikale Schema speichert die Eigenschaften und Graphzugehörigkeit nicht zusammen mit den Elementen, sondern lagert sie in weitere Tabellen aus. Die Tabellen des Grundschemas werden unverändert übernommen und von den neuen Tabellen referenziert (siehe Abbildung 3.3). Die Graphzugehörigkeit wird in einer Tabelle mit *vertexId* und *graphId* für die Knoten und einer mit *edgeId* und *graphId* für die Kanten gespeichert. Da jede Beziehung von Knoten bzw. Kante zu Graph einen eigenen Eintrag erhält, ist für die Speicherung der Graphzugehörigkeit kein benutzerdefinierter Typ mehr notwendig. Auch die Eigenschaften bekommen eigene Tabellen und werden nicht mehr in einem komplexen Typ gespeichert. Jeder Eintrag der Eigenschaften-Tabellen bildet eine einzelne Eigenschaft ab. Sie enthalten Referenzen auf das zugehörige Element, den Namen bzw. Schlüssel der Eigenschaft und den Wert der Eigenschaft.

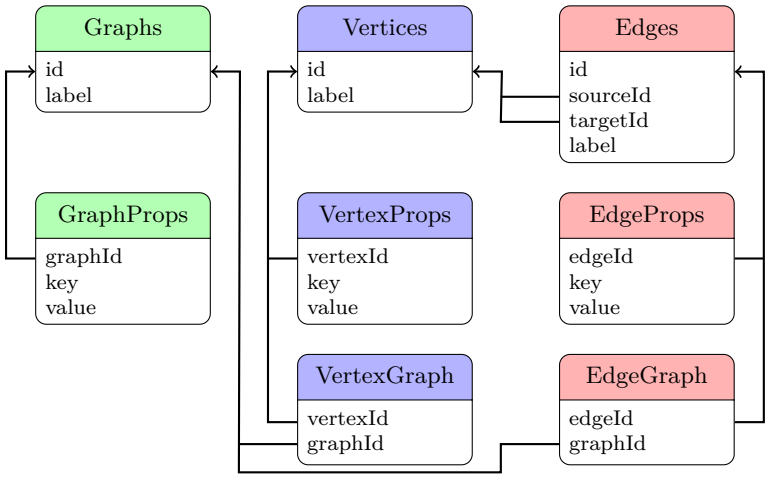


Abbildung 3.3: Vertikales Schema

Dieses Schema enthält keine komplexen Datentypen, wodurch die Implementierung der Operatoren vereinfacht wird und der Ablauf besser optimiert werden kann. Ein Nachteil ist, dass Joins benötigt werden, um die Eigenschaften und Graphzugehörigkeit bei Bedarf wieder mit den Elementen zu

verknüpfen. Die zusätzlichen Joins sind teuer, da sie die Übertragung von Daten über das Netzwerk erfordern. Allerdings kann die Aufteilung ein Vorteil sein, wenn beispielsweise die Anzahl der Knoten durch Filtern stark verringert und für die verbleibende Menge ein Verbund mit den Eigenschaften gebildet wird. Bei dem GVE-Schema muss stattdessen die ganze Menge inklusive der Eigenschaften durchlaufen und gefiltert werden.

Horizontales Schema Das horizontale Schema lagert die Eigenschaften im Vergleich zum vertikalen Schema in noch mehr Tabellen aus. Die Tabellen für die Graphzugehörigkeit entsprechen dem vertikalen Schema, aber die Eigenschaften werden nach ihren Schlüsseln getrennt gespeichert (siehe Abbildung 3.4). Für jeden existierenden Eigenschaftsschlüssel wird eine eigene Tabelle mit allen unter diesem Schlüssel registrierten Werten erstellt. Zum Beispiel stehen alle Eigenschaften mit dem Schlüssel *name* in einer eigenen Tabelle. Mit der Id wird auf das zugehörige Element in einer der drei Haupttabellen verwiesen. Dabei gibt es bei den Eigenschafts-Tabellen keine Unterscheidung zwischen Knoten, Kanten und Graphen, sie enthalten Eingenschaften aller drei Element-Typen.

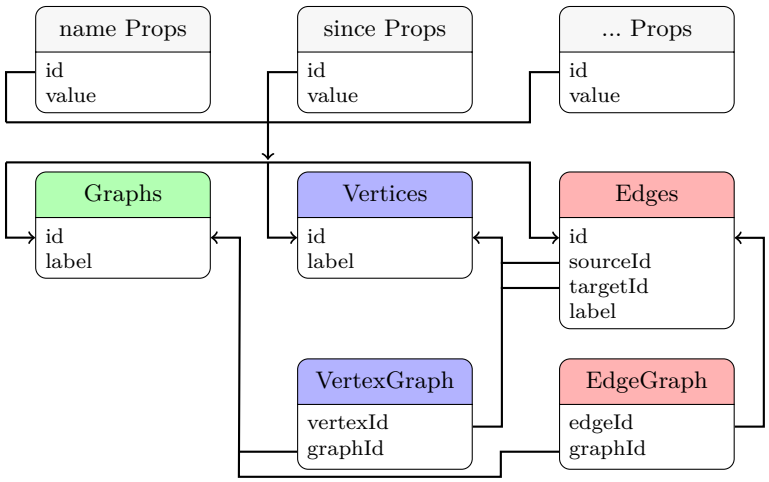


Abbildung 3.4: Horizontales-Schema

Die Aufteilung der Eigenschaften nach Schlüssel erfordert, dass alle Schlüssel vorher bekannt sind. Diese müssen daher bei Konvertierung eines der anderen Schemata in das horizontale Schema als erstes extrahiert werden, was zusätzliche Kosten bedeutet. Dadurch, dass die Schlüssel nicht in den Eigenschafts-Tabellen gespeichert werden müssen, lässt sich mit dem horizontalen Schema ein bisschen Speicherplatz sparen. Durch die hohe Anzahl an Tabellen werden allerdings besonders viele Verbünde und Vereinigungen benötigt und damit manche Operatoren verlangsamt.

Operatoren Für die Evaluation der Schemata implementiert Saalmann für jedes Schema mehrere Operatoren. Die Auswahl beschränkt sich auf die grundlegenden Operatoren Subgraph, Grouping und jeweils drei Mengenoperatoren für logische Graphen und Graphmengen. Diese Operatoren werden in Abschnitt 4.2 formal definiert und genauer erklärt. *Subgraph* erlaubt durch Filtern der Knoten- und Kantenmenge aus einem logischen Graph einen Teilgraphen zu extrahieren. *Grouping* extrahiert aus einem logischen Graphen ein strukturelles Schema des Graphen. Dafür werden nach benutzerdefinierten Kriterien Gruppen von Knoten und Kanten gebildet, die als Superknoten und Superkanten ausgegeben werden. Die auf logischen Graphen definierten Mengenoperatoren

Combination, *Overlap* und *Exclusion* basieren auf den Mengenoperatoren Vereinigung, Schnitt und Differenz. Sie erzeugen aus zwei möglicherweise überschneidenden Graphen unter Verwendung dieser Mengenoperationen auf den Knoten- und Kantenmengen einen neuen Graphen. Analog gibt es die auf Graphmengen definierten Mengenoperatoren *Union*, *Intersection* und *Difference*. Hier werden nicht die einzelnen Knoten und Kanten der Graphen betrachtet, sondern aus den logischen Graphen zweier Graphmengen wird eine neue Graphmenge erzeugt.

Implementierung Die Flink Table-API abstrahiert die verteilt vorliegenden Daten in Form von Tabellen. Abfragen können nur auf Tabellen ausgeführt werden und produzieren ausschließlich Tabellen. Vergleichbar zu den Transformationen der Spark Dataset-API können diese Abfragen verkettet werden und der daraus resultierende logische Plan wird von dem Optimierer *Calcite* optimiert.

Durch die Aufteilung in Knoten, Kanten und Graphen mit jeweils mehreren Tabellen müssen Zwischenergebnisse oft mehrfach verwendet werden. Beispielsweise die Implementierung des Grouping-Operators erfordert die mehrfache Verwendung einer Zwischenergebnis-Tabelle. Herkömmliche Datenbanksysteme würden diese bei Verwendung der `WITH ... AS` Syntax nur einmal berechnen und zwischenspeichern, aber Flink Table berechnet die Tabelle bei jeder Verwendung neu. Der Plan zur Erzeugung des Zwischenergebnisses wird mehrfach optimiert und ausgeführt. Um die Tabellen mit der Table-API zwischenzuspeichern, müssen die Daten in ein Flink DataSet und zurück in eine Tabelle konvertiert werden. Dabei wird der logische Abfrageplan unterbrochen, sodass alle auf diesem Zwischenergebnis basierenden Abfragen dieses ohne Neuberechnung verwenden können. In Apache Spark tritt das gleiche Problem auf, allerdings ist zum Zwischenspeichern keine Konvertierung der Daten in ein anderes Format notwendig.

Saalmann implementiert für jedes seiner Schemata ein auf CSV basierendes Datenformat zum Speichern der Graphdaten auf einem verteilten Dateisystem. Die existierenden CSV-Formate von Gradoop eignen sich nicht für das vertikale und horizontale Schema, da sie eine Aufteilung nach dem GVE-Schema voraussetzen. Besonders das horizontale Schema müsste beim Einlesen von Gradoops Formaten die Eigenschaften nach Eigenschafts-Schlüssel in die verschiedenen Tabellen aufteilen. Durch ein eigenes Format, bei dem die Dateien entsprechend dem Schema aufgeteilt sind, lässt sich das Lesen und Speichern der Daten optimieren.

Da die in Gradoop-Spark implementierten Schemata besser für die CSV-Formate von Gradoop geeignet sind, werden diese mit Gradoop-Spark direkt unterstützt und kein neues CSV-Format entwickelt (siehe Abschnitt 5.7.1). Das ermöglicht eine bessere Kompatibilität mit vorhandenen Graphdaten und Interoperabilität mit Gradoop. Im Gegensatz zu den drei simplen CSV-Formaten von Saalmann erfordern die Formate von Gradoop die Extraktion von Metadaten über die Eigenschaften. Mögliche Einflüsse der komplexeren Formate auf die Performanz zeigen sich in der Evaluation (siehe Kapitel 6). Die Vergleichbarkeit von Gradoop-Spark und Gradoop-Table ist aufgrund der Extraktion der Metadaten eingeschränkt.

Evaluation Die Komplexität der verteilten Ausführung wird durch die Flink Table-API erfolgreich mit relationalen Abfragen abstrahiert. Nutzer können unter Verwendung von SQL eigene

Operatoren implementieren, müssen dafür aber das Tabellenschema kennen und verstehen. Da die Tabellen und Spalten über Zeichenketten referenziert werden, ist nur geringe Unterstützung von Programmierumgebungen vorhanden, was die Implementierung eigener Operatoren erschwert. Teile der Probleme lassen sich jedoch mit Hilfsfunktionen zur Handhabung der SQL-Zeichenketten lösen. Durch die Einschränkung auf relationale Operationen weist die Table-API im Vergleich zur Flink DataSet-API eine geringere Ausdrucksstärke auf.

Die Evaluation der Performanz verwendet das *dbclu*-Cluster der Abteilung Datenbanken der Universität Leipzig mit 16 Knoten. Die verwendeten Graphen und Graphmengen basieren auf dem synthetischen LDBC-Benchmark und liegen jeweils in drei Skalierungen (1, 10 und 100) vor. Da für die Evaluation von Gradoop-Spark die gleichen Datensätze und Cluster zur Verfügung stehen, werden diese in Kapitel 6.1 genauer betrachtet.

Die Benchmarks zeigen für alle Schemata bei maximaler Parallelität eine bessere Performanz als die Gradoop Implementierung. Saalmann erklärt dies mit den folgenden drei Gründen:

Optimierung des Operatorbaums Mit dem *Apache Calcite*-Optimierer können die explizit deklarierten Abfragen der Flink Table-API besser als die Operatoren der Flink DataSet-API optimiert werden. Der Hauptspeicherverbrauch ist geringer, der `Row`-Datentyp kann besser serialisiert und optimiert werden und schlecht optimierbare benutzerdefinierte Funktionen werden weniger verwendet.

Optimierung der CSV-Datenquelle Die Datenquelle lädt nur die Daten, die tatsächlich benötigt werden. Sowohl Projektion, als auch Selektion können teilweise bereits in der Datenquelle durchgeführt werden.

Komplexere Datenquelle und -senke in Gradoop Gradoops Datenquelle und -senke lesen bzw. schreiben zusätzlich Metadaten zu den vorhandenen Eigenschaften. Die simple Implementierung in der Table-API hat einen größeren Speicherplatzverbrauch, ist aber schneller.

Das vertikale Schema und das GVE-Schema haben oft die beste Laufzeit, während das horizontale Schema bei vielen Operatoren leicht zurückliegt. Bei dem horizontalen Schema zeigt sich eine höhere Auslastung des Netzwerkes, weswegen für eine erfolgreiche Ausführung oft die Erhöhung des Netzwerkpuffers notwendig ist. Der Aufwand, den stark fragmentierten Graphen über das Netzwerk zu vereinen, und der durch den erhöhten Netzwerkpuffer verlorene Hauptspeicher könnten zu der schlechteren Performanz beitragen.

Die Arbeit zeigt, dass eine Implementierung des EPGM unter Verwendung einer relationalen API möglich ist und zu besserer Performanz führen kann. Allerdings ist die ausschließliche Verwendung der Table-API mit der verwendeten Version noch nicht möglich. Zum Speichern von Zwischenergebnissen oder für komplexe Operatoren wie *flatmap* muss weiterhin auf die Flink DataSet-API zurückgegriffen werden.

Gradoop-Spark kann von vielen der hier gezeigten Ansätze profitieren. Durch die Ähnlichkeiten von Spark SQL und Flink Table sollten sich viele der Folgerungen in Bezug auf die vorgestellten Schemata übertragen lassen. Der Catalyst-Optimierer von Spark SQL hat ähnliche Funktionalität

wie Apache Calcite und ermöglicht optimierte relationale Abfragen. Auch die eingeschränkte Ausdrucksstärke und doppelte Berechnung mancher Zwischenergebnisse treten mit Spark SQL auf.

3.2 Graphanalyse auf Spark

In den folgenden Abschnitten werden die Graph-Frameworks GraphX und GraphFrames vorgestellt, die auf RDDs bzw. DataFrames basieren. Diese populären Graph-Frameworks zeigen verschiedene Ansätze zur Nutzung und Integration von iterativen Graphalgorithmen in Spark. Mit verschiedenen Techniken werden bekannte Modelle der Graph-Verarbeitung für Spark angepasst und optimiert. Für Gradoop-Spark sind dabei besonders die Graph-spezifischen Optimierungen interessant.

3.2.1 GraphX

GraphX [18] ist ein verteiltes Graph-Framework, das auf Spark aufbaut und die effiziente Ausführung von iterativen Graphalgorithmen ermöglicht. Es implementiert mit RDDs eine Variante des Pregel-Modells [28] und eine Anzahl von Graphalgorithmen für Property-Graphen (PGM). GraphX wurde 2014 mit Spark 0.9.0 veröffentlicht und ist seitdem ein Bestandteil von Spark.

GraphX zeigt, dass moderne In-Memory-Frameworks wie Spark bei der Verarbeitung von Graphen mit spezialisierten verteilten Graph-Frameworks mithalten können. Es erzielt vergleichbare Performanz zu GraphLab [27] und Giraph [10], behält dabei aber die Vielfalt von allgemeinen Datenflusssystemen bei [18]. Mit Spark und GraphX lassen sich Workflows, die nicht nur Graphen, sondern auch andere Daten enthalten, in einem einzelnen Framework durchführen. Die Übertragung der Daten zwischen verschiedenen Systemen und der zusätzliche Verwaltungsaufwand entfallen.

Pregel Typische Graphalgorithmen wie PageRank basieren auf iterativen Transformationen der Knoten abhängig von den Eigenschaften der benachbarten Knoten und Kanten. Diese iterativen lokalen Transformationen bilden die Grundlage des *Pregel*-Modells [28]. Ein *Vertexprogramm* wird für jeden Knoten instanziiert und interagiert mit benachbarten Vertexprogrammen. Vertexprogramme können die Eigenschaften ihrer eigenen Knoten modifizieren (engl. Update) und senden Nachrichten an benachbarte Vertexprogramme. Ein *Superstep* besteht aus einer Ausführung aller Vertexprogramme inklusive dem Senden der Nachrichten. Nachrichten, die im Superstep N gesendet werden, kommen beim Empfänger in Superstep $N + 1$ an. Die Supersteps werden iterativ ausgeführt, bis alle Vertexprogramme konvergieren und abstimmen, das Programm zu beenden.

Vertexprogramme können parallel ausgeführt werden, was bei den meisten verteilten Graphsystemen mit einem synchronen Modell realisiert wird. Bei dem *bulk synchronous parallel*-Modell (BSP-Modell) [38] besteht jeder Superstep aus der lokalen Berechnung, der Kommunikation und einer Barriere zur Synchronisation. Die lokale Berechnung enthält das Update der Knoten und die Erzeugung der Nachrichten, die bei der Kommunikation übermittelt werden. Ein Superstep endet immer mit einer Barriere, die sicherstellt, dass die Berechnung und Kommunikation aller Vertexprogramme abgeschlossen ist. Erst wenn alle Vertexprogramme mit einem Superstep fertig sind, kann der nächste Superstep beginnen. Die Länge eines Supersteps setzt sich dabei aus dem Overhead

der Barriere zur Synchronisation und der Dauer des langsamsten Programms zusammen. Manche Systeme unterstützen alternativ ein asynchrones Modell, bei dem Vertexprogramme so bald wie möglich ausgeführt werden. Der Einfluss von einzelnen verspäteten Programmen wird so minimiert, aber die erhöhte Komplexität gleicht diesen Vorteil meist wieder aus.

GAS-Dekomposition Gonzalez et al. [19] beobachten, dass die meisten Vertexprogramme ankommende Nachrichten aggregieren und ausgehende Nachrichten in einer inhärent parallelen Schleife senden. Sie stellen die GAS-Dekomposition vor, die Vertexprogramme in drei jeweils parallelisierbare Phasen aufteilt: Gather, Apply und Scatter. In der *Gather*-Phase werden Nachrichten von Nachbarn gesammelt und zu einem einzelnen Wert aggregiert. *Apply* verwendet diese aggregierte Nachricht, um die Knoteneigenschaften zu aktualisieren und *Scatter* definiert die ausgehenden Nachrichten für jeden Nachbarn. Diese Aufteilung eignet sich als Grundlage für Implementierungen in verteilten Systemen, da die Phasen jeweils parallel ausgeführt werden können. Die GAS-Dekomposition führt zu einem Pull-basierten Nachrichtenaustausch, bei dem Nachrichten vom benachbarten Knoten abgefragt werden, anstatt sie an beliebigen Knoten senden zu können. Das schränkt die Kommunikation auf direkte Nachbarn ein, ermöglicht aber eine Reihe von Optimierungen.

GraphX-Abstraktion GraphX repräsentiert Property-Graphen in Spark als Paar von Knoten- und Kanten-RDDs. Jeder Knoten enthält Eigenschaften und eine einzigartige Id. Die Kanten enthalten ebenfalls Eigenschaften und werden durch die zwei Ids für den Startknoten und den Zielknoten identifiziert. Durch diese Darstellung lassen sich Graphen in Spark leicht mit anderen Daten kombinieren, um zum Beispiel weitere Eigenschaften hinzuzufügen und auch der Vergleich und die Kombination mehrerer Graphen wird ermöglicht. Solche Verknüpfungen sind typisch in der Graphanalyse, aber mit dem Pregel-Modell schwer zu realisieren.

```
1  class Graph[V, E] {
2      // Konstruktor
3      def Graph(v: Collection[(Id, V)], e: Collection[(Id, Id, E)])
4      // Sichten
5      def vertices: Collection[(Id, V)]
6      def edges: Collection[(Id, Id, E)]
7      def triplets: Collection[Triplet]
8      // Pregel
9      def mrTriplets(f: (Triplet) => M, sum: (M, M) => M): Collection[(Id, M)]
10     ...
11 }
```

Listing 3.1: Graph und seine Operatoren

Listing 3.1 enthält die `Graph`-Klasse von GraphX, die verschiedene Sichten auf den Graphen definiert. Neben den simplen Sichten auf die Knoten- und Kantenmengen ist ein Kernkonzept die Tripel-Sicht, wobei ein Tripel aus einer Kante zusammen mit ihren Start- und Zielknoten besteht. Die Tripel haben Ähnlichkeiten zu Tripeln im RDF-Graphmodell [8], die aus Subjekt, Prädikat und Objekt bestehen und zur Beschreibung semantischer Beziehungen genutzt werden. Die Tripel-Sicht `mrTriplets` wird von GraphX materialisiert und aktualisiert, um sie in der effizienten Berechnung von iterativen Graphalgorithmen zu verwenden.

Berechnungen nach dem Pregel-Modell können mit Join, GroupBy und Map Operatoren in Spark dargestellt werden. Mit Joins werden Tripel erzeugt, indem jede Kante mit ihren Start- und Zielknoten verbunden wird. GroupBy gruppiert diese Knoten-Kante-Knoten Tripel nach Zielknoten und berechnet Aggregate. Dadurch wird die Nachbarschaft jedes Knoten erzeugt und die Nachrichten aggregiert. Zwischen den Join- und GroupBy-Schritten wird die Map-Funktion genutzt, um die Nachrichten zu erzeugen und mit den aggregierten Nachrichten die Knoten zu modifizieren. Diese Transformationen entsprechen den Phasen der GAS-Dekomposition. Die Gruppierung sammelt entsprechend der *Gather*-Phase die Nachrichten, die an den gleichen Knoten geschickt werden. Die *Apply*-Phase entspricht dem zusätzlichen Map-Schritt, der die Knoten modifiziert. Und der Join-Schritt sendet den modifizierten Knoten ähnlich wie *Scatter* an alle Nachbarn.

Mit diesen Transformationen lassen sich die meisten iterativen Graphalgorithmen auf Spark ausführen. Für die schnelle Ausführung wird vor allem die Materialisierung und Aktualisierung der `mrTriplets`-Sicht durch verschiedene Techniken optimiert. Viele der Optimierungen profitieren von typischen Eigenschaften realer Graphen und Graphalgorithmen. Ohne Optimierungen sind die Laufzeiten von PageRank und Connected Components auf Spark um etwa eine Größenordnung schlechter als auf spezialisierten Graphsystemen [18]. Mit den Optimierungen werden vergleichbare Laufzeiten erreicht.

Optimierungen Die Hauptfaktoren bei den Optimierungen sind die Verteilung typischer realer Graphen und die Semantik des Pregel-Modells. Dabei spielen besonders die Konvergenz der Vertexprogramme und die iterative Neuberechnung der Tripel eine große Rolle.

Reale Graphen haben typischerweise um ein Vielfaches mehr Kanten als Knoten. Um dies auszunutzen, werden verschiedene Optimierungen verwendet, die die Bewegung von Kanten über das Netzwerk verhindern. Zum Beispiel bei dem Join-Schritt zwischen Knoten und Kanten werden die Knoten zu den Partitionen der Kanten bewegt, anstatt beides zu bewegen.

GraphX nutzt auch aus, dass die Tripel nicht nur einmal erstellt, sondern häufig aktualisiert werden. Zusätzlich ändern sich durch die Konvergenz der Vertexprogramme bei den meisten Aktualisierungen nur wenige Knoten. Anstatt alle Knoten erneut zu übertragen, werden zum Beispiel nur die geänderten Knoten übertragen, um die Tripel-Sicht zu aktualisieren.

Im Gegensatz zu Pregel sind für die Erzeugung der Nachrichten nicht nur die Eigenschaften der Startknoten, sondern auch die der Zielknoten verfügbar. Manche Algorithmen wie Connected Components können mit diesen zusätzlichen Informationen das Senden unnötiger Nachrichten vermeiden, während andere Algorithmen wie PageRank diese Information nicht benötigen. Mit der Join-Optimierung werden unnötige Joins eliminiert, wenn ein Vertexprogramm nicht auf die Start- oder Zielknoten zugreift.

GraphX implementiert ein effizientes Framework zur Ausführung von iterativen Graphalgorithmen auf Spark unter der Verwendung von zahlreichen Graph-spezifischen Optimierungen. Da GraphX allerdings auf RDDs basiert und damit nicht von dem Catalyst-Optimierer profitiert, gibt es keine Optimierungen über mehrere Schritte eines Arbeitsablaufs. Die Performanz ist mit auf Graphen

spezialisierten verteilten Frameworks vergleichbar [18]. Manche der Graph-spezifischen Optimierungen können auch in spezialisierten Systemen implementiert werden und deren Performanz weiter verbessern.

Durch den Fokus auf iterative Graphalgorithmen und die Verwendung von RDDs sind die meisten Prinzipien nicht in Gradoop-Spark anwendbar. Von der Verteilung realer Graphen sollte allerdings auch Gradoop-Spark bei der Optimierung profitieren können. Eine Integration von GraphX in Gradoop-Spark würde die Ausführung iterativer Graphalgorithmen ermöglichen. Da Gradoop-Spark allerdings Datasets verwendet und sich das Graphmodell unterscheidet, wäre eine Transformation der Daten notwendig.

3.2.2 GraphFrames

Dave et al. [12] stellen mit GraphFrames eine Weiterentwicklung der Konzepte hinter GraphX und ähnlichen Systemen vor. GraphFrames ermöglicht nicht nur relationale Transformationen und iterative Graphalgorithmen, sondern auch Pattern Matching. Pattern Matching nutzt ein vom Nutzer vorgegebenes Muster (engl. Pattern), um Teilgraphen zu extrahieren. GraphFrames baut auf der relationalen Spark DataFrame-API auf und lässt sich leicht mit existierenden DataFrames integrieren. Außerdem ermöglicht das dem Catalyst-Optimierer, über den kompletten Arbeitsablauf zu optimieren, während GraphX nur die Graphalgorithmen optimiert.

```
1  class GraphFrame {
2      // Verschiedene Sichten des Graphen
3      def vertices: DataFrame
4      def edges: DataFrame
5      def triplets: DataFrame
6      // Pattern matching
7      def pattern(pattern: String): DataFrame
8      // Relational-ähnliche Operatoren
9      def filter(predicate: Column): GraphFrame
10     def select(cols: Column*): GraphFrame
11     def joinV(v: DataFrame, predicate: Column): GraphFrame
12     def joinE(e: DataFrame, predicate: Column): GraphFrame
13     // Sicht Erstellung
14     def createView(pattern: String): DataFrame
15     // Partitionierung
16     def partitionBy(Column*) GraphFrame
17 }
```

Listing 3.2: GraphFrame und seine Operatoren

Listing 3.2 zeigt die GraphFrame-Klasse, in der die Sichten und Operatoren definiert werden. Ein GraphFrame wird ähnlich wie GraphX aus zwei DataFrames erstellt: Knoten und Kanten. Es bietet vier Sichten an, mit denen auf diese Daten zugegriffen wird. Ähnlich wie in GraphX können die Knoten und Kanten einzeln oder als Tripel von Knoten, Kante und Knoten betrachtet werden. Dies wird um die Pattern Matching Sicht erweitert, die ein Graph-Muster verlangt und ein GraphFrame mit allen Treffern zurückgibt. Mit `createView` können mit Graph-Mustern Sichten materialisiert werden, die zur Optimierung von Pattern Matching-Anfragen genutzt werden.

Zusätzlich gibt es einige relationale Operatoren, die für Graphen angepasst wurden. Zum Beispiel filtert `filter` die Knoten und Kanten mit dem übergebenen Prädikat und entfernt danach endständige Kanten, um einen korrekten Graphen zu garantieren. `filter` entspricht damit dem Subgraph-Operator von Gradoop.

Pattern Matching Der Pattern Matching Operator nutzt als Strings definierte Muster, um Teilgraphen mit diesem Muster zu extrahieren. Die Syntax ist dabei ähnlich zu Cypher [16] und definiert Mustergraphen aus Knoten mit runden Klammern und Kanten mit eckigen Klammern. Mit weiteren Einschränkungen können die Eigenschaften und Labels spezifiziert werden. Das resultierende DataFrame enthält für jeden Treffer alle benannten Elemente als Structs. Zum Beispiel beschreibt der folgende String einen Nutzer `u`, der die Artikel `x` und `y` gesehen hat.

```
(u:Person)-[viewed]->(x:Item), u-[viewed]->(y:Item)
```

Mit diesen Mustern können auch Sichten definiert werden, die für die Beschleunigung zukünftiger Anfragen materialisiert werden. Das können einfache Sichten wie die Tripel oder auch komplexe Formen wie Ringe sein. Der Optimierer von GraphFrames nutzt diese Sichten in Pattern Matching-Anfragen, wenn sie ein Teil des gesuchten Musters sind.

Implementierung Die Implementierung des Query-Planers für Pattern Matching setzt auf dem Catalyst-Planer auf. Der logische Plan wird mit den materialisierten Sichten optimiert und zur weiteren Optimierung und Umwandlung in einen physischen Plan an Catalyst weitergegeben. Für die Bestimmung der Kosten und Nutzen der Sichten werden Statistiken von der Catalyst-API gesammelt. Für einige Optimierungen, wie die in GraphX verwendete Join-Optimierung, wurde Catalyst selber modifiziert.

GraphFrames hat mit den Tripeln die gleiche Funktionalität wie GraphX, erweitert dies aber um Pattern Matching. GraphFrames ermöglicht die Erstellung beliebiger Sichten, die von einem kostenbasierten Planer zur Beschleunigung von Anfrage verwendet werden. Durch die Integration von GraphFrames in Spark ist es möglich, komplette Arbeitsabläufe von ETL (Extract, Transform, Load) über Graphanalyse bis zum Machine Learning mit MLlib [29] in einem System durchzuführen. Dabei kann mit dem Catalyst-Optimierer über den gesamten Arbeitsablauf optimiert werden. Mit Optimierungen wie der Join-Optimierung wurde Catalyst um weitere Optimierungen erweitert, die auch in anderen Systemen verwendet werden können.

Die Techniken zur Erweiterung des Catalyst-Planers sind ebenfalls für Gradoop-Spark interessant. Damit lassen sich Graph-spezifische Statistiken und Optimierungen zusätzlich zu allgemeinen Optimierungen mit Catalyst verwenden. Die Integration von GraphFrames in Gradoop-Spark würde Pattern Matching und iterative Graphalgorithmen ermöglichen. Da sowohl GraphFrames als auch Gradoop-Spark auf Spark SQL aufbauen, wäre eine solche Integration nicht schwer umzusetzen.

4 Konzept

In diesem Kapitel werden die Konzepte vorgestellt, die der Implementierung in Kapitel 5 zugrunde liegen. Die verwendeten Graph-Schemata und Graph-Operatoren werden definiert.

4.1 Schemata

Graphen können in Gradoop als Graphmenge oder als einzelner logischer Graph gespeichert werden. Sie bestehen beide aus Graphköpfen, Knoten und Kanten, wobei ein logischer Graph genau einen Graphkopf enthält. Jedes dieser Elemente besitzt eine eindeutige Id, ein Label und eine Menge von Eigenschaften (Properties). Das Label ordnet dem Element einen semantischen Typ zu. Die Eigenschaften bestehen aus Schlüssel-Wert-Paaren, mit denen Attribute der Elemente gespeichert werden. Knoten und Kanten müssen außerdem den logischen Graphen zugeordnet werden, in denen sie enthalten sind, indem Ids der Graphköpfe als Verweise gespeichert werden. Von und zu welchem Knote eine Kante verläuft wird durch Verweise auf die Start- und Zielknoten gespeichert.

Ein Schema ist die interne Darstellungsweise der Elemente eines Graphen bzw. einer Graphmenge. Vergleichbar mit einem Tabellenschema bei relationalen Datenbanken bestimmt es die Aufteilung der Daten. Das Schema beeinflusst die Performanz und Komplexität der Operatoren. Die Daten werden in Datasets von Spark SQL gespeichert, mit denen die Verteilung und Verarbeitung der Daten auf dem Cluster abstrahiert wird. Im Folgenden werden Datasets auch als Tabellen bezeichnet. In diesem Abschnitt werden die beiden Schemata GVE und TFL vorgestellt, die verschiedene Ansätze verfolgen. Da ein logischer Graph auch als Sonderfall einer Graphmenge mit nur einem Graph betrachtet werden kann, lassen sich mit dem gleichen Schema sowohl Graphmengen als auch einzelne logische Graphen darstellen. So wird der Aufwand der Implementierung verringert, die Wartbarkeit verbessert und die Extraktion einzelner logischer Graphen aus einer Graphmenge erleichtert.

4.1.1 Graph Vertices Edges

Das *Graph Vertices Edges*-Schema, im Folgenden mit *GVE* abgekürzt, entspricht dem Schema der Referenzimplementierung und dem GVE-Schema von Gradoop-Table. Für jeden der drei Elementtypen existiert eine Tabelle (siehe Abbildung 4.1), wobei jeder Eintrag sämtliche Informationen des Elements enthält. Die Graphzugehörigkeit der Knoten und Kanten wird als eine Menge von Ids dargestellt, die auf alle Graphköpfe verweisen, zu denen das Element gehört. Die Eigenschaften werden als Abbildung von Schlüssel zu Wert gespeichert. Da Spark in Tabellen mengenwertige Typen unterstützt, können die Eigenschaften und die Graphzugehörigkeit direkt in Spark abgebildet werden. Ein benutzerdefinierter Typ, der die Menge enthält, ist im Gegensatz zu Gradoop-Table nicht notwendig.

Das GVE-Schema weist hohe Datenlokalität auf, da die Eigenschaften und Graphzugehörigkeit an den Elementen gespeichert werden. Das minimiert in Operatoren die Anzahl der benötigten

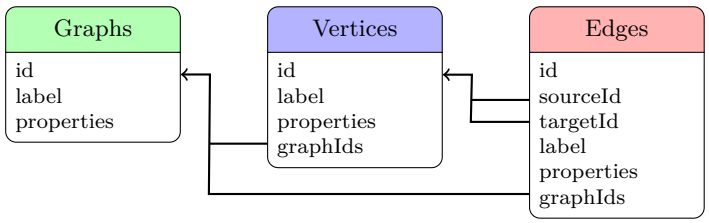


Abbildung 4.1: GVE-Schema

Joins und Vereinigungen, wodurch auch die Implementierung leichter wird. Es kann allerdings vorkommen, dass bei Joins mehr Daten übertragen werden als notwendig. Wenn zum Beispiel die Eigenschaften für die Berechnung eines Operators nicht benötigt werden, werden diese trotzdem in allen Schritten mit übertragen. Da im Endergebnis die Eigenschaften vorhanden sein müssen, kann der Optimierer diese nicht zur Reduktion der Datenmenge auslassen. Mit getrennten Tabellen für Eigenschaften und Elemente könnten solche Operatoren ausschließlich Elemente verwenden und die Eigenschaften erst am Ende Filtern. Bei besonders vielen Eigenschaften werden außerdem die einzelnen Elemente groß und erschweren die gleichmäßige Verteilung der Daten auf dem Cluster.

4.1.2 Tables for Labels

In dem *Tables for Labels*-Schema, im Folgenden *TFL* abgekürzt, werden die Elemente nach Labels aufgeteilt. Pro Label gibt es für Graphen, Knoten und Kanten jeweils zwei Tabellen: Die Eigenschaften-Tabelle für die Eigenschaften des Elements und die Element-Tabelle für den Rest (siehe Abbildung 4.2). Diese beiden Tabellen haben eine 1:1-Beziehung. Auch Elemente ohne Eigenschaften haben einen Eintrag in der Eigenschaften-Tabelle, wodurch die Komplexität der Operatoren verringert wird. Die Eigenschaften können so oft analog zu den Elementen verarbeitet werden. Die Eigenschaften und Graphzugehörigkeit werden wie in dem GVE-Schema als Abbildung bzw. Menge gespeichert.

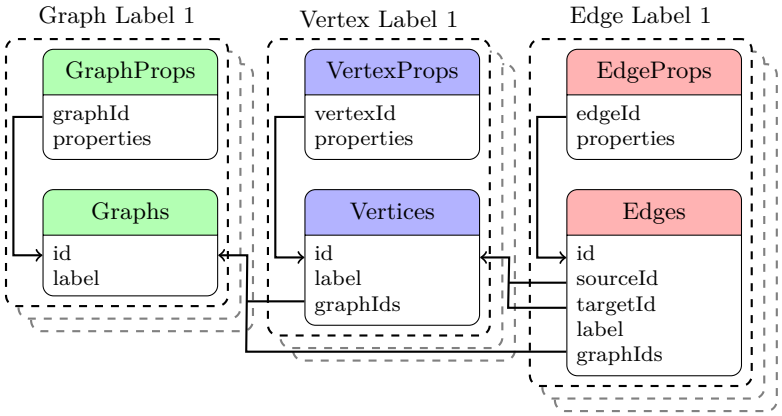


Abbildung 4.2: TFL-Schema

Die Aufteilung nach Label soll dem Optimierer ermöglichen, unnötige Tabellen komplett zu verwerfen. Zum Beispiel müsste ein Subgraph-Operator der nach den Labels *Person* und *knows* filtert nur die Tabellen für diese Labels betrachten. Eine ähnliche Aufteilung wird in dem Schema von Morpheus [31] verwendet, wodurch eine Integration von Morpheus erleichtert wird.

Die Aufteilung in Element-Tabelle und Property-Tabelle soll die Daten des Graphs von der strukturellen Information trennen. Strukturelle Operatoren können so auf vergleichsweise geringen Datenmengen arbeiten und die Eigenschaften hinterher herleiten. Ein alternatives TFL-Schema ohne diese Aufteilung wird in dieser Arbeit nicht weiter betrachtet.

Es wäre alternativ möglich, jeden Eigenschaftswert in einem eigenen Eintrag zu speichern, anstatt alle Eigenschaften eines Elements in einer Abbildung zusammenzufassen. Das könnte bei Graphen mit vielen Properties zu Geschwindigkeitsvorteilen und besserer Optimierbarkeit führen. Da Graphen oft nur wenige Properties pro Element enthalten, wird dieser Ansatz hier nicht weiter verfolgt.

4.2 Operatoren

Operatoren werden zur Transformation von logische Graphen und Graphmengen verwendet. Sie lassen sich basierend auf der Menge und Art der Eingabegraphen in verschiedene Kategorien einteilen. Es gibt Operatoren mit ein oder zwei Eingaben für logische Graphen oder Graphmengen. Zum Beispiel hat ein unärer Operator $f\colon G \mapsto G'$ einen logischen Graphen bzw. eine Graphmenge als Eingabe und gibt einen Graph oder eine Graphmenge aus. Ein binärer Operator $f\colon (G_1, G_2) \mapsto G'$ hat stattdessen zwei Graphen bzw. Graphmengen als Eingabe und wieder eine Ausgabe. In Tabelle 4.1 ist eine Übersicht der Operatoren mit den jeweiligen Eingabe- und Ausgabeformaten zu sehen. Mit diesen Operatoren ist genug Funktionalität für einfache analytische Abläufe gegeben.

Operator	Eingabe	Ausgabe
Subgraph	LG	LG
Grouping	LG	LG
Combination	(LG, LG)	LG
Overlap	(LG, LG)	LG
Exclusion	(LG, LG)	LG
Union	(GC, GC)	GC
Intersection	(GC, GC)	GC
Difference	(GC, GC)	GC

Tabelle 4.1: Operatorübersicht mit Eingabe- und Ausgabeformaten: Logischer Graph (LG) oder Graphmenge (GC)

4.2.1 Subgraph

Subgraph extrahiert aus einem logischen Graphen $G = (L, V, E)$ einen Teilgraphen G'. Von der Knotenmenge V und Kantenmenge E werden durch Filterfunktionen Teilmengen erzeugt, die als neuer logischer Graph zurückgegeben werden. Die Filterfunktionen sind von der Form $\Phi\colon A \mapsto \{true, false\}$ und werden für Knoten und Kanten getrennt definiert. Der Graphkopf L der Eingabe wird nicht benötigt und unverändert im Ergebnis weiterverwendet. Der Subgraph-Operator kann mit verschiedenen Strategien angewandt werden.

Zum Beispiel wären Filterfunktionen, die nur Knoten mit dem Label „Person“ und Kanten mit dem Label „friendOf“ behalten, so definiert:

$$\Phi_v(v) = \begin{cases} true & \text{wenn} \quad label(v) = \text{Person} \\ false & \text{sonst} \end{cases}$$

$$\Phi_e(e) = \begin{cases} true & \text{wenn} \quad label(e) = \text{friendOf} \\ false & \text{sonst} \end{cases}$$

Strategien Die *BOTH*-Strategie ist die simpelste Subgraph Strategie. Die Kanten- und Knotenmenge werden beide jeweils von einer Filterfunktion gefiltert und das Ergebnis wird direkt in einem neuen Graphen zurückgegeben:

$$\begin{aligned} G'_B =& (L, V'_B, E'_B) \\ V'_B =& \{\, v \in V \mid \Phi_v(v) = true \,\} \\ E'_B =& \{\, e \in E \mid \Phi_e(e) = true \,\} \end{aligned}$$

Abhängig von den Filterfunktionen und dem Eingabegraphen kann G'_B Kanten enthalten, deren Start- oder Zielknoten durch Φ_v herausgefiltert wurden, wodurch ein ungültiger Graph entsteht. Der Nutzer kann solche endständigen Kanten in einem weiteren Operator herausfiltern. Das Ergebnis enthält nun nur die Kanten, deren Ziel- und Startknoten in der Knotenmenge enthalten sind:

$$\begin{aligned} G''_B =& (L, V'_B, E''_B) \\ E''_B =& \{\, e \in E'_B \mid s(e), t(e) \in V'_B \,\} \end{aligned}$$

Die *VERTEX_INDUCED*-Strategie verwendet nur die Filterfunktion für Knoten Φ_v. Anstatt beide Mengen zu filtern, werden die Knoten gefiltert und die Kanten aus der gefilterten Knotenmenge hergeleitet. Dies funktioniert wie das Entfernen endständiger Kanten:

$$\begin{aligned} G'_V =& (L, V'_V, E'_V) \\ V'_V =& \{\, v \in V \mid \Phi_v(v) = true \,\} \\ E'_V =& \{\, e \in E \mid s(e), t(e) \in V'_V \,\} \end{aligned}$$

Bei der *EDGE_INDUCED*-Strategie werden nur die Kanten mit Φ_e gefiltert und die Knoten aus den gefilterten Kanten hergeleitet. Es werden alle Knoten entfernt, die mit keiner Kante aus der gefilterten Kantenmenge verbunden sind. Nur Knoten, die mit mindestens einer Kante verbunden sind, verbleiben. Sollten bereits vor dem Filtern unverbundene Knoten im Graphen vorhanden sein, werden auch diese entfernt.

$$\begin{aligned} G'_E =& (L, V'_E, E'_E) \\ V'_E =& \{\, v \in V \mid \exists e \in E'_E \colon (v = s(e) \vee v = t(e)) \,\} \\ E'_E =& \{\, e \in E \mid \Phi_e(e) = true \,\} \end{aligned}$$

4.2.2 Grouping

Grouping ist ein zentraler Operator in Gradoop, der einen logischen Graphen durch strukturale Gruppierung der Knoten und Kanten kondensiert. Gradoop-Spark implementiert die aktuelle *Keyed Grouping* Variante von Gradoop, dessen Funktionsumfang im Vergleich zum ursprünglichen Grouping erweitert wurde. Die folgende formale Definition basiert auf der Definition von Junghanns et al. [24] und wurde um die wichtigsten Punkte von Keyed Grouping erweitert. Für einen logischen Graphen $G = (L, V, E)$, Mengen von Grouping-Funktionen F_v und F_e und Mengen von Aggregationsfunktionen Λ_v und Λ_e erzeugt der Grouping-Operator einen zusammenfassenden logischen Graphen $G' = (L', V', E')$. Dieser zusammenfassende Graph ist ein Schemagraph, der die Struktur des ursprünglichen Graphen darstellt. Die Knoten V' heißen *Superknoten* und die Kanten E' *Superkanten*.

Der Graphkopf L des Eingabegraphen wird verworfen. Für das Ergebnis wird ein neuer Graphkopf L' erstellt.

Superknoten Die Knoten V der Eingabe G werden mit den Grouping-Funktionen durch identifizierende Merkmale zu Superknoten zusammengefasst. Sei $V' = \{v'_1, v'_2, \ldots, v'_n\}$ die Knotenmenge des zusammenfassenden Graphen G' und $s_v \colon V \mapsto V'$ eine surjektive Funktion, dann wird v'_i *Superknoten* genannt. Für alle $v \in V$ ist $s_v(v)$ der Superknoten von v. Die Knoten V werden basierend auf den Knoten-Grouping-Funktionen gruppiert, sodass gilt:

$$\forall u, v \in V \colon s_v(u) = s_v(v) \iff \forall f \in F_v \colon f(u) = f(v)$$

Der Wert $f(v)$ repräsentiert eine Gruppe und wird an dem Superknoten je nach Grouping-Funktion als Eigenschaft $\kappa(s(v), k_f(f)) = f(v)$ oder Label $\tau(s(v)) = f(v)$ gespeichert. Für die Speicherung als Eigenschaft wird der Schlüssel durch eine Funktion $k_f \colon F_v \cup F_e \mapsto K$ bestimmt.

Superkanten Sei $E' = \{e'_1, e'_2, \ldots, e'_n\}$ die Kantenmenge des zusammenfassenden Graphen G' und $s_e \colon E \mapsto E'$ eine surjektive Funktion, dann wird e'_i *Superkante* genannt. Für alle $(u, v) \in E$ ist $s_e((u, v))$ die Superkante von (u, v). Die Gruppierung der Kanten in E basiert sowohl auf den Kanten-Grouping-Funktionen als auch auf den Superknoten der Start- und Zielknoten. Durch die Abhängigkeit von den Start- und Zielsuperknoten ist das Ergebnis ein korrekter Graph, der die Struktur des ursprünglichen Graphen widerspiegelt:

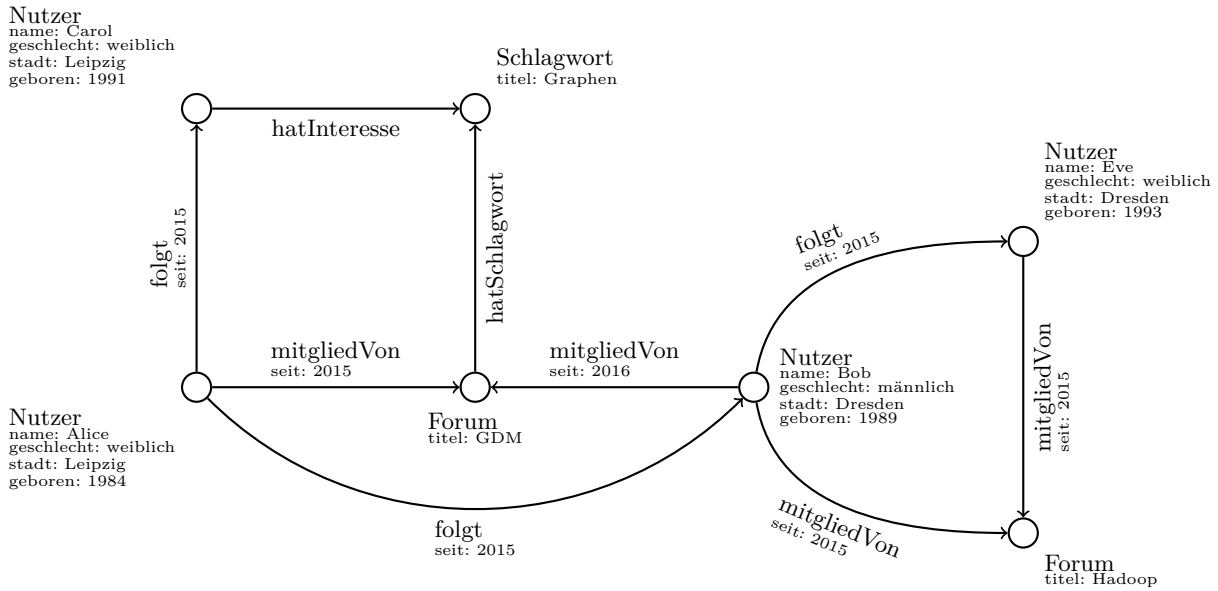
$$\begin{aligned}\forall (u, v), (s, t) \in E \colon s_e((u, v)) = s_e((s, t)) \iff & s_v(u) = s_v(s) \ \wedge \ s_v(v) = s_v(t) \\ & \wedge \ \forall f \in F_e \colon f((u, v)) = f((s, t))\end{aligned}$$

Analog zu den Superknoten repräsentiert der Wert $f((u, v))$ eine Gruppe und wird an der Superkante je nach Grouping-Funktion als Eigenschaft $(\kappa(s((u, v)), k_f(f)) = f((u, v)))$ oder Label $(\tau(s((u, v))) = f((u, v)))$ gespeichert.

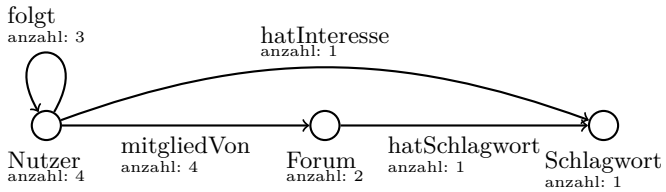
Aggregation Zusätzlich werden mit assoziativen und kommutativen Aggregationsfunktionen die Eigenschaften von Knoten und Kanten aggregiert. Diese Funktionen haben die Form $\Lambda_v\{\,\mathcal{P}(V) \mapsto A\,\}$ bzw. $\Lambda_e\{\,\mathcal{P}(E) \mapsto A\,\}$. Der resultierende Eigenschaftswert wird an den jeweiligen Superelementen gespeichert, wobei der Schlüssel durch eine Funktion $k_\alpha \colon \Lambda_v \cup \Lambda_e \mapsto K$ bestimmt wird:

$$\forall \alpha_v \in \Lambda_v, v' \in V' \colon \kappa(v', k_\alpha(\alpha_v)) = \alpha_v(\{\, v \in V \mid s_v(v) = v' \,\})$$
$$\forall \alpha_e \in \Lambda_e, e' \in E' \colon \kappa(e', k_\alpha(\alpha_e)) = \alpha_e(\{\, e \in E \mid s_e(e) = e' \,\})$$

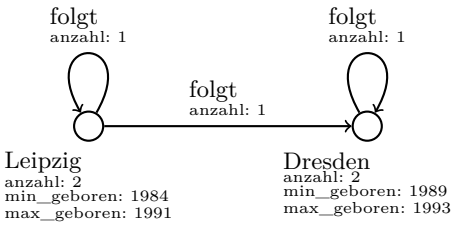
Beispiel Der Graph 4.3a zeigt ein soziales Netzwerk bestehend aus Knoten der Typen Nutzer, Forum und Schlagwort. Nutzer können anderen Nutzern folgen, Mitglied eines Forums sein und Interesse in Schlagwörter zeigen. Mit den Schlagwörtern werden die Foren nach Thema kategorisiert. Durch Eigenschaften werden zusätzliche Informationen wie Name, Wohnort und Titel gespeichert.



(a) Soziales Netzwerk



(b) Gruppiert nach Label



(c) Gruppiert nach Stadt

Abbildung 4.3: Sozialer Netzwerkgraph (a) gruppiert nach Label (b) bzw. Stadt (c)

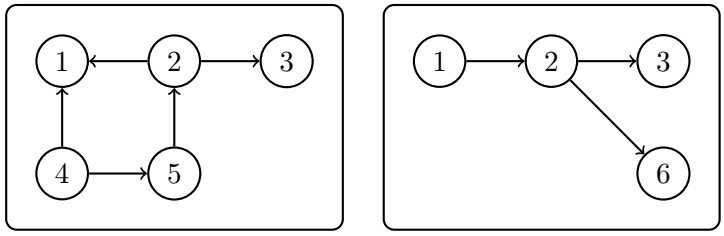
Der Graph 4.3b wird durch Gruppierung nach Label erzeugt. Die Anzahl der Knoten bzw. Kanten pro Gruppe wird mit den Aggregationsfunktionen $\alpha_{v_{anzahl}} \colon V \mapsto |V|$ und $\alpha_{e_{anzahl}} \colon E \mapsto |E|$ gezählt und als Eigenschaft gespeichert. Die vier Nutzer Alice, Bob, Carol und Eve werden zu dem `Nutzer`-Superknoten zusammengefasst, wobei die Anzahl ursprünglicher Nutzer in der Eigenschaft „anzahl“ mit dem Wert „4“ gespeichert ist. Die zwei Foren „GDM“ und „Hadoop“ und das eine Schlagwort „Graphen“ werden analog zu Superknoten zusammengefasst. Die Kanten des Typs `folgt` zwischen verschiedenen Nutzern werden zu einer `folgt`-Superkante von Nutzer zu sich selber

zusammengefasst. Die `mitgliedVon`-Kanten verlaufen von Nutzer zu Forum und werden entsprechend als Superkante von Nutzer zu Forum zusammengefasst. Analog werden die `hatInteresse` und `hatSchlagwort`-Kanten zusammengefasst.

In Graph 4.3c wird für die Gruppierung der Knoten der Wert der Eigenschaft „stadt“ verwendet. Da Alice und Carol aus Leipzig sind, werden sie in dem `Leipzig`-Knoten zusammengefasst. Entsprechend resultieren Bob und Eve in dem `Dresden`-Knoten. Die restlichen Knoten ohne die Eigenschaft „stadt“ werden in einem separaten Superknoten zusammengefasst, der in dieser Abbildung nicht aufgeführt wird. Aggregiert werden die Eigenschaften der Superknoten nach Anzahl, Minimum der Geburtsjahre und Maximum der Geburtsjahre. Die Kanten werden wieder nach dem Label gruppiert. Hier gibt es mehrere Superkanten des Typs `folgt`, da diese verschiedene Superknoten als Start- und Zielknoten haben. Eine `folgt`-Kante läuft von Alice aus Leipzig zu Carol aus Leipzig und resultiert in einer Superkante von Leipzig zu sich selber. Die `folgt`-Beziehungen zwischen Leipzigern und Dresdnern bzw. Dresdnern und Dresdnern führen zu den anderen Superkanten.

4.2.3 Mengenoperatoren – Graphen

Combination, *Overlap* und *Exclusion* sind Mengenoperatoren, die auf logischen Graphen definiert sind. Die Eingabe besteht aus zwei logischen Graphen und die Ausgabe aus einem neuen logischen Graphen. Abbildung 4.4 enthält die Beispielgraphen G_1 und G_2, die in den folgenden Abschnitten zur Veranschaulichung der Operatoren verwendet werden.



(a) G_1 (b) G_2

Abbildung 4.4: Graphen für Mengenoperator-Beispiele

4.2.3.1 Combination

Combination ist eine auf logischen Graphen definierte Vereinigung (siehe Beispiel in Abbildung 4.5). Zur Berechnung werden jeweils die Knotenmengen und Kantenmengen der beiden Graphen vereinigt und ein neuer Graphkopf wird erstellt. Der Ergebnisgraph enthält alle einzigartigen Knoten und Kanten beider Eingabegraphen. Der Graphkopf L' wird neu erzeugt:

$$G' = G_1 \cup G_2 = (L', V', E')$$
$$V' = V_1 \cup V_2$$
$$E' = E_1 \cup E_2$$

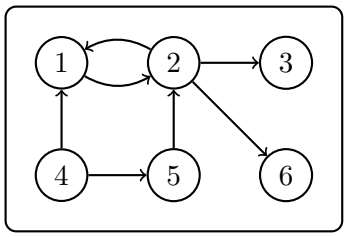


Abbildung 4.5: Combination Beispiel $G_1 \cup G_2$

4.2.3.2 Overlap

Der *Overlap*-Operator definiert den Schnitt der beiden Graphen (siehe Beispiel in Abbildung 4.6). Der Ergebnisgraph enthält die Knoten und Kanten, die sowohl in dem ersten als auch in dem zweiten Eingabegraphen enthalten sind. Um das zu erreichen, wird von den Knoten- und Kantenmengen beider Graphen die Schnittmenge gebildet. Da der neue Graph nur Elemente des ersten Graphen enthält, kann der Graphkopf des ersten Graphs weiterverwendet werden:

$$
\begin{aligned}
G' &= G_1 \cap G_2 = (L_1, V', E') \\
V' &= V_1 \cap V_2 \\
E' &= E_1 \cap E_2
\end{aligned}
$$

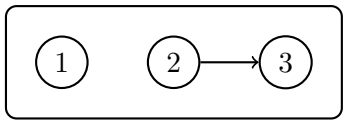


Abbildung 4.6: Overlap Beispiel $G_1 \cap G_2$

4.2.3.3 Exclusion

Exclusion bildet aus zwei logischen Graphen den Differenzgraphen (siehe Beispiel in Abbildung 4.7). Dies geschieht durch das Bilden der Differenzmengen der Knoten und Kanten. Allerdings kann eine einfache Differenz dazu führen, dass ein Start- oder Zielknoten einer Kante entfernt wurde, die Kante aber verbleibt. Um daraus resultierende ungültige Graphen zu vermeiden, ist es zusätzlich notwendig diese endständigen Kanten zu entfernen. Da der neue Graph nur Elemente des ersten Graphen enthält, kann der Graphkopf des ersten Graphen weiterverwendet werden:

$$
\begin{aligned}
G' &= G_1 \setminus G_2 = (L_1, V', E') \\
V' &= V_1 \setminus V_2 \\
E' &= \{\, e \in E_1 \setminus E_2 \mid s(e), t(e) \in V' \,\}
\end{aligned}
$$

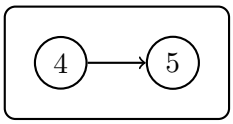
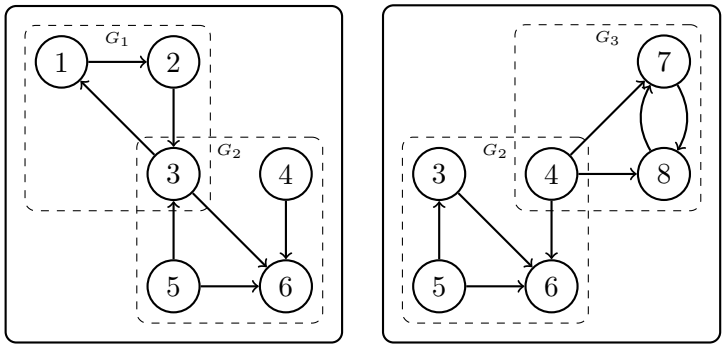


Abbildung 4.7: Exclusion Beispiel $G_1 \setminus G_2$

4.2.4 Mengenoperatoren – Graphmengen

Union, *Intersection* und *Difference* sind die entsprechenden Mengenoperatoren für Graphmengen. Die Eingabe besteht aus zwei Graphmengen und die Ausgabe aus einer neuen Graphmenge. Abbildung 4.8 enthält zwei Graphmengen, die in den folgenden Abschnitten als Beispiel verwendet werden.



(a) $\mathcal{G}_1 = \{G_1, G_2\}$ (b) $\mathcal{G}_2 = \{G_2, G_3\}$

Abbildung 4.8: Graphmengen für Mengenoperator-Beispiele

4.2.4.1 Union

Union bildet die Vereinigung von zwei Graphmengen (siehe Beispiel in Abbildung 4.9). Dafür wird die Vereinigung der Graphköpfe L_1 und L_2 gebildet, wodurch die Ergebnis-Graphmenge alle Graphen beider Eingabemengen enthält. Die zugehörigen Knoten und Kanten müssen aus den Graphköpfen hergeleitet werden. Da bei Union alle Knoten und Kanten benötigt werden, reicht dafür die Vereinigung der Mengen:

$$\begin{aligned}
\mathcal{G}' &= \mathcal{G}_1 \cup \mathcal{G}_2 = (L', V', E') \\
L' &= L_1 \cup L_2 \\
V' &= \bigcup_{G \in L'} V(G) = V_1 \cup V_2 \\
E' &= \bigcup_{G \in L'} E(G) = E_1 \cup E_2
\end{aligned}$$

4.2.4.2 Intersection

Intersection ist der Schnittmengenoperator für Graphmengen (siehe Beispiel in Abbildung 4.10). Die Graphköpfe L_1 und L_2 beider Eingaben werden geschnitten, sodass die Graphen verbleiben,

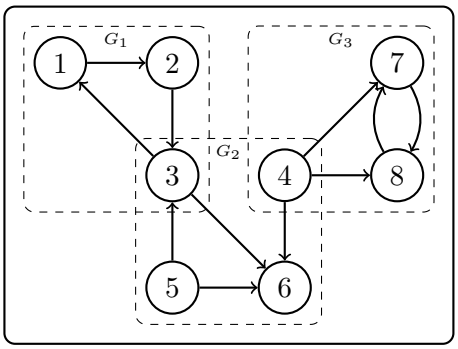


Abbildung 4.9: Union Beispiel $\mathcal{G}_1 \cup \mathcal{G}_2 = \{G_1, G_2, G_3\}$

die sowohl in der ersten als auch in der zweiten Eingabe enthalten sind. Von den resultierenden Graphköpfen L' werden die zugehörigen Knoten und Kanten abgeleitet, indem vom ersten Graphen diejenigen ausgewählt werden, die zu einem Graphkopf in L' gehören:

$$\begin{aligned}
\mathcal{G}' &= \mathcal{G}_1 \cap \mathcal{G}_2 = (L', V', E') \\
L' &= L_1 \cap L_2 \\
V' &= \bigcup_{G \in L'} V(G) = \{\, v \in V_1 \mid L' \cap l(v) \neq \emptyset \,\} \\
E' &= \bigcup_{G \in L'} E(G) = \{\, e \in V_1 \mid L' \cap l(e) \neq \emptyset \,\}
\end{aligned}$$

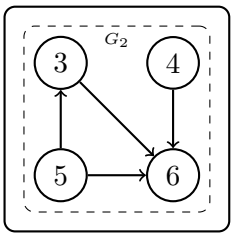


Abbildung 4.10: Intersection Beispiel $\mathcal{G}_1 \cap \mathcal{G}_2 = \{G_2\}$

4.2.4.3 Difference

Der *Difference*-Operator erzeugt die Differenz der beiden Graphmengen (siehe Beispiel in Abbildung 4.11). Die Graphköpfen L_2 der zweiten Menge werden von denen der ersten Menge abgezogen, sodass nur noch die Graphen erhalten bleiben, die in der ersten Menge, aber nicht in der zweiten enthalten sind. Von den resultierenden Graphköpfen L' werden die zugehörigen Knoten und Kanten

abgeleitet, indem vom ersten Graphen diejenigen ausgewählt werden, die zu einem Graphkopf in L' gehören:

$$\begin{aligned}
\mathcal{G}' &= \mathcal{G}_1 \setminus \mathcal{G}_2 = (L', V', E') \\
L' &= L_1 \setminus L_2 \\
V' &= \bigcup_{G \in L'} V(G) = \{\, v \in V_1 \mid L' \cap l(v) \neq \emptyset \,\} \\
E' &= \bigcup_{G \in L'} E(G) = \{\, e \in V_1 \mid L' \cap l(e) \neq \emptyset \,\}
\end{aligned}$$

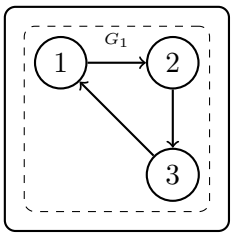


Abbildung 4.11: Difference Beispiel $\mathcal{G}_1 \setminus \mathcal{G}_2 = \{G_1\}$

4.3 Eigenschaften der Implementierung

Die API soll sich an Gradoops *GrALa* (**Gr**aph **A**nalytical **La**nguage) orientieren, um Nutzern und Entwicklern von Gradoop eine gewohnte Syntax zu bieten. Graphoperatoren sind hier unter anderem direkt als Methoden an logischen Graphen und Graphmengen verfügbar und lassen sich verketten. Auch die Namen der Operatoren und die zu übergebenden Argumente sind hier definiert.

Durch einen modularen Aufbau soll das Hinzufügen weiterer Schemata und Operatoren erleichtert werden. Auch Erweiterungen des EPGM zum Beispiel durch temporale Attribute sollten ermöglicht werden.

Zur Speicherung der logischen Graphen und Graphmengen wurde in Gradoop ein auf CSV basiertes Format entwickelt. Es gibt drei Dateien mit je einer für Graphköpfe, Knoten und Kanten. Jede Zeile einer Datei speichert ein Element inklusive der Werte der Eigenschaften. Um Speicherplatz zu sparen werden die Schlüssel und Typen der Eigenschaften für jedes Label extrahiert und als Metadaten in eine extra Datei geschrieben. Da die Aufteilung der Dateien dem GVE-Schema entspricht, wird dieses Format zum Einlesen und Schreiben von Graphen mit dem GVE-Schema verwendet. Zusätzlich gibt es eine indizierte Variante des CSV-Formats, bei der die drei Dateien nach den Labels aufgeteilt werden. Diese Format eignet sich für das Einlesen und Schreiben von Graphen im TFL-Format. Bei verteilter Ausführung werden von jeder Maschine Teildateien erzeugt, die verteilt auf dem verteilten HDFS Dateisystem liegen. Da diese Dateien auch verteilt eingelesen werden können, ist zum Speichern keine Umverteilung der Daten notwendig.

Zum Testen der Implementierung sind weitere Hilfsoperatoren notwendig. Mit dem *Equality*-Operator können zwei Graphen unter Verwendung einer kanonische Adjazenzmatrix auf Gleichheit überprüft

werden. Außerdem ist eine Datenquelle für die Sprache GDL [23] notwendig, um einfache Testgraphen aus Strings zu erzeugen. Auch die Transformation eines Graphen von dem GVE- in das TFL-Schema und umgekehrt muss implementiert werden.

5 Implementierung

Die in Kapitel 4 vorgestellten Konzepte wurden in einem Prototyp mit Apache Spark umgesetzt. Die in diesem Kapitel beschriebene Implementierung der Schemata und Operatoren basiert auf Apache Spark 2.4.5 und Scala 2.12.

5.1 Grundlagen

5.1.1 Graphen und Operatoren

Das EPGM unterscheidet zwischen logischen Graphen mit einem einzelnen Graphen und Graphmengen, die mehrere Graphen enthalten können. Diese werden von den Klassen `LogicalGraph` und `GraphCollection` repräsentiert. Das intern verwendete Schema ist für den Anwender weitgehend unsichtbar und hat keinen Einfluss auf die Verwendung des Graphen.

Operatoren sind als Klassen definiert, die die Methode `execute` implementieren. Je nach Art des Operators nimmt diese Methode ein oder zwei logische Graphen oder Graphmengen als Parameter und gibt den resultierenden Graphen zurück. Die Konfiguration des Operators und andere benötigte Daten werden bei der Initialisierung des Operators überreicht. Ein Operator muss für jedes Schema einzeln implementiert werden. Mit Methoden an LogicalGraph und GraphCollection wird abhängig von dem verwendeten Schema eine Instanz des Operators erzeugt und ausgeführt. Der Graph, von dem der Operator aufgerufen wird, wird als Eingabe an den Operator weitergereicht, während weitere Optionen durch Konstanten definiert werden oder als Parameter übergeben werden. Da jede Methode das Ergebnis des Operators zurückgibt, lassen sich die Operatoren intuitiv verketten. Bei manchen Operatoren werden mehrere Methoden definiert, die verschiedene Konstanten verwenden. Zum Beispiel gibt es für Subgraph drei Methoden, bei denen die verwendete Strategie bereits vordefiniert ist. Diese API ist an Gradoops *GrALa* orientiert.

5.1.2 Eigenschaften

Anstatt Eigenschaftswerte in Tabellen direkt abzuspeichern, wird der Typ `PropertyValue` verwendet. Da der Typ in Tabellen fest ist, müssten ohne diese Schachtelung Spalten für jeden möglichen Typ definiert werden. Die Implementierung unterstützt alle Typen der Referenzimplementierung (siehe Tabelle 5.1).

```
1  case class PropertyValue(bytes: Array[Byte]) { ... }
```

Listing 5.1: Definition von PropertyValue

Da Spark keine benutzerdefinierten Typen (UDT) mit eigener Serialisierung erlaubt, werden die Eigenschaftswerte in PropertyValue nicht als Objekt, sondern als `Array[Byte]` gespeichert (siehe Listing 5.1). Beim Erzeugen von PropertyValues und Auslesen der Werte müssen diese zu einem binären Format kodiert und dekodiert werden. Im ersten Byte des Arrays wird der gespeicherte Typ

Kategorie	Scala Typ	Größe (Bytes)
Numerisch	Short	2
	Int	4
	Long	8
	Float	4
	Double	8
	BigDecimal	dynamisch
Temporal	LocalDate	12
	LocalTime	16
	LocalDateTime	28
Andere	Null	0
	Boolean	1
	GradoopId	12
	String	dynamisch
Mehrwertig	Seq	dynamisch
	Set	dynamisch
	Map	dynamisch

Tabelle 5.1: Von PropertyValue unterstützte Datentypen

kodiert, um den Wert richtig dekodieren zu können (siehe Abbildung 5.1). Das reicht für Typen fester Länge, aber um Typen dynamischer Länge effizient dekodieren zu können, muss zusätzlich die Größe des Wertes in Bytes gespeichert werden. Je nach Größe wird dafür ein Short mit 2 Bytes oder ein Integer mit 4 Bytes verwendet, der die Größe des Wertes in Bytes angibt. Wenn ein Wert größer als 32767 Bytes ist, passt die Größe nicht in ein Short und ein Integer wird verwendet. Das erste Bit des Typen kennzeichnet, ob ein Short oder ein Integer verwendet wurde. Die Anzahl der möglichen Typen wird so halbiert, aber die verbleibenden 7 Bits reichen für die Kodierung aller unterstützten Typen.

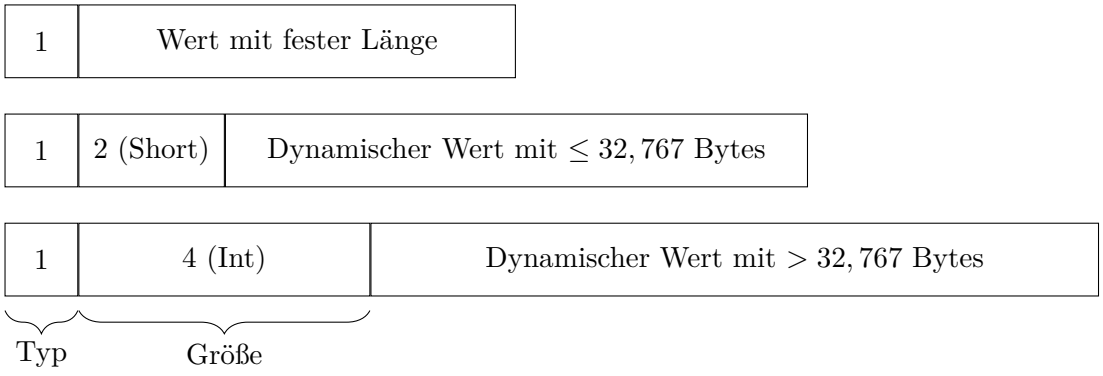


Abbildung 5.1: Byte Repräsentation von PropertyValue

GradoopId `GradoopId` ist die Implementierung einer verteilten einzigartigen Id in Gradoop. Um von Gradoop erstellte Graphen einlesen zu können, wurde diese Implementierung übernommen. `GradoopId` hat eine nichtdeterministische Funktion zum Erzeugen neuer Ids, die einen einzigartigen Wert aus verschiedenen Werten zusammensetzt. Einzigartige Ids der Maschine und des Prozesses garantieren, dass verschiedene Knoten im Cluster verschiedene Ids erzeugen, während ein Zeitstempel und ein mit jeder `GradoopId` steigender Zähler die Einzigartigkeit gegenüber vorhandenen Ids

garantieren. Wird diese Funktion in einem UDF verwendet, kommt es durch den Nichtdeterminismus zu falschen Ergebnissen (siehe Abschnitt 5.1.3). Um diese Limitierung von UDFs zu umgehen, wurde eine deterministische Abbildung von Sparks eingebauter Id auf `GradoopId` implementiert. Die Spark-Id ist vom Typ `Long`, welcher mit 8 Bytes kürzer ist als `GradoopId` mit 12 Bytes. Die Spark-Id wird komplett in der `GradoopId` eingebettet und die übrigen 4 Bytes werden mit einer Konstante anstelle der Maschinen-Id belegt. Für diese Konstante wird eine ungültige Maschinen-Id verwendet, wodurch Kollisionen zwischen den aus Spark-Ids hergeleiteten `GradoopIds` und nativen `GradoopIds` verhindert werden.

Mehrwertige Typen Die Typen `Seq`, `Set` und `Map` sind Sammlungen von Werten. Die Werte in den Sammlungen können selber wieder Sammlungen sein. Seq ist eine Liste, Set eine ungeordnete Menge und Map eine Abbildung von Schlüssel auf Wert, wobei alle Werte und die Schlüssel der Map vom Typ `PropertyValue` sind. Durch die Schachtelung in `PropertyValue` kann eine Sammlung sogar Werte mit verschiedenen Typen enthalten. Eine mögliche praktische Anwendung wäre ein Set mit Zahlen, die immer das kleinstmögliche numerische Format verwenden. Zur Speicherung im CSV-Datenformat müssen die in einer Sammlung enthaltenen Typen allerdings einheitlich sein. Pro Eigenschaft wird in den Metadaten nur ein Typ gespeichert, der für die ganze Sammlung gelten muss.

5.1.3 User Defined Functions

In den meisten Operatoren werden *User Defined Functions* (UDF) verwendet. Die Verwendung von UDFs ist notwendig, um die volle Funktionalität der Operatoren zu implementieren. Alleine mit den von Spark angebotenen relationalen Funktionen ist besonders die Manipulation komplexer Datentypen stark eingeschränkt. UDFs ermöglichen die beliebige Manipulation von deserialisierten Objekten in relationalen Transformation wie *select* oder *filter*. Dazu wird die gewünschte Scala-Funktion mit Spark registriert und kann dann in Transformationen verwendet werden.

In der verwendeten Version 2.4.5 unterstützen UDFs noch keine Case-Klassen als Eingabeparameter. Diese werden benötigt, wenn Eigenschaften oder GradoopIds in UDFs verwendet werden. Als Übergangslösung lässt sich das Problem durch die direkte Verwendung des internen Typs *Row* umgehen. Case-Klassen werden von Spark als Struct-Typen geparst, die intern als Row repräsentiert werden. Die Daten sind in einer Art Tupel gespeichert, dessen Felder die entsprechenden Felder der Case-Klasse enthalten. Da bei Row keine Typen sichtbar sind, müssen die Typen manuell umgewandelt werden. Dadurch wird die Nutzung von UDFs erschwert und der resultierende Code schwerer lesbar.

Im Gegensatz zu manchen optimierten Transformationen müssen die Objekte bei UDFs immer deserialisiert und hinterher wieder serialisiert werden. Auch kann der Optimierer UDFs nur teilweise analysieren, wodurch bestimmte Optimierungen wie *predicate pushdown* und *constant folding* verhindert werden können. Der Begriff constant folding kommt von der Compileroptimierung und beinhaltet das Erkennen und Vereinfachen von konstanten Ausdrücken. Bei predicate pushdown werden Filter und Selektion bereits in der Datenquelle durchgeführt, wodurch weniger Daten in den Speicher geladen werden müssen. Ähnliche Probleme treten bei den Typen

`UserDefinedAggregateFunction` (UDAF) und `Aggregator` auf, die für Aggregationen verwendet werden.

Von Spark wird angenommen, dass das Ergebnis der UDFs deterministisch ist, um mehr Optimierungen anwenden zu kennen. Diese Annahme lässt sich allerdings nicht deaktivieren und führt bei der Erzeugung von GradoopIds in UDFs zu Fehlern (siehe Abschnitt 5.1.2). Die Optimierung kann in manchen Fällen dazu führen, dass eine UDF mehrfach berechnet wird, anstatt einen einmalig berechneten Wert zu speichern. Bei nichtdeterministischen Funktionen führt die mehrfache Berechnung zu verschiedenen Werten, obwohl gleiche Werte erwartet werden.

5.2 Schema

Für jede Tabelle wird eine Scala-Case-Klasse definiert, deren Felder den Spalten entsprechen. Sparks Dataset-API kann aus den Case-Klassen automatisch die Spalten der Tabellen und deren Typen ableiten, wobei jedes Feld einer Case-Klasse einer Spalte der Tabelle entspricht. Die Datasets eines Schemas sind in einem Layout-Objekt enthalten, welches in LogicalGraph und GraphCollection verwendet wird. Da ein logischer Graph auch als Sonderfall einer Graphmenge mit genau einem Graphen betrachtet werden kann, wird in beiden Fällen die gleiche Layout-Klasse verwendet.

5.2.1 Typ-Klassen

Um Schema-unabhängige Klassen nicht für jedes Schema implementieren zu müssen, wird ein Schema von einem generischen Typ-Parameter repräsentiert. Auch die Schema-spezifischen Operatoren enthalten diesen Parameter, um zukünftige alternative Implementierungen des Schemas zu unterstützen. Zu jedem Schema gehört eine Vielzahl an Klassen für die Repräsentation der Tabellenspalten, dessen Typen für die Implementierung der Operatoren bekannt sein müssen. Mit Scalas Typsystem ist es möglich, in einem generischen Typ-Parameter mehrere Typen zu übergeben. Dazu werden in der Klasse des übergebenen Parameters Typen als Felder definiert, auf die mit dem `#`-Operator zugegriffen werden kann. Listing 5.2 zeigt die Klasse `EpgmTfl`, die ausschließlich als Typ-Parameter verwendet wird und eine Implementierung des EPGM-Modells im TFL-Schema definiert. Über diesen einen Typ-Parameter können alle für das Schema wichtige Typen verwendet werden. Zum Beispiel kann der Typ `type E = EpgmTflEdge` mit `EpgmTfl#E` verwendet werden.

```
1  class EpgmTfl extends Tfl[EpgmTfl] with Epgm {
2      type T = EpgmTfl            // selbst
3      type L = EpgmTflLayout      // Layout
4
5      // Elemente
6      type G = EpgmTflGraphHead   // Id, Label
7      type V = EpgmTflVertex      // Id, Label, Graph Ids
8      type E = EpgmTflEdge        // Id, Label, Graph Ids, Source Id, Target Id
9      type P = EpgmTflProperties  // Element Id, Label, Properties
10     ...
```

Listing 5.2: Typ-Klasse für EPGM in TFL. Typen werden endgültig definiert.

Um allgemeine Typ-Parameter für beide Schemata nutzen zu können, ist es notwendig, eine gemeinsame Oberklasse zu definieren (siehe Listing 5.3). Anstatt den Typ-Feldern einen festen Typ zuzuweisen, wird dieser mit dem `<:`-Operator auf ein Trait bzw. eine abstrakte Klasse und dessen Implementierungen beschränkt. Erst die endgültigen Typ-Klassen `EpgmGve` und `EpgmTfl` weisen die Typen fest zu. Wird ein generischer Typ `L` auf `LayoutType` beschränkt, dann kann zum Beispiel mit `L#L` das Layout verwendet werden, welches erst in `EpgmGve` und `EpgmTfl` endgültig definiert wird. Dabei lassen sich die in der abstrakten Klasse `Layout` definierten Methoden verwenden. Zusätzlich wird mit den Klassen `Gve` und `Tfl` eine Ebene dazwischen implementiert, die für die Implementierung der Operatoren verwendet wird.

```
1  class LayoutType[T <: LayoutType[T]] {
2      type L <: Layout[T]   // Layout
3      type M <: ModelType   // EPGM, TPGM, etc.
4
5      // Graphen
6      type LG <: LogicalGraph[T]
7      type GC <: GraphCollection[T]
8
9      // Graph Factories
10     type LGF <: LogicalGraphLayoutFactory[T]
11     type GCF <: GraphCollectionLayoutFactory[T]
12 }
```

Listing 5.3: Basis-Typ-Klasse

Die Operator-Methoden an `LogicalGraph` und `GraphCollection` wurden für GVE und TFL in Traits implementiert. Die Typen `LG` und `GC` werden in den Typ-Klassen `Gve` und `Tfl` um den Trait des jeweiligen Schemas erweitert. Zum Beispiel wird `LG` in `Gve` um den Trait `GveLogicalGraphOperators` erweitert. Mit diesen Typen können die Graphen mit beiden Schemata verwendet werden, ohne den Zugriff auf die Operator-Methoden zu verlieren.

5.2.2 GVE

```
1  class EpgmGveLayout(graphHeads: Dataset[EpgmGveGraphHead],  // Graphköpfe
2      vertices: Dataset[EpgmGveVertex],                       // Knoten
3      edges: Dataset[EpgmGveEdge])                            // Kanten
4      extends GveLayout[EpgmGve](graphHeads, vertices, edges) // Abstrakte Klasse
```

Listing 5.4: GVE-Layout Definition

Für die Implementierung des GVE-Schemas müssen drei Tabellen definiert werden: Graphköpfe, Knoten und Kanten. Diese Tabellen werden als Datasets in der Layout-Klasse gespeichert (siehe Listing 5.4). Die Typen, auf denen die Datasets basieren, definieren das Schema der Tabellen. Zum Beispiel die Klasse `EpgmGveVertex` enthält für jede der Tabellenspalten einen Parameter, aus dem der Typ und Name der Spalte abgeleitet wird (siehe Listing 5.5). Damit Spark die Tabellen richtig herleiten kann, müssen unterstützte Typen verwendet werden. Die unterstützte Typen basieren auf simplen Typen wie `String` und `Array[Byte]`, die zu komplexen Typen wie `Map` und Case-Klassen kombiniert werden können. Die anderen Tabellen sind analog nach dem in Abschnitt 4.1.1 definierten Schema definiert.

```
final case class EpgmGveVertex(var id: GradoopId,    // Id
    var label: String,                               // Label
    var properties: Map[String, PropertyValue],      // Eigenschaften
    var graphIds: Set[GradoopId])                    // Graphzugehörigkeit
    extends GveVertex                                // Abstrakte Klasse
```

Listing 5.5: GVE-Vertex Definition

5.2.3 TFL

Das TFL-Schema hat für jede der Element-Tabellen eine weitere Tabelle, die die Eigenschaften der Elemente enthält. Außerdem werden alle Tabellen in Abbildungen von Label zu Tabelle gespeichert. Um diese Abbildung zu erzeugen, müssen die Label beim Einlesen des Graphen bekannt sein. Sind die Label nicht bekannt, zum Beispiel beim Konvertieren vom GVE-Schema in das TFL-Schema, dann müssen die sie erst teuer extrahiert werden. Durch die Aufteilung nach Label ist in den Tabellen keine Label-Spalte notwendig. Anstatt diese zu entfernen, wird das Label in Spark als Konstante gespeichert. So kann leichter mit relationalen Ausdrücken auf die Label zugegriffen werden. Erst wenn das Label wirklich benötigt wird, materialisiert der Optimierer die Label-Spalte.

```
class EpgmTflLayout(graphHeads: Map[String, Dataset[EpgmTflGraphHead]], // Graphköpfe
    vertices: Map[String, Dataset[EpgmTflVertex]],                      // Knoten
    edges: Map[String, Dataset[EpgmTflEdge]],                           // Kanten
    graphHeadProperties: Map[String, Dataset[EpgmTflProperties]], // G Eigenschaften
    vertexProperties: Map[String, Dataset[EpgmTflProperties]],    // V Eigenschaften
    edgeProperties: Map[String, Dataset[EpgmTflProperties]])      // E Eigenschaften
    extends TflLayout[EpgmTfl](graphHeads, vertices, edges,     // Abstrakte Klasse
        graphHeadProperties, vertexProperties, edgeProperties)
```

Listing 5.6: TFL-Layout Definition

Für oft genutzte Transformationen sind Hilfsfunktionen implementiert, die die Handhabung der Maps erleichtern. Zum Beispiel `inducePropMap` wird genutzt, um Eigenschaften zu entfernen, zu denen kein Element existiert. Da Datasets für jedes Label einzeln verglichen und transformiert werden müssen, sind Transformationen ohne diese Hilfsfunktionen komplex und fehleranfällig.

5.3 Subgraph

Subgraph verwendet Filterausdrücke, um die Knoten- und Kantenmenge zu filtern. Filterausdrücke sind Spaltenausdrücke, die einen Wahrheitswert berechnen und der *WHERE*-Klausel in SQL entsprechen. Für die Konstruktion von oft verwendeten Filtern gibt es Hilfsfunktionen, die ohne Wissen über das Tabellenschema verwendet werden können. Für den *Subgraph*-Operator müssen drei Strategien implementiert werden. Die simpelste ist die *BOTH*-Strategie. Hier werden sowohl die Knoten als auch die Kanten mit den angegebenen Filterausdrücken gefiltert. Durch das Filtern können endständige Kanten zurückbleiben, bei denen einer der Knoten entfernt wurde. Diese Kanten lassen sich mit dem Hilfsoperator *RemoveDanglingEdges* entfernen. Bei der *VERTEX_INDUCED*-Strategie werden nur die Knoten direkt gefiltert. Danach werden die Kanten entfernt, von

denen der Start- oder Zielknoten entfernt wurde. Hierfür lässt sich wieder der Hilfsoperator *RemoveDanglingEdges* verwenden. Als letztes gibt es die *EDGE_INDUCED*-Strategie, bei der nur die Kanten direkt gefiltert werden. Die Knoten werden hinterher so gefiltert, dass jeder Knoten mit irgendeiner Kante verbunden ist.

GVE-Schema Für das GVE-Schema ist die Implementierung der *BOTH*-Strategie simpel. Die Knoten- und Kantenmengen werden mit Sparks Filter-Operator gefiltert und in einem neuen Graphen zurückgegeben. Die Anwendung von *RemoveDanglingEdges* bleibt hier dem Anwender überlassen. Bei *VERTEX_INDUCED* werden die Kanten in *RemoveDanglingEdges* durch zwei Joins vom Typ `LEFT SEMI JOIN` mit den Knoten hergeleitet. Erst werden die Kanten über die SourceId und dann über die TargetId mit den Knoten verbunden. Durch den Join fallen dabei die endständige Knoten weg. `LEFT SEMI JOIN` ist equivalent zu `INNER JOIN`, bei dem das Ergebnis nur die Spalten der linken Tabelle enthält. Dadurch ist kein zusätzlicher `SELECT`-Schritt nach der Vereinigung notwendig. Bei *EDGE_INDUCED* werden die Knoten gefiltert und die Knoten mit einem `LEFT SEMI JOIN` mit den Kanten hergeleitet. Als Join-Bedingung wird `vertices.id isin (edges.sourceId, edges.targetId)` verwendet, wodurch nur die Knoten verbleiben, die mit einer Kante verbunden sind.

TFL-Schema Die Implementierung für das TFL-Schema ist etwas komplizierter. Grundsätzlich sind in TFL die gleichen Spalten wie in GVE vorhanden, nur sind diese aufgeteilt. Da die Filterausdrücke sich potentiell auf jede dieser Spalten beziehen können, muss die Aufteilung der Elemente beachtet werden. Beispielsweise benötigt ein Filter, der alle Knoten nach dem Label „Person“ und der Property `age > 20` filtert, sowohl das Hauptelement von TFL als auch die Properties. Um das möglichst allgemein zu unterstützen, wird vor jeder Filteroperation das Element mit seinen Eigenschaften durch ein Join verbunden. Dies ließe sich in diesem Beispiel verhindern, indem Knoten und Properties einzeln gefiltert und danach mit einem Join verbunden werden. Da dies allerdings nicht allgemeingültig möglich ist, werden die Elemente mit den Eigenschaften kombiniert. Die Aufteilung des TFL nach Labels spielt beim Filtern keine Rolle. Erst beim Herleiten der Datasets in der *VERTEX_INDUCED* und *EDGE_INDUCED*-Strategie sind die Label relevant. Da die Start- und Zielknoten einer Kante im Allgemeinen andere Label als die Kante selber haben, müssen alle Kombinationen der Label beachtet werden. Um diese Joins zu berechnen, werden die Tabellen aller Label auf einer Seite des Joins vereinigt. Bei *VERTEX_INDUCED* werden die gefilterten Knoten vereinigt. Danach wird über die Labels der Kanten iteriert und der Join mit den vereinigten Knoten gebildet. Entsprechend werden bei *EDGE_INDUCED* die Kanten vereinigt und Joins mit den einzelnen Knotenmengen gebildet.

5.4 Grouping

Grouping kondensiert einen logischen Graphen durch strukturale Gruppierung der Knoten und Kanten (siehe Definition in Kapitel 4.2.2). Insbesondere orientiert sich diese Implementierung an einer neuen allgemeineren Variante des Grouping-Operators, die in Gradoop *Keyed Grouping* heißt. Knoten und Kanten werden nach identifizierenden Merkmalen (Grouping-Key) zu Superknoten und

Superkanten zusammengefasst. Diese Grouping-Keys werden durch benutzerdefinierte Grouping-Funktionen extrahiert. Hierbei können für Knoten und Kanten je beliebig viele Funktionen übergeben werden. Beispielsweise kann das Label oder der Wert einer Eigenschaft als Grouping-Key verwendet werden. Falls für Knoten oder Kanten keine Grouping-Funktion übergeben wird, werden mit einer konstanten Funktion alle Elemente zu einem Superelement zusammengefasst.

Alle Knoten mit den gleichen Grouping-Keys werden zu Superknoten aggregiert. Danach werden die Superknoten verwendet, um die Superkanten zu berechnen. Die Kanten verwenden zusätzlich zu den angegebenen Grouping-Keys ihre Start- und Zielsuperknoten als Grouping-Keys. Die ursprünglichen Elemente werden verworfen, sodass nur die Superelemente verbleiben.

Zusätzlich zu den Grouping-Funktionen können Aggregationsfunktionen übergeben werden. Diese werden bei der Gruppierung berechnet und das Ergebnis an die jeweiligen Superelemente geschrieben. Hier kann zum Beispiel die Anzahl der ursprünglichen Elemente pro Superelement berechnet werden oder der maximale Wert einer Eigenschaft.

Eingabe Die Funktionalität des Grouping-Operators wird durch die vom Nutzer übergebenen Funktionen bestimmt. Zusammengefasst besteht die Eingabe des Grouping-Operators aus den folgenden Daten:

- Ein logischer Graph
- Menge von Knoten-Grouping-Funktionen
- Menge von Kanten-Grouping-Funktionen
- Menge von Knoten-Aggregationsfunktionen
- Menge von Kanten-Aggregationsfunktionen

Für die Grouping-Funktionen und Aggregationsfunktionen werden die Typen `KeyFunction` und `AggregationFunction` verwendet. Eine `KeyFunction` enthält einen Spark Spaltenausdruck zur Berechnung des Grouping-Keys und einen Namen. Der Name wird als Spaltenname und als Schlüssel zum Speichern des Grouping-Keys als Eigenschaft genutzt. Die Speicherung des Grouping-Keys an dem Element wird ebenfalls von der `KeyFunction` übernommen, wodurch jeder Grouping-Key die Art der Speicherung selber bestimmen kann. So können die Grouping-Keys nicht nur als Eigenschaft oder Label gespeichert werden, sondern auch beispielsweise von anderen `KeyFunctions` gesetzte Labels und Eigenschaften verändern.

Eine `AggregationFunction` definiert ebenfalls einen Namen und einen Spaltenausdruck. Der Name wird analog zur `KeyFunction` als Schlüssel beim Speichern der Ergebnisse als Eigenschaft verwendet. Der Spaltenausdruck ist ein Aggregationsausdruck, der eine Aggregation von mehreren Werten definiert. Spark beinhaltet vordefinierte Aggregationsausdrücke wie `count`, `min` und `max`, aber es können auch eigene Aggregationsfunktionen implementiert werden. Dazu wird von der abstrakten Klasse `UserDefinedAggregateFunction` (UDAF) geerbt und die zur Aggregation benötigten Funktionen und Datentypen definiert. Da UDAFs mit typenlosen DataFrames verwendet werden, müssen die verwendeten Datentypen für die Eingabe, Ausgabe und den Buffer explizit angegeben werden. Die Funktionen `initialize`, `update`, `merge` und `evaluate` werden für die Berechnung der

Aggregation verwendet. `initialize` initialisiert einen Buffer, `update` aktualisiert den Buffer mit einer Eingabezeile, `merge` kombiniert zwei Buffer und `evaluate` berechnet aus dem finalen Buffer das Ergebnis. Zusätzlich muss angegeben werden, ob die `AggregationFunction` sich deterministisch verhält.

Alternativ gibt es die typisierte Variante von UDAFs, `Aggregator`. Die Definition ist ähnlich zu UDAFs, allerdings werden generische Typ-Parameter verwendet, um die Typen zu bestimmen. `Aggregator` wird mit der neuen Gruppierung der Dataset-API von Spark verwendet, während UDAFs mit DataFrames verwendet werden. Da die Dataset Variante sich nicht mit beliebig vielen Aggregationsfunktionen verwenden lässt, wird auf DataFrames gruppiert und das Ergebnis am Ende zurück in ein Dataset übertragen.

Ablauf Die Berechnung des Grouping-Operators lässt sich in jeweils 4 Schritte für die Knoten- und Kantenmengen aufteilen:

1. Knoten: Berechne die Knoten-Grouping-Keys.
2. Knoten: Gruppiere die Knoten zu Superknoten und aggregiere die Eigenschaften.
3. Knoten: Extrahiere eine Abbildung von ursprünglicher Knoten-Id auf neue Super-Id.
4. Knoten: Erstelle Superknoten und speichere die Aggregationsergebnisse als Eigenschaften.
5. Kanten: Berechne die Kanten-Grouping-Keys.
6. Kanten: Ersetze Start- und Zielknoten durch die entsprechenden Superknoten.
   Nutze hierfür die in Schritt 3 extrahierte Knoten-Id-Abbildung.
7. Kanten: Gruppiere die Kanten zu Superkanten und aggregiere die Eigenschaften.
8. Kanten: Erstelle Superkanten und speichere die Aggregationsergebnisse als Eigenschaften.

Die Abspaltung der Eigenschaften vom Hauptelement beim TFL-Schema lässt sich bei Grouping nicht ausnutzen, da beide Teile zur Berechnung der Grouping-Keys benötigt werden. Auch die Aufteilung nach Label kann programmatisch nicht ausgenutzt werden, weswegen der grundlegende Ablauf gleich ist.

5.4.1 GVE-Schema

Knoten In Schritt 1 werden die Knoten-Grouping-Funktionen berechnet und in eine neue Spalte geschrieben. Da Spark zur Gruppierung nur bereits existierende Spalten als Schlüssel verwenden kann, muss die Erzeugung dieser Spalten separat erfolgen. Für alle übergebenen Grouping-Funktionen werden die Keys zusammen als `Struct` in einer einzelnen Spalte gespeichert. Da dafür eine neue Spalte angelegt wird, werden ab hier `DataFrames` statt `Datasets` verwendet.

Die Gruppierung und Aggregation in Schritt 2 erfolgt mit den von Dataset bereitgestellten Methoden `groupBy` und `agg` (siehe Listing 5.7). `groupBy` benötigt den Namen der Grouping-Key-Spalte als Argument und gibt das Zwischenformat `RelationalGroupedDataset` zurück. Um daraus wieder

ein `DataFrame` zu bekommen, muss `agg` ausgeführt werden. An diese Methode werden Spaltenausdrücke zur Aggregation übergeben, die in den Aggregationsfunktionen enthalten sind. Nach der Gruppierung verbleiben nur die Spalten der Grouping-Keys und Aggregationsergebnisse. Zur eindeutigen Identifikation muss eine neue Id für jeden Superknoten erzeugt werden. Da das direkte Erzeugen von `GradoopIds` in UDFs nicht möglich ist (siehe Abschnitt 5.1.2), muss stattdessen eine Spark-Id erzeugt werden und in eine GradoopId übersetzt werden. Die fehlenden Spalten `graphIds` und `properties` werden in Schritt 4 erstellt.

```
1  var superVertices = verticesWithKeys  // DataFrame (id, label, ..., groupingKeys)
2      .groupBy("groupingKeys")          // RelationalGroupedDataset
3      .agg(vertexAggExpressions: _*)    // DataFrame (groupingKeys, agg1, agg2, ...)
```

Listing 5.7: Gruppierung der Knoten in GVE

Schritt 3 ist für die Extraktion einer Abbildung von Knoten-Id auf Superknoten-Id verantwortlich. Mit einem Join werden die gruppierten Kanten vor und die ungruppierten Kanten nach Schritt 2 verbunden. Durch Verwendung des Grouping-Keys, der in beiden Tabellen vorhanden ist, wird damit eine Abbildung jeder einzelnen Kante zu ihrer Superkante erstellt. Bei einem Verbund einer Tabelle mit einem früheren Stand von sich selbst muss beachtet werden, dass der ältere Stand der Tabelle mit `cache` persistiert wird. Ohne die Persistierung berechnet Spark die Tabelle für jede Seite des Verbundes einzeln, wodurch die ältere Tabelle doppelt berechnet wird. Spark kann die Zwischenergebnisse von Berechnungen nicht mehrfach verwenden, wenn sie nicht manuell persistiert wurden.

Im Vergleich zu Gradoop gibt es einen Unterschied im Ablauf, weil die relationale API den Ablauf von Gradoop nicht direkt abbilden kann. Dort werden Schritte 2 und 3 verbunden, indem die Knoten bei Schritt 2 nur markiert werden. Die markierten Knoten sollen zu Superknoten für alle Knoten der gleichen Gruppe werden. Außerdem wird die Id des markierten Knotens gespeichert und an alle anderen Knoten der Gruppe angehängt. So kann die Abbildung von Knoten-Id auf neue Super-Id in Schritt 3 mit einem einfachen Map-Schritt erzeugt werden und benötigt keine Joins. Um von den markierten Knoten die Superknoten zu extrahieren, wird nur ein Filter-Schritt benötigt. Mit den relationalen Funktionen von Spark SQL ist es nicht möglich, Elemente bei der Gruppierung nur zu markieren. Mit relationalen Funktionen können von einem `RelationalGroupedDataset` die Gruppen nur aggregiert werden.

Für das Erstellen der Superknoten in Schritt 4 sind nun alle Daten vorhanden. Die in Schritt 2 verloren gegangenen Spalten müssen neu angelegt werden. Auch wenn diese Spalten keine Werte enthalten, sind sie notwendig zur Überführung des `DataFrames` zurück in ein `Dataset`. Die in verschiedenen Spalten vorliegenden Aggregationsergebnisse müssen nun als Eigenschafts-Spalte gespeichert werden, die aus einer Abbildung von Schlüssel zu Wert besteht. Zuletzt werden die Grouping-Keys mithilfe der Grouping-Funktion als Label oder Eigenschaft gespeichert. Nach Entfernen nicht mehr benötigter Spalten sind die Superkanten fertig berechnet und können als `Dataset` in einem neuen Graphen zurückgegeben werden.

Kanten Schritt 5 ist analog zur Berechnung der Grouping-Keys von den Knoten in Schritt 1. Die Grouping-Keys der Kanten werden berechnet und in eine neue Spalte geschrieben.

In Schritt 6 wird die in Schritt 3 erzeugte Abbildung verwendet, um die Start- und Zielknoten aller Kanten mit den entsprechenden Superknoten zu ersetzen. Listing 5.8 zeigt diesen Schritt für das GVE-Schema. Hier wird ein Verbund der Kanten `edgesWithKeys` und der Abbildung `vertexIdMap` berechnet, wobei die Startknoten-Id der Kantentabelle der Knoten-Id der Abbildung zugeordnet wird. Diese beiden Spalten werden nun verworfen und die Superknoten-Id wird als Startknoten-Id verwendet. Analog werden die Zielknoten mit einem zweiten Verbund durch Superknoten ersetzt.

```
1  val updatedEdges = edgesWithKeys  // DataFrame
2    .join(vertexIdMap, col("vertexId") === col("sourceId"))  // Update source id
3    .drop("sourceId", "vertexId")
4    .withColumnRenamed("superId", "sourceId")
5    .join(vertexIdMap, col("vertexId") === col("targetId"))  // Update target id
6    .drop("targetId", "vertexId")
7    .withColumnRenamed("superId", "targetId")
```

Listing 5.8: Ersetzung der Start- und Zielknoten durch Superknoten in GVE

Die Gruppierung in Schritt 7 weicht nur in einem Punkt von der Gruppierung der Knoten ab. Anstatt nur auf den Grouping-Keys zu gruppieren, werden zusätzlich die soeben ersetzten Start- und Zielknoten-Ids angegeben. Superkanten werden nicht nur durch die Grouping-Keys, sondern auch durch die verbundenen Superknoten unterschieden. So wird ein gültiger Graph erzeugt, der das Schema des ursprünglichen Graphen darstellt.

Die Erzeugung der Superkanten in Schritt 8 ist analog zur Erzeugung der Superknoten.

5.4.2 TFL-Schema

Die Implementierung für das TFL-Schema ist zu großen Teilen analog zur Implementierung für das GVE-Schema. In diesem Abschnitt werden die Unterschiede zur GVE-Implementierung erläutert. Die Aufspaltungen der Tabellen in TFL können für den Grouping-Operator nicht programmatisch ausgenutzt werden. Deswegen werden die Tabellen der Hauptelemente und Eigenschaften am Anfang des Operators kombiniert und am Ende wieder aufgespalten. Die Aufteilung nach Labels bleibt bis zur Gruppierung in Schritt 2 bzw. 7 bestehen, wird dann aber zu einer einzigen Tabelle kombiniert. Ob der Optimierer die Aufteilung trotzdem sinnvoll ausnutzen kann, wird sich in der Evaluation zeigen (siehe Kapitel 6).

Knoten Schritt 1 ist weitgehend analog zu GVE, allerdings muss die Berechnung der Grouping-Keys für jedes Label einzeln ausgeführt werden (siehe Listing 5.9). Die DataFrames mit Element inklusive Eigenschaften sind in einer `Map` mit dem Label als Schlüssel gespeichert sind und werden mit `mapValues` transformiert.

```
1  val verticesWithKeys = graph.verticesWithProperties.mapValues(df =>  // für jedes DF
2      df.withColumn("groupingKeys", struct(vertexKeyExpressions: _*))  // neue Spalte
3  )  // Map[String, DataFrame]
```

Listing 5.9: Berechnung der Grouping-Keys in TFL

Vor der Gruppierung in Schritt 2 wird die `Map` mit den nach Label aufgeteilten `DataFrames` zu einem einzelnen `DataFrame` vereint. Die Gruppierung selber ist analog zu GVE.

Für die Abbildung der Ids (Schritt 3) wird analog zu GVE ein Verbund der gruppierten Knoten mit den Knoten vor der Gruppierung ausgeführt. Die Knoten sind hier nach Label aufgeteilt, aber die Abbildung von Ids zu Super-Ids soll in einer einzelnen Tabelle vorliegen. Erst wird für jedes Label der Knoten ein Verbund mit den gruppierten Knoten gebildet und dann werden die Ergebnisse vereint.

Da die resultierenden Superknoten am Ende wieder im TFL-Schema vorliegen sollen, müssen sie zur Fertigstellung der Superknoten (Schritt 4) nach Label getrennt werden. Außerdem ist es notwendig, die Eigenschaften und Hauptelemente wieder voneinander zu trennen. Direkt nach der Gruppierung gibt es noch keine Labels, allerdings können beim Hinzufügen der Grouping-Keys auch Labels gesetzt werden. Um weiter dem TFL-Schema zu folgen, muss die Tabelle also wieder nach Label aufgeteilt werden. Da nicht allgemein bekannt ist, welche Labels gesetzt werden, muss erst eine Liste der Labels extrahiert und dann die Tabelle nach dieser Liste aufgeteilt werden (siehe Listing 5.10). Das Extrahieren der Labels erfordert einen teuren `collect`-Schritt, der die verteilt vorliegenden Labels auf einem Knoten vereint. In dem Fall, dass das ursprüngliche Label als Grouping-Key verwendet wird und dieser am Superknoten wieder als Label gespeichert wird, ist bereits von der Eingabe die Liste von Labels bekannt. Dann werden die Labels der Eingabe zum Aufteilen der Superknoten verwendet und eine Extraktion mit `collect` ist nicht notwendig. Danach wird die Tabelle mit Filtertransformationen für jedes Label aufgeteilt.

```
1  val labels = superVertices.select("groupingKeys.someKey").distinct  // neue Labels
2      .as[String].collect    // DataFrame -> Dataset[String] -> Array[String]
3
4  val splitSV = labels.map(l => (l, superVertices    // Label -> (Label, SV)
5      .filter("groupingKeys.someKey" === lit(l))     // filter nach Label
6      .withColumn("label", lit(l))                   // setze Label als Konstante
7  ))  // Map[String, DataFrame]
```

Listing 5.10: Aufteilung der Superknoten nach Label

Kanten Die Berechnung der Grouping-Keys für die Kanten (Schritt 5) ist analog zu den Knoten.

Das Ersetzen der Start- und Zielknoten durch die Superknoten (Schritt 6) ist analog zu GVE. Da die Abbildung der Ids im gleichen Format wie bei GVE vorliegt, muss nur beachtet werden, dass die Kanten nach Label aufgeteilt sind. Für jedes Label muss die Kantenmenge einzeln aktualisiert werden.

Die Gruppierung der Kanten (Schritt 7) benötigt wieder die Vereinigung der nach Label aufgeteilten Kanten zu einem einzelnen `DataFrame`. Der Rest ist analog zu GVE.

Die Erzeugung der Superkanten (Schritt 8) ist analog zur Erzeugung der Superknoten (Schritt 4).

5.5 Mengenoperatoren – Graphen

Die Mengenoperatoren *Combination*, *Exclusion* und *Overlap* modifizieren die Knoten und Kanten in logischen Graphen (siehe Abschnitt 4.2.3). Die Eingabe besteht aus zwei logischen Graphen und das Ergebnis ist ein neuer logischer Graph. *Combination* erzeugt für die Knoten und Kanten jeweils die Vereinigungsmenge, *Exclusion* die Differenzmenge und *Overlap* die Schnittmenge.

5.5.1 Combination

GVE-Schema Für die Implementierung von *Combination* in GVE wird die Vereinigung der Knoten und Kanten gebildet. Die Knoten des linken und rechten Eingabegraphen werden mit `union` vereinigt. Da die Vereinigung bei Spark Duplikate nicht entfernt, müssen sie nach der Vereinigung mit `dropDuplicates("id")` entfernt werden. Das Argument ist dabei der Name der Spalte, die zur Identifizierung der Duplikate verwendet wird. Die Vereinigung der Kanten wird analog gebildet.

TFL-Schema Bei der TFL-Implementierung ist das Berechnen der Vereinigung durch die Aufteilung des Schemas etwas komplizierter. Für Hauptelement und Eigenschaften der Knoten und Kanten wird die Vereinigung auf die gleiche Weise gebildet. Die Aufteilung durch die Label erfordert die paarweise Vereinigung der Datasets pro Label, wobei es zu beachten gilt, dass nicht jedes Label in beiden Eingaben vorkommt. Von den Datasets der Labels, die auf beiden Seiten vorkommen wird eine Vereinigung gebildet, während die restlichen Datasets nur kopiert werden. Für ein Label $t_i \in T_L \cup T_R$ ergibt sich damit:

$$\texttt{result}_{t_i} = \begin{cases} \texttt{left}_{t_i} \cup \texttt{right}_{t_i} & t_i \in T_L \wedge t_i \in T_R \\ \texttt{left}_{t_i} & t_i \in T_L \wedge t_i \notin T_R \\ \texttt{right}_{t_i} & t_i \notin T_L \wedge t_i \in T_R \end{cases}$$

Nach der Vereinigung müssen wieder in allen Datasets die Duplikate mit `dropDuplicates("id")` entfernt werden.

5.5.2 Overlap

GVE-Schema Der *Overlap*-Operator erzeugt die Schnittmengen der Knoten- und Kantenmengen durch einen `LEFT SEMI JOIN` der linken und rechten Eingabe. Als Join-Bedingung müssen die Ids der linken Seite gleich den Ids der rechten Seite sein.

TFL-Schema Für TFL wird ebenfalls der `LEFT SEMI JOIN` verwendet. Die Datasets müssen pro Label paarweise verbunden werden. Im Gegensatz zur Vereinigung in *Combination* müssen nur Labels betrachtet werden, die in beiden Eingaben vorkommen. Es wird über die linke Eingabe iteriert und für jedes Label, das auch in der rechten Eingabe vorkommt, ein `LEFT SEMI JOIN` gebildet. Die linken Datasets der Label, die nicht in der rechten Eingabe vorkommen, werden verworfen.

5.5.3 Exclusion

GVE-Schema *Exclusion* erzeugt die Differenzmenge der Knoten- und Kantenmengen mit einem `LEFT ANTI JOIN` der linken und rechten Eingabe. `LEFT ANTI JOIN` ist ein besonderer Join-Typ in Spark, der einen `LEFT OUTER JOIN` minus die Zeilen eines `INNER JOIN` berechnet. Nur die Zeilen, die in der linken Tabelle, aber nicht in der rechten Tabelle enthalten sind, werden zurückgegeben. Da im Ergebnis nur Zeilen aus der linken Tabelle enthalten sind, kommen auch keine weiteren Spalten hinzu. Als Join-Bedingung müssen die Ids der linken Seite den Ids der rechten Seite entsprechen. Hinterher ist es notwendig, mit dem Hilfsoperator *RemoveDanglingEdges* (siehe Abschnitt 4.2.1) endständige Kanten zu entfernen, deren Start- oder Zielknoten entfernt wurden.

TFL-Schema Für TFL muss wieder der Join für jedes Label durchgeführt werden. Im Gegensatz zu den bisherigen Mengenoperatoren werden alle Einträge der linken Abbildung benötigt. Es wird über die linke Abbildung iteriert und für jedes Label, das auch in der rechten Eingabe vorkommt, ein `LEFT ANTI JOIN` gebildet. Die linken Datasets der Label, die nicht in der rechten Eingabe vorkommen, werden unverändert mit ausgegeben. Danach werden mit dem Hilfsoperator *RemoveDanglingEdges* endständige Kanten entfernt.

5.6 Mengenoperatoren – Graphmengen

Die Mengenoperatoren *Union*, *Difference* und *Intersection* berechnen Mengenoperationen mit den Graphen der Graphmengen (siehe Abschnitt 4.2.4). Die Eingabe besteht aus zwei Graphmengen und das Ergebnis ist eine neue Graphmenge. *Union* erzeugt die Vereinigung der Graphen, *Difference* die Differenz und *Intersection* die Schnittmenge. Für die Berechnung werden jeweils nur die Graphköpfe betrachtet und danach die Knoten und Kanten hergeleitet.

5.6.1 Union

GVE-Schema *Union* funktioniert ähnlich zu *Combination*, allerdings wird die Vereinigung aller Mengen, anstatt nur der Knoten und Kanten, gebildet. Die Graphköpfe, Knoten und Kanten der linken Graphmenge werden mit denen der rechten Graphmenge vereinigt und danach werden die Duplikate mit `dropDuplicates("id")` entfernt.

TFL-Schema Die TFL-Implementierung ist ebenfalls zum großen Teil analog zu *Combination*. Für Graphköpfe, Knoten, Kanten und alle Eigenschaften der linken Graphmenge werden die Datasets pro Label paarweise mit denen der rechten Graphmenge vereinigt. Dabei müssen auch die Labels betrachtet werden, die nur in einer der beiden Eingaben vorkommen.

5.6.2 Intersection

GVE-Schema Die Graphköpfe werden analog zu *Overlap* mit einem `LEFT SEMI JOIN` berechnet. Die Knoten und Kanten werden analog zur formalen Definition über die Graphköpfe hergeleitet. Die als Menge gespeicherten Ids der Graphzugehörigkeit werden mit der SQL-Funktion `explode` in einer neuen Spalte auf einzelne Zeilen aufgeteilt (siehe Listing 5.11). Dabei wird für jeden Wert der Menge eine Zeile erzeugt und der Wert in die neue Spalte eingetragen, während die Werte aller anderen Spalten dupliziert werden. Mit `LEFT SEMI JOIN` werden die Zeilen entfernt, deren GraphId nicht in der Menge der Graphköpfe enthalten ist. Nach Entfernen der neuen Spalte und den duplizierten Zeilen verbleiben die Elemente, die zu den berechneten Graphköpfen gehören.

```
1  val result = elements       // (id, ...)
2      .withColumn("graphId", explode(col("graphIds")))    // (id, ..., graphId)
3      .join(graphs, col("graphId") === graphs("id"), "leftsemi")  // join (id, ...)
4      .drop("graphId").dropDuplicates("id")   // (id, ...)
```

Listing 5.11: Filtern der Elemente mit GraphIds

Das Erstellen neuer Zeilen mit `explode` scheint zwar ineffizient, ist im Vergleich zu der Mengenfunktion `array_contains` allerdings um ein Vielfaches schneller. Die Funktion `array_contains` überprüft, ob ein Element Teil eines Arrays oder einer Menge ist und gibt einen Wahrheitswert zurück. Damit ließe sich in der Join-Bedingung prüfen, ob eine Id der Graphköpfe in der Menge *graphIds* enthalten ist. Dass die Funktion erst seit kurzem in Spark enthalten und bisher nur für die SQL-API verfügbar ist lässt vermuten, dass sie in Zukunft weiter optimiert wird.

TFL-Schema Für die TFL-Implementierung werden grundsätzlich die gleichen Schritte benötigt, allerdings muss wieder die Aufteilung der Tabellen beachtet werden. Für die Berechnungen werden nur die Hauptelemente verwendet, während die Eigenschaften am Ende aus den Elementen hergeleitet werden. Die Berechnung der Graphköpfe wird pro Label analog zu GVE durchgeführt. Für das Herleiten der Knoten und Kanten werden die Graphköpfe zu einer einzelnen Tabelle vereint. Mit dieser Tabelle kann für jedes Label der Knoten und Kanten die Herleitung analog zu GVE durchgeführt werden.

5.6.3 Difference

GVE-Schema Die Graphköpfe werden mit `LEFT ANTI JOIN` der beiden Eingaben berechnet. Das Herleiten der zugehörigen Knoten und Kanten ist analog zu *Intersection*.

TFL-Schema Für jedes Label werden die Graphköpfe wieder mit `LEFT ANTI JOIN` aus den Eingaben berechnet. Hierbei muss beachtet werden, dass die Label der linken Eingabe erhalten bleiben. Label, die ausschließlich in der rechten Eingabe vorkommen, können ignoriert werden. Die Herleitung der zugehörigen Knoten und Kanten ist analog zu *Intersection*.

5.7 Zusätzliche Implementierung

Zusätzlich zu den Graphoperatoren sind weitere Operatoren zum Laden und Speichern der Daten, für Tests und zum Transformieren des Schemas notwendig. Im folgenden Abschnitt werden diese Hilfsoperatoren und weitere Details zur Implementierung erläutert.

5.7.1 Input/Output

Für das Einlesen und Speichern von Daten wurden CSV-Datenquellen und -Datensenken für die CSV-Formate von Gradoop implementiert. Das erste Format besteht aus vier CSV-Dateien. Die Knoten, Kanten und Graphköpfe benötigen jeweils eine Datei und in einer weiteren Datei werden Metadaten gespeichert. Die Elemente werden jeweils von einer Zeile dargestellt, die alle Daten des Elements enthält. Um Speicherplatz zu sparen, werden für jedes Label die Schlüssel und Typen der Eigenschaften extrahiert und als Metadaten gespeichert. Für jedes Label der Knoten, Kanten und Graphköpfe gibt es einen Eintrag mit allen vorkommenden Kombinationen von Eigenschafts-Schlüssel und Typ. Diese Einträge bestimmen, wie die Eigenschafts-Werte der Elemente gespeichert werden. So müssen Schlüssel und Typ nicht für jedes Element einzeln gespeichert werden. Dieses Format wird in Gradoop-Spark für die Speicherung von Graphen im GVE-Schema genutzt.

Das indizierte CSV-Format von Gradoop teilt die Dateien für Knoten, Kanten und Graphköpfe nach Label auf. Für jedes Label gibt es eine eigene Datei. Das Format ist ansonsten identisch zum normalen CSV-Format. Es wird in Gradoop-Spark für die Speicherung von Graphen im TFL-Schema genutzt.

Beim Einlesen von CSV-Dateien übernimmt Spark das Trennen der Felder und bietet verschiedene Möglichkeiten zum Escaping an. Die einzelnen Felder liegen als Strings vor und müssen in die entsprechenden Datentypen übersetzt werden. Bei den Eigenschaften müssen dafür erst die Metadaten mit Informationen über die Schlüssel und Typen gelesen werden. Danach wird ein Parser für den korrekten Typ gewählt und der Schlüssel hinzugefügt.

Entsprechend müssen beim Schreiben die Elemente zurück in Strings übersetzt werden. Bei den Eigenschaften müssen dafür die Metadaten erst aus dem Graph extrahiert werden. Dafür werden bei den Knoten, Kanten und Graphköpfen mit einer Gruppierung auf dem Label die Eigenschaften zu einer Liste aller Kombinationen von Schlüssel und Typ aggregiert. Für jedes Label wird in die Metadaten-Datei ein Eintrag mit dieser Liste geschrieben. Sie wird genutzt, um die Werte der Eigenschaften zu einem String zu kombinieren.

Das indizierte CSV-Format wird auf die gleiche Art geschrieben und gelesen. Durch Beschränkungen verschiedener Dateisysteme gibt es allerdings eine Besonderheit: Da nicht alle Zeichen in Dateinamen erlaubt sind, müssen verbotene Zeichen ersetzt werden. Das kann dazu führen, dass zwei unterschiedliche Label auf dem Dateisystem gleich geschrieben werden. Beispielsweise sind < und > in vielen Systemen nicht erlaubt und müssen ersetzt werden. Dadurch würden die Label „age>60“ und „age<60“ beide in `age_60` resultieren. Diese Labels werden in einer Datei kombiniert, damit sie sich nicht gegenseitig überschreiben. Beim Einlesen wird die Datei wieder in mehrere Labels aufgeteilt.

Spark SQL enthält Lade- und Speicherfunktionen für Datasets, die verschiedene Formate wie JSON, CSV und Parquet unterstützen. Dadurch sind diese Formate besonders leicht zu implementieren und können gut optimiert werden. Bisher sind nur CSV-Quellen und -Senken implementiert, aber die Verwendung von einem Apache Parquet [3] basierten Format könnte das Laden und Speichern von Graphen weiter beschleunigen. Parquet ist ein binäres verteiltes Datenformat des Hadoop Ökosystems mit Unterstützung für *predicate pushdown*. Mit CSV ist predicate pushdown allerdings nicht unterstützt. In der Referenzimplementierung ist die Implementierung von Parquet derzeit in Planung.

5.7.2 Weitere Operatoren

Um in Tests zwei Graphen auf Gleichheit zu überprüfen, wurde der *Equality*-Operator implementiert. Der Operator erzeugt aus den Eingabegraphen je eine kanonische Adjazenzmatrix in Form eines Strings. Diese Strings können direkt miteinander verglichen werden. Das ist keine effiziente Lösung des Isomorphismus-Problems, reicht aber zum Testen der Operatoren mit kleinen Testgraphen. Um die Tests möglichst schnell ausführen zu können, wurde der Operator nicht in Spark, sondern lokal implementiert. Bei kleinen Testgraphen wäre durch den Overhead die Ausführung in Spark langsamer. Bei der Erzeugung der Matrix lassen sich für Knoten, Kanten und Graphköpfe verschiedene Gleichheitskriterien verwenden. Zum Beispiel können nur die Ids oder auch alle Daten inklusive Eigenschaften verglichen werden. Der Operator wurde bisher nur für das GVE-Schema implementiert, während das TFL-Schema zum Testen auf Gleichheit erst in das GVE-Schema transformiert werden muss.

Transformationen zwischen den verschiedenen Schemata werden mit den *GveToTfl*- und *TflToGve*-Operatoren durchgeführt. Vom TFL-Schema zum GVE-Schema müssen die Tabellen mit Vereinigungen und Verbünden kombiniert werden. Bei der Transformation vom GVE-Schema in das TFL-Schema muss eine Liste aller vorhandenen Labels extrahiert werden, um die Tabellen nach Label aufzuteilen. Die Tabellen für jedes Label werden mit `filter` von der Ursprungstabelle extrahiert, wofür die Labels außerhalb von Spark vorliegen müssen. Mit `collect` werden die verteilt vorliegenden Labels zum Hauptprogramm (*driver program*) auf dem Cluster geschickt.

Für die Tests der Operatoren werden einfache Graphen mit der auf Cypher [16] basierenden Sprache GDL [23] definiert. Mit GDL-Strings können Graphen und Graphmengen definiert werden. Knoten werden mit runden Klammern definiert, die ein Label und Eigenschaften beinhalten können. Sie werden durch Kanten verbunden, dessen Label und Eigenschaften in eckigen Klammern stehen. Knoten und Kanten können durch Verwendung von Variablen mehrfach verwendet werden. Das folgende Beispiel zeigt den „Community“ Graphen `g1`, der zwei Knoten und zwei Kanten beinhaltet. Der „Person“-Knoten `v1` wird über die „mitgliedVon“-Kante `e1` mit dem „Forum“-Knoten `v2` verbunden. Von `v2` verläuft eine weitere Kante ohne Label zurück zu `v1`:

```
g1:Community [ (v1:Person {alter:23})-[e1:mitgliedVon]->(v2:Forum)-->(v1) ]
```

Die Eigenschaften unterstützen keine mengenwertigen Typen, aber die unterstützten Typen reichen für den Großteil der Tests aus. Aus dem GDL-String wird eine Graphmenge im GVE-Schema erzeugt, die mit *GveToTfl* in das TFL-Schema transformiert werden kann.

6 Evaluation

In diesem Kapitel wird die Performanz der in Kapitel 5 beschriebene Implementierung des EPGM *Gradoop-Spark* evaluiert. Dabei auftretende Besonderheiten und Probleme werden identifiziert und Lösungen vorgeschlagen.

6.1 Grundlagen

Benchmarks Zur Evaluation der Schemata werden verschiedene Benchmarks pro Operator durchgeführt. Als Vergleich stehen dabei die Referenzimplementierung *Gradoop* und die *Gradoop-Table* Implementierung von Elias Saalmann [35] zur Verfügung. Für eine bessere Übersicht enthalten die Diagramme von der *Gradoop-Table* Implementierung nur das GVE-Schema. In den Diagrammen wird Gradoop-Spark mit GS und Gradoop-Table mit GT abgekürzt.

Unter anderem werden die Laufzeiten und der Speedup bei steigender Parallelität verglichen. Der Speedup $S_p = T_1/T_p$ zeigt, wie gut die Laufzeit mit der Anzahl der Knoten skaliert. Der Speedup für p Knoten wird durch Teilen der Laufzeit T_1 von 1 Knoten durch die Laufzeit T_p von p Knoten berechnet. Lineare Skalierung mit dem Speedup p entspricht dem theoretischen Maximum nach dem amdahlschen Gesetz [20].

Die Skalierung mit der Größe des Graphen (Scaleup) wird mit drei verschiedenen Datensätzen geprüft. Kleine Graphen skalieren bei großem Overhead schlecht, während große Graphen durch die Größe des verfügbaren Speichers begrenzt werden können.

Bei Grouping wird zusätzlich der Einfluss verschiedener Konfigurationen des Operators ermittelt. Die Konfigurationen unterscheiden sich in den verwendeten Grouping-Funktionen und Aggregationsfunktionen.

Cluster Die Benchmarks werden auf dem *dbclu*-Cluster der Abteilung Datenbanken der Universität Leipzig ausgeführt. Von dem Cluster werden für Spark 16 Worker-Knoten mit jeweils 6 Kernen von 2,5 GHz und 40 GB RAM bereitgestellt. Die Knoten sind in einem 1-GBit/s-Netzwerk verbunden. Die Benchmarks werden jeweils mit 1, 2, 4, 8 und 16 Knoten ausgeführt. Da Spark bei der Verteilung nur die Anzahl Kerne betrachtet, kann eine Ausführung mit 1 Knoten zum Beispiel auch auf 2 Knoten mit jeweils 3 Kernen stattfinden. Es wird vereinfachend angenommen, dass dies genau 1, 2, 4, 8 und 16 Knoten entspricht. Mögliche durch die Netzwerklokalität und maximale Bandbreite eines Knotens gegebene Unterschiede werden vernachlässigt. Um den Fehler durch Schwankungen gering zu halten, wird jeder Benchmark dreifach ausgeführt und das arithmetische Mittel verwendet.

Graph	Anzahl Knoten	Anzahl Kanten	Größe
LDBC 1	3,2 M	17,3 M	2,3 GB
LDBC 10	30,0 M	176,6 M	23,6 GB
LDBC 100	282,6 M	1.775,5 M	236,0 GB
GA 10	260 K	16,6 M	1,8 GB
GA 100	1,7 M	147 M	15,7 GB
GA 1000	12,7 M	1,36 B	145,1 GB
GA-SG 10	235 K	10,2 M	1,1 GB
GA-SG 100	1,67 M	101 M	10,8 GB
GA-SG 1000	12,67 M	1,01 B	107,9 GB

Tabelle 6.1: Metriken der Graphen

Graphdaten Die meisten Operatoren werden mit den Graphen des LDBC Social Network Benchmarks [15] evaluiert, die in das CSV-Datenformat von Gradoop transformiert wurden. Der LDBC-Datengenerator erzeugt einen künstlichen sozialen Graph, dessen Größe skaliert werden kann. Der Graph bildet mit 11 verschiedenen Knotentypen und 20 verschiedenen Kantentypen die komplexen Beziehungen eines sozialen Netzes ab. Die wichtigsten Knotentypen sind Person, Schlagwort, Forum, Nachricht, Like, Organisation und Ort. Die Person-Knoten bilden mit Freundschaft-Kanten einen zusammenhängenden Graphen. Jede Person ist in einigen Foren aktiv und in jedem Forum werden Diskussionen durch Bäume von Nachrichten dargestellt. Zusätzlich sind die Nachrichten durch ihre Autoren und Likes verbunden. Die übrigen Knoten- und Kantentypen sind in ähnlicher Weise mit dem Rest verbunden und bilden soziale Beziehungen der realen Welt ab.

Es wird mit verschiedenen Techniken versucht, Metriken und Eigenschaften realer Graphen möglichst genau nachzubilden. Die Verteilungen und Skalierungen der verschiedenen Knoten und Kanten werden an natürliche soziale Netze angepasst. Unter anderem folgen viele der Verteilungen dem Potenzgesetz (engl. power law) [30]. Für optimale Vergleichbarkeit werden die Skalierungen 1, 10 und 100 verwendet, die ebenfalls in anderen Evaluationen des EPGM genutzt wurden. Da der LDBC-Graph nur ein einzelner logischer Graph ist, werden für die Mengenoperatoren benötigte Teilgraphen und Graphmengen aus LDBC hergeleitet. Tabelle 6.1 zeigt die Anzahl der Elemente und den benötigten Speicherplatz von LDBC im CSV-Datenformat von Gradoop.

Der Grouping-Operator verwendet stattdessen Graphen, die aus dem LDBC Graphalytics Benchmark (GA) [21] generiert wurden. Der resultierende Graph ist eine Abwandlung des LDBC, bei dem manche Knoten- und Kantentypen bezüglich der Kommunikation zwischen Personen entfernt wurden. Da diese Typen besonders häufig vorkommen, ist die Verteilung der Typen im Vergleich zum LDBC abgeflacht. Zusätzlich ist die Anzahl der Knoten und Kanten geringer, weswegen die Skalierungen 10, 100 und 1000 verwendet werden. Die Größe von GA 1000 ist etwas mehr als die Hälfte der Größe von LDBC 100. Die Anzahl der Kanten ist ähnlich, während die Anzahl der Knoten um eine Größenordnung geringer ist. Tabelle 6.1 zeigt die Metriken für die verwendeten Graphalytics-Graphen.

Eine Konfiguration des Grouping-Operators verwendet nicht GA, sondern einen Teilgraphen (GA-SG). Dieser Teilgraph enthält nur Knoten mit dem Label „User“ und Kanten mit dem Label „knows“. Da diese Typen den Großteil des Graphen ausmachen, ist GA-SG nur ein bisschen kleiner als GA (siehe Tabelle 6.1).

6.2 Bekannte Probleme

Im Laufe der Arbeit sind mehrere Probleme aufgefallen, die sich auf die Performanz auswirken oder die Implementierung erschweren:

1. Bei mehrfacher Verwendung von Tabellen müssen Zwischenergebnisse persistiert werden. Der Ausführungsplan enthält für jede Verwendung einer Tabelle den ganzen Plan zu Erzeugung dieser Tabelle. Anstatt diese Tabellen einmal zu berechnen und mehrfach zu verwenden, wird die Tabelle mehrfach berechnet. Um das zu verhindern, muss die Tabelle mit `cache` persistiert und damit der Ausführungsplan unterbrochen werden. Es gibt zwar eine *Reuse Subquery* Optimierungsregel für Catalyst, die scheint aber nicht in allen Fällen zu funktionieren. Bei Flink Table gibt es das gleiche Problem, auf das Saalmann in seiner Arbeit [35] ebenfalls gestoßen ist (siehe Abschnitt 3.1).

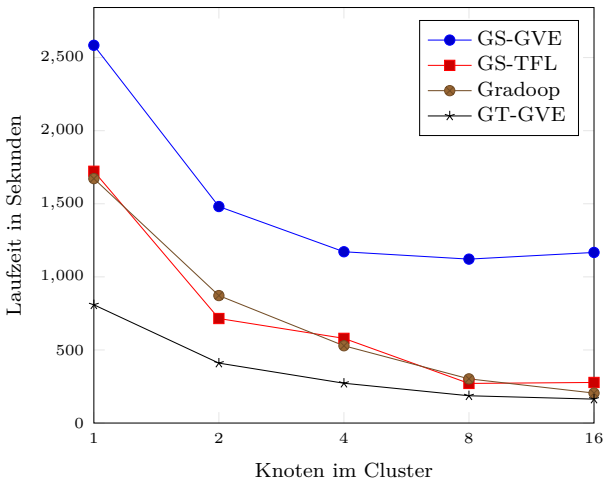
2. In UDFs können keine neuen GradoopIds erzeugt werden. Der Versuch GradoopIds in UDFs zu erzeugen führt zu falschen Ergebnissen, da die Funktion nichtdeterministisch ist, UDFs aber deterministische Funktionen erwarten. Unter gewissen Umständen werden die UDFs aufgrund dieser Annahme mehrfach berechnet, wodurch eigentlich gleiche GradoopIds sich unterscheiden. Um das Problem zu umgehen werden Spark-Ids erzeugt und deterministisch in einzigartige GradoopIds übersetzt (siehe Abschnitt 5.1.2).

3. Probleme mit dem Ablaufplan lassen sich nur schwer auf eine Stelle im Code zurückführen, da hier erst bei der Berechnung Laufzeitfehler geworfen werden, die auf den optimierten Plan verweisen. Der optimierte Plan enthält keine Verweise auf Zeilen im Code und kann sich stark von dem logischen Plan unterscheiden, wodurch die Fehlerdiagnose erschwert wird.

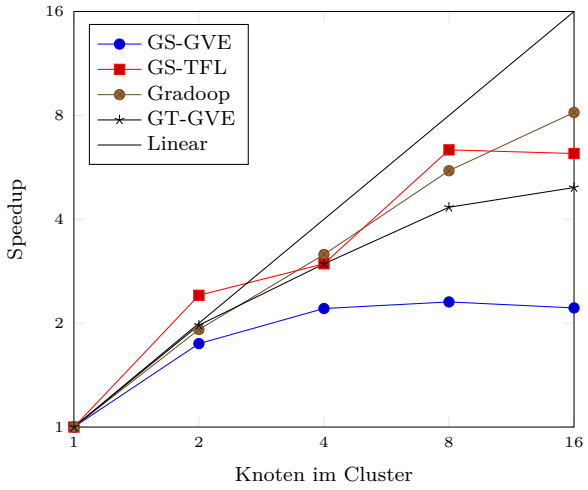
Durch die Verwendung von SQL-artigen Funktionen ist die Syntax relativ vertraut und die dort auftretenden Probleme lassen sich mit Hilfsfunktionen und Übung überwinden. Zum Beispiel führen die Verwendung von Strings zur Auswahl von Spalten und die Verbosität beim TFL-Schema ohne Hilfsfunktionen zu unübersichtlichem Code. Das größte Problem bei der Syntax sind Self Joins. Ein Join einer Tabelle mit sich selbst führt zu Namenskollisionen bei den Spalten, da die internen Bezeichner nicht automatisch geändert werden. Auch wenn vorher beide Seiten des Joins mit Transformationen verändert werden, treten die Kollisionen auf. Wegen dieser Kollisionen vereinfacht der Optimierer die Join-Bedingung fälschlicherweise zu `true` und wirft einen Fehler zur Laufzeit. Da dieser Fehler erst bei der Optimierung auftritt, wird die Diagnose erschwert (siehe Punkt 3). Um den Fehler zu umgehen, müssen die kollidierenden Spalten vor dem Join manuell umbenannt werden.

6.3 Subgraph

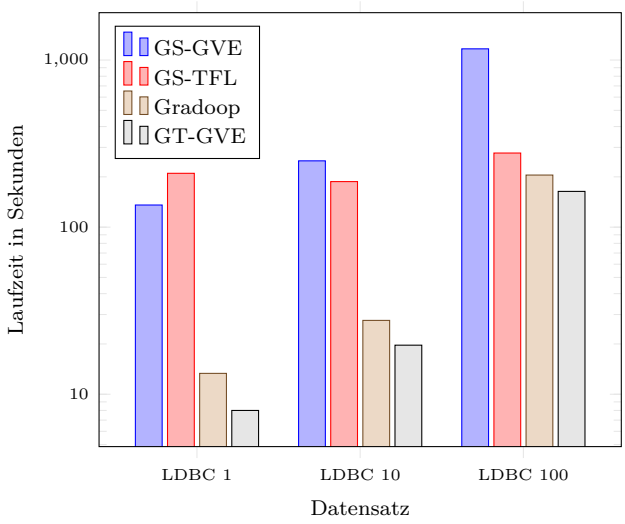
Für die Benchmarks des Subgraph-Operators wird der LDBC-Graph verwendet. Hier wird die *BOTH*-Strategie von Subgraph verwendet, um die Knoten nach dem Label *Person* und die Kanten nach dem Label *knows* zu filtern. Da die *knows*-Beziehung nur zwischen Personen auftritt, entstehen dabei keine endständigen Kanten, die entfernt werden müssten.



(a) Laufzeit, LDBC 100



(b) Speedup, LDBC 100



(c) Scaleup, 16 Knoten

Abbildung 6.1: Evaluationsergebnisse für Subgraph

Abbildungen 6.1a und 6.1b zeigen die Laufzeit und den Speedup von Subgraph mit dem LDBC 100 Graphen. Das GVE-Schema ist dabei signifikant langsamer als die Referenzimplementierung und Gradoop-Table. Bei Betrachtung des Speedups zeigt sich ab 4 Knoten bereits schlechte Skalierung und bei 16 Knoten nimmt der Speedup ab. Das TFL-Schema ist schneller als das GVE-Schema und zeigt ähnliche Laufzeiten wie Gradoop, hat dabei aber große Schwankungen zwischen Benchmarks. Der Speedup ist bei 2 Knoten superlinear und bis zu 8 Knoten nahezu linear. Der Ausreißer bei 2 Knoten und der starke Abfall bei 16 Knoten lassen sich mit den Schwankungen zwischen den Benchmarks erklären. Mit nur drei Benchmarks pro Konfiguration lassen sich bei so hohen Abweichungen keine genauen Mittelwerte ermitteln und die Ursache der Schwankungen könnte auch einen direkten Einfluss auf die Ausreißer haben.

Abbildung 6.1c zeigt, dass Gradoop-Spark schlechter mit kleinen Graphen skaliert als Gradoop und Gradoop-Table. Bei dem GVE-Schema verbessert sich die Laufzeit von LDBC 100 zu LDBC 10 mit 16 Knoten um den Faktor $4,7$. Das TFL-Schema ist mit dem Faktor von $1,5$ noch weiter von dem optimalen Faktor 10 entfernt. Bei nur 1 Knoten kommen beide Schemata mit den Faktoren

Konfiguration	Knoten		Kanten	
	Grouping-Key	Aggregation	Grouping-Key	Aggregation
1	Label	-	-	-
4	Label	count	Label	count
13	city	count	-	count, min(since), max(since)

Tabelle 6.2: Grouping-Konfigurationen

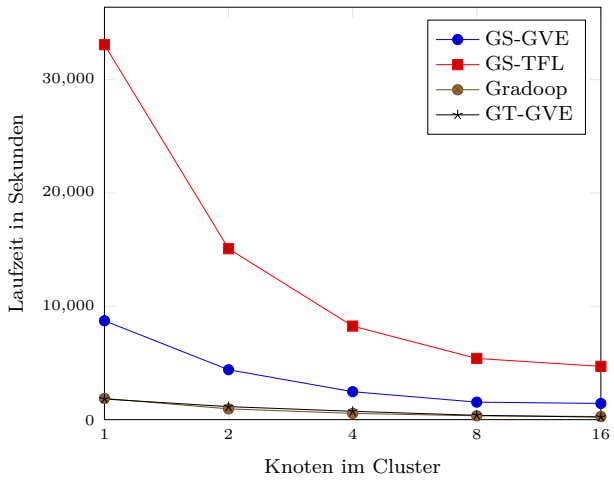
8,7 und 8,4 näher an das Optimum, skalieren aber weiterhin schlechter als die anderen Implementierungen. Spark hat einen besonders hohen Overhead, wodurch geringere Datenmengen schlecht parallelisierbar sind. Der Overhead scheint von hohen Latenzen bei Übertragungen zwischen Knoten zu kommen, die sich durch die vielen Schritten anhäufen. Zum Beispiel brauchen einige Shuffle-Schritte selbst bei LDBC 1 mehrere Minuten, obwohl nur wenige Bytes übertragen werden. Die beim TFL-Schema höhere Anzahl von Tabellen und damit Arbeitsschritten ist vermutlich für einen größeren Overhead und damit die schlechtere Skalierung zuständig.

6.4 Grouping

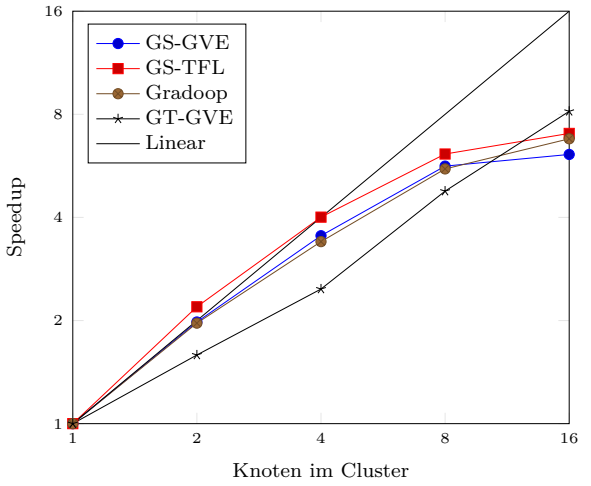
Die Evaluation des Grouping-Operators verwendet die Graphalytics-Graphen und findet in 3 verschiedene Konfigurationen statt. Die Evaluation der Grouping-Referenzimplementierung von Junghans et al. [24] beinhaltet 13 Konfigurationen des Grouping-Operators, von denen Konfigurationen 1, 4 und 13 verwendet werden (siehe Tabelle 6.2). Die gleichen Konfigurationen wurden in Gradoop-Table verwendet, wodurch optimale Vergleichbarkeit gegeben ist. Konfiguration 1 gruppiert nur nach dem Knoten-Label und berechnet keine Aggregationen. Konfiguration 4 gruppiert nach den Knoten- und Kanten-Labels und zählt jeweils die Anzahl der Elemente. Konfiguration 13 gruppiert die Knoten nach der Eigenschaft „city“ und aggregiert dabei die Anzahl. Die Kanten werden nicht weiter gruppiert, aber nach der Anzahl der Elemente und Minimum und Maximum der Eigenschaft „since“ aggregiert. Konfigurationen 1 und 4 verwenden die Graphalytics-Graphen (GA), während für Konfiguration 13 die Teilgraphen (GA-SG) verwendet werden.

Abbildung 6.2 zeigt die Evaluationsergebnisse des Grouping-Operators. Das TFL-Schema hat unabhängig von der Anzahl Knoten im Cluster, der Datenmenge und der Grouping-Konfiguration immer die längste Laufzeit. Das GVE-Schema ist mehr als dreimal so schnell wie das TFL-Schema, kommt aber in keiner Zusammenstellung an die Laufzeiten von Gradoop oder Gradoop-Table heran. Bis 8 Knoten ist bei beiden Schemata der Speedup nahezu linear und fällt ab 16 Knoten ab. Das TFL-Schema hat dabei einen minimal höheren Speedup, der bei 2 Knoten leicht superlinear ist.

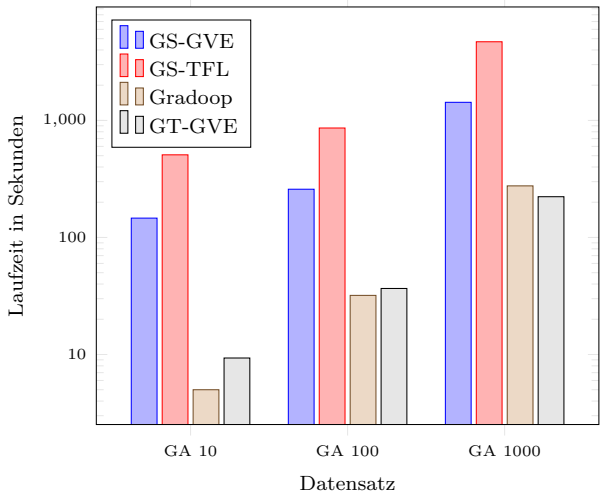
Die schlechte Performanz des TFL-Schemas liegt zum Teil daran, dass die Implementierung ähnlich zur Implementierung des GVE-Schemas ist. Die Performanz des eigentlichen Operators ist gleich, aber zusätzlicher Aufwand ist für die Transformation des Schemas notwendig. Vor der Gruppierung müssen die verschiedenen Tabellen des TFL-Schemas zu je einer Tabelle für die Knoten bzw. Kanten kombiniert werden und nach der Berechnung muss das Ergebnis wieder in das TFL-Schema



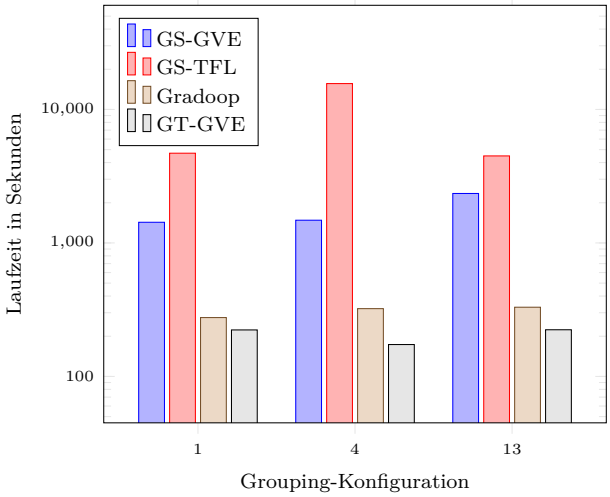
(a) Laufzeit, GA 1000, Konfiguration 1



(b) Speedup, GA 1000, Konfiguration 1



(c) Scaleup, 16 Knoten, Konfiguration 1



(d) Laufzeit, GA(-SG) 1000, 16 Knoten

Abbildung 6.2: Evaluationsergebnisse für Grouping

überführt werden. Das Aufspalten der Tabellen nach dem Label ist dabei besonders teuer. Wegen der Transformation können die Besonderheiten des TFL-Schemas nicht sinnvoll von Catalyst ausgenutzt werden.

Der Vergleich zwischen den verschiedenen Datensätzen in Abbildung 6.2c zeigt schlechte Skalierung mit kleinen Graphen. Das TFL-Schema ist bei allen Datensätzen um den Faktor 3 langsamer als das GVE-Schema, zeigt aber eine ähnlich Skalierung mit der Größe. Beide Schemata verbessern sich von GA 1000 zu GA 100 um den Faktor $5,5$. Das ähnliche Verhalten der Schemata lässt sich mit der ähnlichen Implementierung des Operators begründen. Der Zusatzaufwand beim TFL-Schema scheint für die dreifache Laufzeit zuständig zu sein und ähnlich mit dem Rest des Operators zu skalieren.

Die beiden Schemata zeigen mit verschiedenen Grouping-Konfigurationen unterschiedliches Verhalten (siehe Abbildung 6.2d). Das TFL-Schema braucht bei Konfiguration 4 länger, während das GVE-Schema bei Konfiguration 13 länger braucht. Konfiguration 4 braucht beim TFL-Schema länger, da die Kanten hier nach Label gruppiert werden. Die Labels werden an die resultierenden

Kanten geschrieben, wodurch diese hinterher beim TFL-Schema wieder in mehrere Tabellen aufgeteilt werden müssen. Die Labels müssen zwar nicht aus den Datasets extrahiert werden, da sie vor der Gruppierung bereits bekannt sind, aber trotzdem ist die Aufteilung teuer. Bei Konfiguration 13 wird nach einer Eigenschaft gruppiert, was bei beiden Schemata langsamer ist. Da die Eigenschaften in einem komplexen Typen gespeichert sind, ist der Zugriff teurer als bei dem simplen Typ von Label. Durch den kleineren Graph bei Konfiguration 13 ist der Unterschied gering. Bei dem TFL-Schema ist kein Unterschied zu sehen, da die Transformation des Ergebnisses billiger ist. Es müssen keine Labels an Knoten geschrieben werden und das TFL-Schema muss dadurch nicht in mehrere Tabellen aufgeteilt werden. Der Einfluss der vielen Aggregationen in Konfiguration 13 ist vernachlässigbar, da alle Aggregationen im gleichen Schritt berechnet werden. Die Art und Anzahl der Schritte und deren Verteilung haben einen viel größeren Einfluss auf die Laufzeit, als die Komplexität eines einzelnen Schrittes.

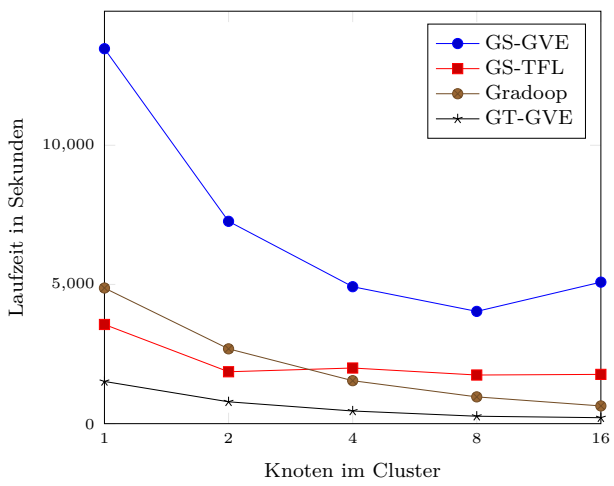
6.5 Mengenoperatoren – Logische Graphen

Die Benchmarks der Mengenoperatoren *Combination*, *Overlap* und *Exclusion* benötigen jeweils zwei logische Graphen. Die verwendeten Graphen sind Teilgraphen der LDBC-Graphen, die vorher mit *Subgraph* erzeugt wurden. Dabei werden die Knoten nach Labels gefiltert und die Kanten mit der *VERTEX_INDUCED*-Strategie hergeleitet. Der erste Teilgraph G_1 enthält die Knoten-Labels *person*, *forum*, *post*, *university* und *city*. Der zweite Teilgraph G_2 enthält die Knoten-Labels *person*, *forum*, *comment*, *tag* und *tagclass*. Durch die Wahl dieser Labels gibt es sowohl bei den Knoten als auch bei den Kanten Überschneidungen zwischen den beiden Teilgraphen. Da die Teilgraphen außerdem nicht identisch sind, gibt keiner der Mengenoperatoren ein leeres Ergebnis zurück.

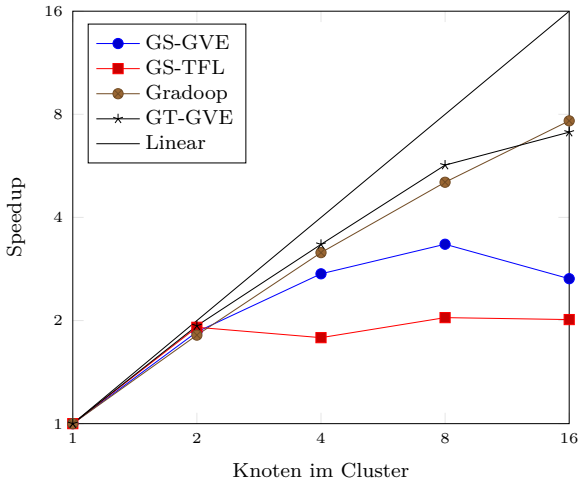
6.5.1 Combination und Overlap

Combination und Overlap zeigen sehr ähnliches Verhalten bei den Benchmarks, weswegen sie hier gemeinsam vorgestellt werden. Combination ist mit allen Implementierungen langsamer als Overlap, aber das Verhältnis zwischen den Implementierungen und die Form der Kurven ist ähnlich. Der einzige Unterschied der Implementierung in Gradoop-Spark ist, dass Combination die Tabellen der beiden Eingaben mit einer Vereinigung kombiniert, während Overlap einen `LEFT SEMI JOIN` verwendet. Combination ist langsamer, da das Ergebnis größer als bei Overlap ist. Die Ausführungspläne der beiden Operatoren sind vom groben Aufbau ähnlich, weisen aber viele Unterschiede im Detail auf.

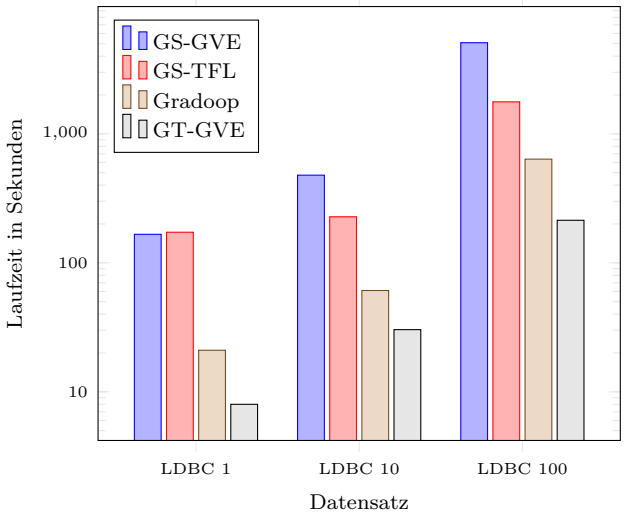
Abbildung 6.3 zeigt exemplarisch die Evaluationsergebnisse des Overlap-Operators auf LDBC 100. Das GVE-Schema ist durchgehend langsamer als das TFL-Schema und skaliert nur bis zu 8 Knoten. Bei 16 Knoten nimmt die Laufzeit wieder zu. Der Speedup ist vergleichsweise schlecht und steigt ebenfalls nur bis zu 8 Knoten. Overlap ist mit dem TFL-Schema bei 1 und 2 Knoten schneller als die Referenzimplementierung, während es bei Combination etwas langsamer ist. Mit mehr als 2 Knoten bleibt die Laufzeit ungefähr konstant und der Speedup bleibt um 2.



(a) Laufzeit, LDBC 100



(b) Speedup, LDBC 100



(c) Scaleup, 16 Knoten

Abbildung 6.3: Evaluationsergebnisse für Overlap

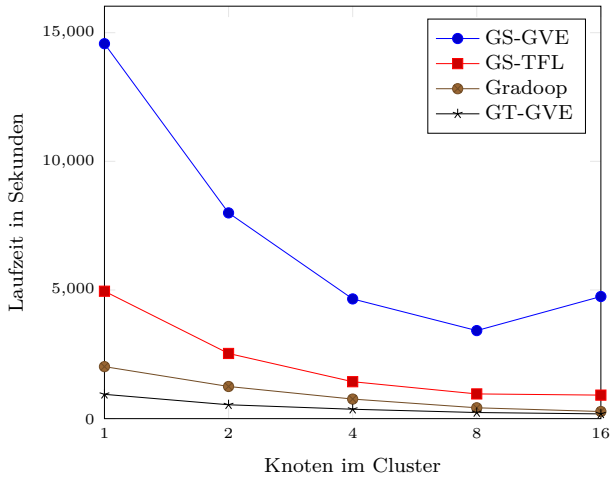
Wenn nicht das Mittel, sondern die einzelnen Benchmarks betrachtet werden, sind beim TFL-Schema ab 4 Knoten teils große Schwankungen sichtbar. Bei manchen Konfigurationen hat der längste Benchmark mehr als die doppelte Laufzeit des kürzesten, weswegen die Mittelwerte mit nur 3 Benchmarks nicht besonders aussagekräftig sind. Die langsamen Ergebnisse könnten durch ungleiche Auslastung der Knoten kommen, während die Arbeit bei den schnelleren Ergebnissen besser verteilt wurde. Da die Tabellen des TFL-Schemas durch die Labels bestimmt werden und die Labels in den meisten Graphen ungleich verteilt sind, sind die Tabellen und Arbeitsschritte verschieden groß. Dadurch wird die gleichmäßige Verteilung auf die Knoten erschwert. Bei manchen Benchmarks ist die Verteilung der Daten und Arbeitsschritte besser und bei anderen schlechter, wodurch es zu den Schwankungen kommt. Das GVE-Schema hat nur wenige große Tabellen, was die gleichmäßige Verteilung erleichtert und Schwankungen verhindert.

Abbildung 6.3c zeigt die Skalierung mit der Graphgröße. Von LDBC 100 zu LDBC 10 hat das GVE-Schema mit einem Faktor von knapp über 10 optimale Skalierung, während das TFL-Schema mit dem Faktor $7,8$ etwas schlechter ist. Die Skalierung ist damit ähnlich zu Gradoop und Gradoop-

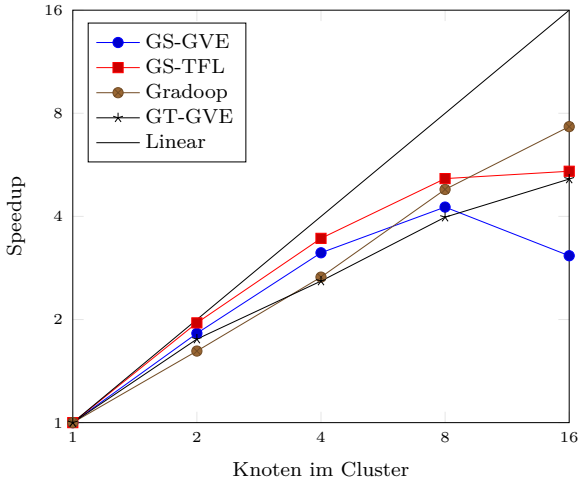
Table. Bei LDBC 1 bricht die Leistung mit Gradoop-Spark allerdings ein, während Gradoop und Gradoop-Table von dem noch kleineren Graphen profitieren können.

6.5.2 Exclusion

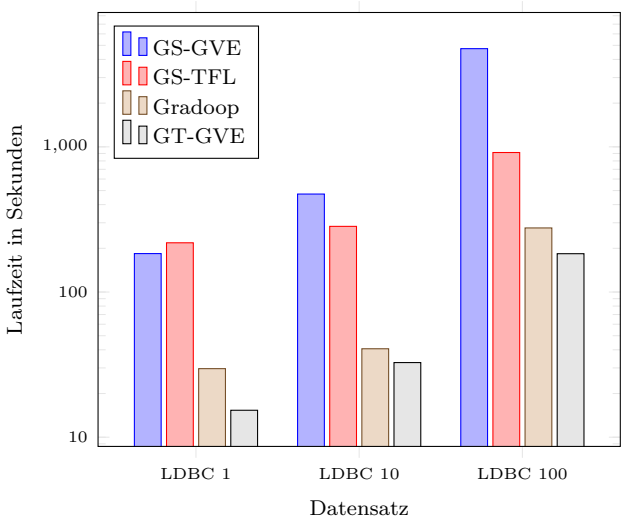
Bei Exclusion müssen nach der Berechnung die endständigen Kanten entfernt werden, was das Verhalten im Vergleich zu Overlap und Combination ändert. Bis auf diesen abweichenden Teil ist der Ausführungsplan identisch zu Overlap.



(a) Laufzeit, LDBC 100



(b) Speedup, LDBC 100



(c) Scaleup, 16 Knoten

Abbildung 6.4: Evaluationsergebnisse für Exclusion

In Abbildung 6.4 sind die Evaluationsergebnisse von Exclusion auf dem LDBC 100 Graphen zu sehen. Das GVE-Schema ist mit der dreifachen Laufzeit langsamer als das TFL-Schema und die Skalierung bricht wieder nach 8 Knoten ein. Das TFL-Schema hat ungefähr die doppelte Laufzeit der Referenzimplementierung und skaliert gut bis 8 Knoten. Die Skalierung beider Schemata ist besser als bei Overlap, während die Laufzeiten relativ zur Referenzimplementierung besonders beim TFL-Schema schlechter sind.

Graphmenge	Knoten	Kanten	Logische Graphen
$\mathcal{G}_1$	27,8 M	27,8 M	48 K
$\mathcal{G}_2$	10,6 M	10,6 M	40 K
$\mathcal{G}_1 \cap \mathcal{G}_2$	9,1 M	9,1 M	24 K

Tabelle 6.3: Metriken der aus LDBC 10 erzeugten Graphmengen

Das Entfernen der endständigen Kanten scheint die Laufzeit zu verschlechtern, aber den Speedup zu verbessern. In dem dafür verwendeten Hilfsoperator *RemoveDanglingEdges* werden die Knoten-Tabellen mehrfach verwendet, ohne sie vorher zu persistieren. Dadurch werden die CSV-Dateien für die Knoten mehrfach eingelesen, was zu der im Vergleich zu Overlap schlechteren Laufzeit beiträgt (siehe Abschnitt 6.2). Durch die größere Datenmenge ist der Operator allerdings besser parallelisierbar und die Werte schwanken weniger zwischen den Benchmarks.

Bei dem Vergleich verschiedener Graphgrößen verhält sich das GVE-Schema ähnlich zu Overlap und hat von LDBC 100 zu LDBC 10 optimale Skalierung (siehe Abbildung 6.4c). Das TFL-Schema skaliert mit einem Faktor von $3,2$ schlechter als bei Overlap.

6.6 Mengenoperatoren – Graphmengen

Die Mengenoperatoren *Union*, *Intersection* und *Difference* benötigen jeweils zwei Graphmengen. Die hier verwendeten Graphmengen $\mathcal{G}_1$ und $\mathcal{G}_2$ wurden von Saalmann [35] für die Evaluation von Gradoop-Table erzeugt. Mit Gradoop wurden die folgenden Schritte ausgeführt, um aus den LDBC-Graphen zwei überschneidende Graphmengen zu erzeugen:

1. Von dem LDBC-Graph wird mit Subgraph ein Teilgraph erstellt. Dieser enthält die Knoten mit den Labels *person*, *post* und *comment* und die Kanten mit dem Label *hasCreator*.

2. Mit einem Operator zur Berechnung von Zusammenhangskomponenten wird der Teilgraph in Komponenten zerlegt und als Graphmenge zurückgegeben. Mit diesem Operator ist jeder Knoten und jede Kante in genau einem logischen Graphen enthalten, wodurch in diesem Fall keine `n:m`-Beziehung von Knoten bzw. Kanten zu den Graphköpfen besteht.

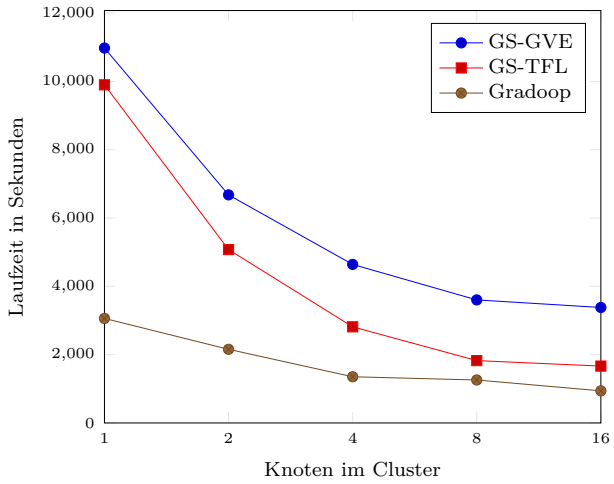
3. Um nun daraus zwei verschiedene überlappende Graphmengen zu erzeugen, werden die logischen Graphen nach dem ältesten enthaltenen *post*-Knoten gefiltert. Die Graphmenge $\mathcal{G}_1$ enthält die logischen Graphen, deren ältester *post*-Knoten vor 2012 erzeugt wurde und $\mathcal{G}_2$ enthält die logischen Graphen, deren ältester *post*-Knoten nach 2010 erzeugt wurde.

Tabelle 6.3 zeigt die Anzahl der Elemente für die aus LDBC 10 erzeugten Graphmengen und dessen Schnitt. Es fällt auf, dass die Anzahl der Knoten und Kanten fast identisch ist und dass jeder logische Graph im Schnitt mehrere Hundert Knoten und Kanten enthält. Sie enthalten jeweils eine Person verbunden mit dessen verfassten Nachrichten, wodurch die Anzahl der *hasCreator*-Kanten bei n Knoten jeweils $n-1$ ist. Die Anzahl an logischen Graphen liegt nicht in für Big Data typischen Größenordnungen, wodurch die Ergebnisse nur geringe Aussagekraft haben. Schlüsse für eine größere Anzahl von logischen Graphen sind schwer zu treffen.

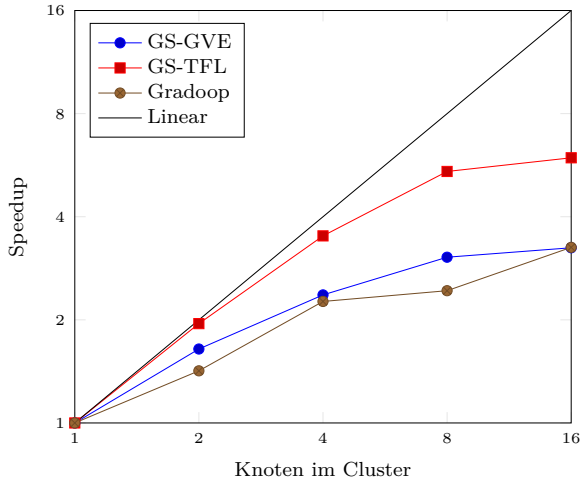
Aufgrund schlechter Skalierung des TFL-Schemas mit kleinen Graphen wird LDBC 100 verwendet, um Besonderheiten besser identifizieren zu können. Da für Gradoop-Table nur Ergebnisse bis LDBC 10 vorliegen, wird Gradoop-Spark hier nur mit der Referenzimplementierung verglichen.

6.6.1 Intersection und Difference

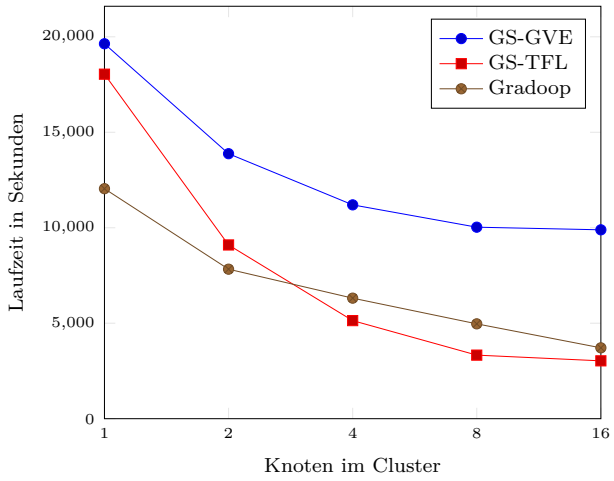
Die Implementierung von Intersection und Difference unterscheidet sich nur in einem Join bei der Berechnung der Graphköpfe. Intersection verwendet `LEFT SEMI JOIN` und Difference `LEFT ANTI JOIN`, die im Ausführungsplan in den gleichen Schritten resultieren. Trotz der gleich aufgebauten Ausführungspläne verhalten sich die Operatoren in den Benchmarks leicht unterschiedlich.



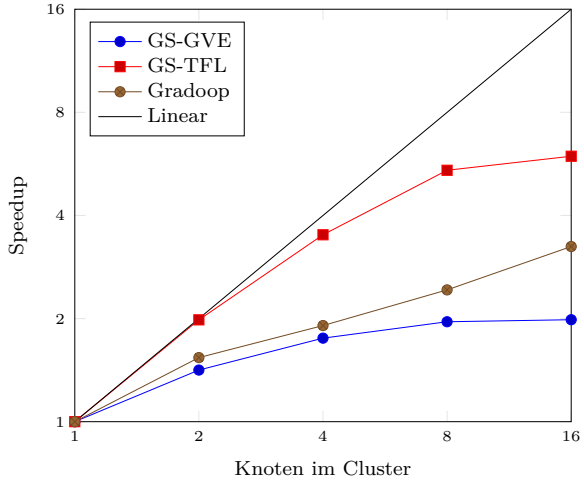
(a) Intersection, Laufzeit, LDBC 100



(b) Intersection, Speedup, LDBC 100



(c) Difference, Laufzeit, LDBC 100



(d) Difference, Speedup, LDBC 100

Abbildung 6.5: Evaluationsergebnisse für Intersection und Difference

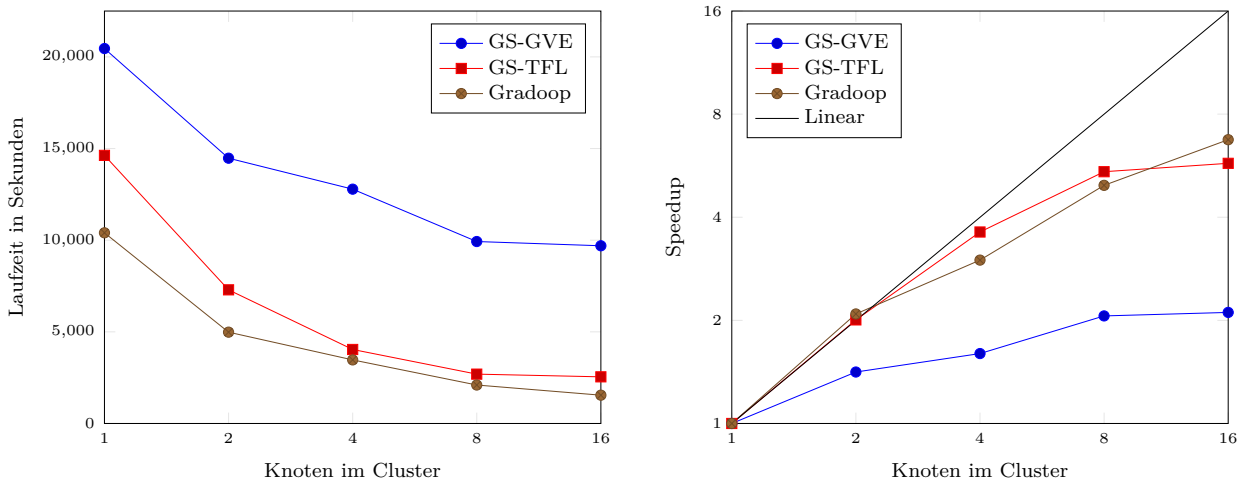
Abbildung 6.5 zeigt die Laufzeiten und den Speedup der Intersection- und Difference-Operatoren. Bei beiden Operatoren startet das GVE-Schema bei 1 Knoten mit nur leicht schlechterer Laufzeit als das TFL-Schema, entfernt sich aber mit zunehmender Knotenzahl aufgrund der schlechteren Skalierung. Bei Difference ist der Speedup des GVE-Schemas schlechter als bei Intersection und erreicht selbst bei 16 Knoten nur den Wert 2. Der Speedup des TFL-Schemas ist bis 8 Knoten

nahezu linear und flacht bei 16 Knoten ab. Da die Referenzimplementierung schlecht skaliert, wird sie bei Difference von dem TFL-Schema ab 4 Knoten überholt.

Die Laufzeit von Difference ist in etwa doppelt so lang wie die von Intersection und skaliert beim GVE-Schema schlechter als beim TFL-Schema. Der Unterschied kommt großteils durch die resultierende Datenmenge und dessen Verteilung zustande. Die resultierende Graphmenge von Difference ist in etwa doppelt so groß wie die von Intersection und die Laufzeit ist etwa doppelt so lang. Die Verteilung zwischen den einzelnen Arbeitsschritten unterscheidet sich beim GVE-Schema allerdings. Bei Difference ist die Berechnung der Knoten und Kanten in etwa gleich lang, während bei Intersection die Berechnung der Kanten stark überwiegt. Möglicherweise führt schlechtere Skalierbarkeit der Knoten zu dem geringeren Speedup des GVE-Schemas bei Difference.

Bei kleinen Graphen skaliert das GVE-Schema wie in den vorherigen Tests besser als das TFL-Schema. Von LDBC 100 zu LDBC 10 haben beiden Operatoren bei dem GVE-Schema optimale Skalierung, während sie sich bei dem TFL-Schema unterscheiden. Bei Intersection verbessert sich die Laufzeit mit dem Faktor $6,3$, während der Faktor bei Difference knapp über dem optimalen Faktor 10 liegt. Da die Laufzeit von Difference etwa doppelt so lang wie die von Intersection ist, hat der Overhead bei Difference einen geringeren Einfluss auf die Skalierung.

6.6.2 Union



(a) Laufzeit, LDBC 100

(b) Speedup, LDBC 100

Abbildung 6.6: Evaluationsergebnisse für Union

Die Evaluationsergebnisse des Union-Operators zeigen in Abbildung 6.6 wieder vergleichsweise schlechte Laufzeiten und Skalierung für das GVE-Schema. Das TFL-Schema kommt teils relativ nah an die Referenzimplementierung dran, bleibt aber durchgehend langsamer. Es skaliert bis zu 8 Knoten sehr gut und flacht mit 16 Knoten ab. Das GVE-Schema zeigt schlechte Skalierung und kommt selbst bei 16 Knoten nur auf einen Speedup von 2. Dabei treten leicht erhöhte Schwankungen zwischen den Benchmarks auf, was untypisch für das GVE-Schema ist. Ähnlich zum TFL-Schema bei Combination und Overlap in Abschnitt 6.5.1 lässt sich das möglicherweise auf ungleich verteilte

Daten und Berechnungen zurückführen. Der Effekt ist hier vermutlich geringer, da das GVE-Schema eine gleichmäßigere Verteilung ermöglicht.

Ähnlich zum Difference-Operator skalieren beide Schemata optimal mit dem kleineren LDBC 10-Graphen. Dies lässt sich mit der vergleichsweise langen Laufzeit begründen, wodurch der Overhead keinen Einfluss auf die Skalierung hat.

6.7 Zusammenfassung

Beide untersuchten Schemata haben generell längere Laufzeiten als die Referenzimplementierung Gradoop und Gradoop-Table. Bei den meisten Operatoren ist das TFL-Schema schneller und hat einen besseren Speedup als das GVE-Schema. Mit 2 Knoten weist das TFL-Schema bei einigen Operatoren superlineare Skalierung auf und liegt auch bei den anderen Operatoren nahe dem linearen Speedup.

Ein Grund für die schlechte Leistung des GVE-Schemas ist die Verteilung der Daten auf dem Cluster. Bei allen Operatoren des GVE-Schemas liegt der Großteil der Knoten und Kanten im letzten Schritt in nur 1–2 Partitionen vor. In einem der Arbeitsschritte werden alle Daten durch ein Shuffle zu einem Knoten übertragen und sind ab dann nicht mehr gleichmäßig verteilt. Die Schritte bis dahin skalieren gut, aber alle Schritte danach skalieren schlecht mit der Anzahl der Knoten, was in geringerem Speedup resultiert. Da dies bei allen Operatoren in einem der letzten Schritte auftritt, ist davon auszugehen, dass die Datensenke verantwortlich ist. Die Laufzeit des Grouping-Operators wird dadurch nicht beeinflusst, da der resultierende Graph und damit die zu speichernden Daten sehr klein sind. Das TFL-Schema ist auch nicht betroffen, da es ein anderes CSV-Format und damit eine andere Datensenke verwendet (siehe Abschnitt 5.7.1). Durch Ersetzung der verantwortlichen Transformation oder einen Hinweis an Catalyst sollte sich die Partitionierung vom GVE-Schema verbessern lassen. Welches Schema nach Behebung dieses Fehlers schneller ist, lässt sich aus diesen Untersuchungen nicht beurteilen.

Beide Schemata haben große Overheads, wodurch sie schlecht mit kleinen Graphen skalieren. Das GVE-Schema verhält sich dabei besser und zeigt zwischen LDBC 100 und LDBC 10 oft optimale Skalierung, was mit dem TFL-Schema nur bei zwei Operatoren der Fall ist. Bei jedem Operator außer Grouping hat das TFL-Schema einen größeren Overhead als das GVE-Schema. Der Overhead entsteht durch hohe Latenzen bei Übertragungen zwischen Knoten und häuft sich durch die vielen Schritte an. Die beim TFL-Schema höhere Anzahl von Tabellen und Arbeitsschritten sorgt für den größeren Overhead. Die Latenzen und sonstige Overheads könnten sich durch die Optimierung der Speicherkonfiguration des Clusters und die Verwendung von dem Ressourcenmanager YARN [39] verringern lassen.

Die Logs zeigen, dass bei allen Operatoren des TFL-Schemas manche der CSV-Dateien doppelt eingelesen werden. Das CSV-Format speichert die Eigenschaften und den Rest des Elemente zusammen in einer Datei. Um die Dateien für das TFL-Schema in mehrere Tabellen aufzuteilen, werden die Eigenschaften und Elemente durch Projektion hergeleitet. Dabei werden die eingelesenen Tabellen

doppelt verwendet, ohne sie vorher zu persistieren. Dieses Problem wurde in Abschnitt 6.2 beschrieben und verursacht das doppelte Einlesen der CSV-Dateien. Durch Behebung dieses Fehlers sollte sich die Laufzeit reduzieren lassen.

Bei beiden Schemata nimmt die Berechnung der Metadaten für die Datensenke einen signifikanten Teil der Laufzeit ein. Außerdem kommt es in der Datensenke zur doppelten Berechnung von Zwischenergebnissen, da sowohl die Metadaten als auch der Graph dort mehrfach verwendet werden. Teile des Operators und die teure Berechnung der Metadaten werden dabei wiederholt. Durch Identifizieren dieser Situationen und Persistieren der Zwischenergebnisse sollte sich die Leistung steigern lassen. Um das Problem zu umgehen, kann ein Dateiformat ohne separate Metadaten verwendet werden. Dieser Ansatz wird von Saalmann [35] mit Gradoop-Table verfolgt und zeigt gute Ergebnisse (siehe Abschnitt 3.1). Alternativ könnten die bereits existierenden Metadaten nach dem Einlesen weiterverwendet werden, anstatt sie zu verwerfen. Jeder Operator müsste zusätzlich zu den Graphen auch die Metadaten transformieren, damit sie für die Datensenke korrekt vorliegen.

7 Fazit und Ausblick

Die folgenden zwei Abschnitte fassen das Ergebnis dieser Arbeit zusammen und schlagen mögliche zukünftige Arbeiten vor.

7.1 Zusammenfassung und Beurteilung

Das Ziel der Arbeit war die Implementierung und Evaluation des EPGM auf Basis von Apache Spark. Die Vor- und Nachteile sollten mit der Referenzimplementierung auf Basis von Apache Flink und Gradoop-Table verglichen werden. Mit dem GVE-Schema und TFL-Schema wurden zwei relationale Darstellungen vom EPGM erarbeitet und mit der Dataset-API implementiert. Für die Evaluation wurde eine Auswahl von Graph-Operatoren für beide Schemata implementiert.

Das EPGM und die Operatoren konnten im vollen Umfang erfolgreich implementiert werden. Mit der Auswahl von *Subgraph*, *Grouping* und den Mengenoperatoren für logische Graphen bzw. Graphmengen ist grundlegende Funktionalität für einfache Graphanalysen vorhanden. Sowohl die neue *Keyed Grouping*-Variante als auch alle Typen der Eigenschaften werden unterstützt. Es wurde darauf geachtet, das System flexibel genug für Erweiterungen des Modells und weitere Schemata zu gestalten. Von Gradoop im CSV-Format gespeicherte Graphen können ohne Transformation eingelesen werden und im gleichen Format geschrieben werden. So kann Gradoop-Spark besonders leicht in Arbeitsabläufe von Gradoop eingebunden werden. Gradoop-Spark bildet eine Grundlage für Integrationen in Systeme wie MLLib zum Machine Learning und Morpheus zum Pattern Matching.

Die Ausdrucksstärke der relationalen Transformationen übersteigt die von typischen relationalen Datenbanken. Mit UDFs können neben den vorgegebenen Funktionen beliebige Map-Schritte berechnet werden, die trotzdem von Catalyst optimiert werden. Die Operatoren ließen sich meist direkt von der Referenzimplementierung in die relationale Spark Dataset-API übersetzen. Für die meisten Operatoren gibt es ein relationales Equivalent in Spark, mit dem der Ablauf in Gradoop nachgebildet werden kann. Einzig beim Grouping-Operator sind zusätzliche Schritte notwendig, weil die Ausdrucksstärke der relationalen API nicht für eine direkte Übersetzung ausreicht. Durch die Verwendung von nicht relationalen Funktionen der Dataset-API ließe sich auch Grouping direkt nachbilden, wobei allerdings die meisten Optimierungen von Spark SQL verloren gingen.

Durch die relationale API und die direkte Unterstützung von SQL ist die Implementierung von Operatoren relativ simpel. Schwierigkeiten bei der Syntax wie Self Joins und die Verbosität des TFL-Schemas lassen sich mit Praxis und Hilfsfunktionen relativ leicht überkommen. Die Optimierung von Operatoren wird allerdings durch einige Punkte erschwert. Dabei fielen besonders die nach manchen Abläufen ungleiche Partitionierung von Tabellen und die Notwendigkeit, mehrfach verwendete Tabellen explizit zu persistieren, auf.

Die Performanz der Implementierung ist meist schlechter als die Referenzimplementierung und Gradoop-Table. Dies lässt sich auf ungleiche Partitionierung der Daten und mehrfach berechnete Tabellen zurückführen. Die Datensenke des GVE-Schemas überträgt zum Beispiel alle Daten auf 1-2

Partitionen, wodurch die Performanz bei großen Ergebnisgraphen leidet. Das TFL-Schema wird von diesem Problem weniger beeinflusst, hat aber mehrere Zwischenergebnisse, die mehrfach berechnet werden. In allen Operatoren außer Grouping ist das TFL-Schema schneller als das GVE-Schema. Ein Großteil des Unterschiedes kommt von den erwähnten Problemen, aber das TFL-Schema scheint auch von der Aufteilung der Tabellen zu profitieren. Wegen dieser Probleme lässt sich von den beiden Schemata kein endgültiger Sieger bestimmen. Mit kleinen Graphen skalieren beide Schemata schlecht, wobei besonders das TFL-Schema einen großen Overhead hat. Der Overhead lässt sich auf hohe Latenzen bei Übertragungen zwischen den Knoten zurückführen.

Spark bietet mit der relationalen Dataset-API eine vielfältige Möglichkeit an, Probleme der Graphanalyse zu bewältigen. Aktuell kann die Performanz von Gradoop-Spark im Vergleich zu der Referenzimplementierung und Gradoop-Table nicht mithalten. Nach der Implementierung weiterer Optimierungen bietet Gradoop-Spark eine Grundlage des EPGM in Apache Spark, mit der Integrationen von anderen Systeme und Frameworks erforscht werden können.

7.2 Zukünftige Arbeiten

Vor dem Hinzufügen weiterer Operatoren und Integrationen sollte die Performanz von Gradoop-Spark verbessert werden. In Kapitel 6 wurden bereits Gründe für die schlechte Performanz erforscht und Vorschläge für Optimierungen präsentiert. Unter anderem sollte auf eine gleichmäßige Partitionierung geachtet, die Anzahl der verwendeten UDFs verringert und die Persistierung von mehrfach verwendeten Tabellen beachtet werden.

Neben der Optimierung der Implementierung ist auch die Optimierung der Cluster-Konfiguration und die Verwendung eines besseren Cluster-Managers wichtig. Um den Overhead von Spark zu verringern sind hierbei besonders Netzwerk- und Speicheroptionen interessant. Auch sollte der Cluster-Manager YARN [39] in Bezug auf die Leistung und den Overhead evaluiert werden.

Die während dieser Arbeit veröffentlichte neue Spark Version 3 enthält Verbesserungen des Catalyst-Optimierers, die die Performanz in Benchmarks im Schnitt verdoppeln [5]. Ein großer Teil der Verbesserung stammt von dem neuen *Adaptive Query Execution*-Framework, das zur Laufzeit Statistiken sammelt und damit den Ausführungsplan optimiert. Eine weitere große Verbesserung kommt von *Dynamic Partition Pruning*, mit dem zur Laufzeit erkannt wird, welche Partitionen für eine Transformation nicht benötigt werden und übersprungen werden können. Viele der kleinen Änderungen erweitern die Funktionalität vorhandener Funktionen und beheben Fehler. Unter anderem wird ein Fehler von Catalyst behoben, der die Wiederverwendung von mehrfach verwendeten Unterabfragen verhinderte. Dadurch könnte die in Kapitel 6 erwähnte Persistierung von mehrfach verwendeten Tabellen in manchen Fällen entfallen. Da sich die API mit der neuen Version nur geringfügig ändert, ist der Wechsel auf Spark 3 kein großer Aufwand. Da Spark 3 erst zum Ende der Arbeit veröffentlicht wurde, wurde es im Rahmen dieser Arbeit nicht untersucht.

Neben den Optimierungen der Implementierung ist auch die Evaluation weiterer Schemata interessant. Dafür bieten sich kleine Variationen der vorhandenen Schemata an, indem zum Beispiel beim TFL-Schema die Elemente und Eigenschaften nicht voneinander getrennt werden. Das *Vertikale Schema* von Gradoop-Table ist ebenfalls ein guter Kandidat (siehe Abschnitt 3.1).

Für mehr als einfache Analyseabläufe sind weitere Operatoren und Integrationen notwendig. Mit Operatoren wie Pattern Matching und ETL-Operatoren kann zum Beispiel die Funktionalität an die Referenzimplementierung angenähert werden. Iterative Graphalgorithmen, für die in Gradoop derzeit Gelly [4] verwendet wird, könnten mit einer Einbindung von GraphX [18] oder GraphFrames [12] implementiert werden (siehe Abschnitt 3.2). Andererseits sind Integrationen in Frameworks wie Morpheus [31] und MLlib [29] nur mit Spark möglich und wären ein Alleinstellungsmerkmal für Gradoop-Spark. Die äquivalenten Frameworks in Flink haben oft weniger Funktionalität oder werden wegen der geringeren Nutzerbasis schlechter unterstützt. Um zum Beispiel ETL-Aufgaben mit Gradoop zu erledigen und dann Frameworks wie MLlib mit Gradoop-Spark zu verwenden, ist es auch sinnvoll den Austausch der Graphen zwischen Gradoop und Gradoop-Spark zu optimieren. Anstelle von einfachen CSV-Dateien auf dem HDFS bieten sich binäre Dateiformate wie Parquet [3] und verteilte Datenbanken wie HBase [17] an. Gradoop beinhaltet bereits eine Integration mit HBase und plant die zukünftige Unterstützung von Parquet.

Aktuell werden mit Gradoop auch temporale Graphen erforscht [33, 34]. Bei temporalen Graphen wird für jedes Element gespeichert, in welchem Zeitraum dieses gültig oder relevant ist. Zusätzlich gibt es verschiedene temporale Operatoren wie Snapshot und temporales Grouping, die diese Zeitwerte in ihren Berechnungen verwenden. Es bietet sich an, die Implementierung des EPGM in Gradoop-Spark ebenfalls um temporale Merkmale zu erweitern.

Ein weiteres aktuelles Forschungsgebiet sind Streaming-Graphen [32, 1], bei denen dauerhaft neue Elemente erzeugt werden. Die Flink DataSet-API von Gradoop unterstützt nur Batch-Verarbeitung bei der Datenmengen abgeschlossen vorliegen und im Gesamten analysiert werden, während die Spark Dataset-API auch Streaming-Daten unterstützt. Hier können mit normalen logischen Plänen sowohl Batch-Daten als auch Streaming-Daten bearbeitet werden, weswegen sich die Evaluation von Gradoop-Spark in Bezug auf Streaming-Graphen anbietet. Interessant sind dabei die Fragen, inwiefern das EPGM-Modell, die Schemata und die Operatoren sich für Streaming-Graphen eignen und welche Änderungen notwendig sind.

Literatur

[1] A. Alkamel. *Distributed Pattern Matching on Graph Streams*. 2020.

[2] Apache Software Foundation. *Apache Flink*. 2014. URL: `https://flink.apache.org/` (besucht am 30. 04. 2020).

[3] Apache Software Foundation. *Apache Parquet*. 2018. URL: `https://parquet.apache.org/` (besucht am 30. 04. 2020).

[4] Apache Software Foundation. *Introducing Gelly: Graph Processing with Apache Flink*. 2015. URL: `https://flink.apache.org/news/2015/08/24/introducing-flink-gelly.html` (besucht am 27. 08. 2020).

[5] Apache Software Foundation. *Spark Release 3.0.0*. 2020. URL: `https://spark.apache.org/releases/spark-release-3-0-0.html` (besucht am 27. 08. 2020).

[6] M. Armbrust u. a. „Scaling spark in the real world: performance and usability“. In: *Proceedings of the VLDB Endowment* 8.12 (2015), S. 1840–1843.

[7] M. Armbrust u. a. „Structured streaming: A declarative API for real-time applications in apache spark“. In: *Proceedings of the 2018 International Conference on Management of Data*. 2018, S. 601–613.

[8] D. Beckett und B. McBride. „RDF/XML syntax specification (revised)“. In: *W3C recommendation* 10.2.3 (2004).

[9] P. Carbone u. a. „Apache flink: Stream and batch processing in a single engine“. In: *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 36.4 (2015).

[10] A. Ching u. a. „One trillion edges: Graph processing at facebook-scale“. In: *Proceedings of the VLDB Endowment* 8.12 (2015), S. 1804–1815.

[11] S. Chintapalli u. a. „Benchmarking streaming computation engines: Storm, flink and spark streaming“. In: *2016 IEEE international parallel and distributed processing symposium workshops (IPDPSW)*. IEEE. 2016, S. 1789–1792.

[12] A. Dave u. a. „Graphframes: an integrated api for mixing graph and relational queries“. In: *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems*. 2016, S. 1–8.

[13] J. Dean und S. Ghemawat. „MapReduce: simplified data processing on large clusters“. In: *Communications of the ACM* 51.1 (2008), S. 107–113.

[14] R. Diestel. „The basics“. In: *Graph Theory*. Springer, 2017, S. 1–34.

[15] O. Erling u. a. „The LDBC social network benchmark: Interactive workload“. In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 2015, S. 619–630.

[16] N. Francis u. a. „Cypher: An evolving query language for property graphs“. In: *Proceedings of the 2018 International Conference on Management of Data*. 2018, S. 1433–1445.

[17] L. George. *HBase: the definitive guide: random access to your planet-size data.* ” O’Reilly Media, Inc.”, 2011.

[18] J. E. Gonzalez u. a. „Graphx: Graph processing in a distributed dataflow framework“. In: *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*. 2014, S. 599–613.

[19] J. E. Gonzalez u. a. „Powergraph: Distributed graph-parallel computation on natural graphs“. In: *Presented as part of the 10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*. 2012, S. 17–30.

[20] J. L. Gustafson. „Reevaluating Amdahl's law“. In: *Communications of the ACM* 31.5 (1988), S. 532–533.

[21] A. Iosup u. a. „LDBC Graphalytics: A benchmark for large-scale graph analysis on parallel and distributed platforms“. In: *Proceedings of the VLDB Endowment* 9.13 (2016), S. 1317–1328.

[22] JCC Consulting, Inc. *Graph Query Language GQL*. 2018. URL: `https://www.gqlstandards.org/` (besucht am 15. 10. 2019).

[23] M. Junghanns. *GDL - Graph Definition Language*. 2017. URL: `https://github.com/s1ck/gdl` (besucht am 19. 08. 2020).

[24] M. Junghanns, A. Petermann und E. Rahm. „Distributed grouping of property graphs with GRADOOP“. In: *Datenbanksysteme für Business, Technologie und Web (BTW 2017)* (2017).

[25] M. Junghanns u. a. „Cypher-based graph pattern matching in GRADOOP“. In: *Proceedings of the Fifth International Workshop on Graph Data-management Experiences & Systems*. 2017, S. 1–8.

[26] M. Junghanns u. a. „GRADOOP: Scalable Graph Data Management and Analytics with Hadoop“. In: (1. Juni 2015). arXiv: `1506.00548v2 [cs.DB]`.

[27] Y. Low u. a. „Graphlab: A new parallel framework for machine learning“. In: *Conference on uncertainty in artificial intelligence (UAI)*. Bd. 20. 2010.

[28] G. Malewicz u. a. „Pregel: a system for large-scale graph processing“. In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. 2010, S. 135–146.

[29] X. Meng u. a. „Mllib: Machine learning in apache spark“. In: *The Journal of Machine Learning Research* 17.1 (2016), S. 1235–1241.

[30] A. Mislove u. a. „Measurement and analysis of online social networks“. In: *Proceedings of the 7th ACM SIGCOMM conference on Internet measurement*. 2007, S. 29–42.

[31] Neo4j, Inc. *Morpheus: Cypher for Apache Spark*. 2016. URL: `https://github.com/opencypher/morpheus` (besucht am 16. 04. 2020).

[32] R. Noureddin. *Distributed Grouping of Property Graph Streams*. 2020.

[33] C. Rost, A. Thor und E. Rahm. „Temporal graph analysis using gradoop“. In: *BTW 2019–Workshopband* (2019).

[34] C. Rost u. a. „Evolution Analysis of Large Graphs with Gradoop“. In: *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Springer. 2019, S. 402–408.

[35] E. Saalmann. „Relationale Abstraktion des EPGM unter Verwendung der Apache Flink Table-API“. Magisterarb. Universität Leipzig, 2019.

[36] S. Salloum u. a. „Big data analytics on Apache Spark“. In: *International Journal of Data Science and Analytics* 1.3-4 (2016), S. 145–164.

[37] K. Shvachko u. a. „The hadoop distributed file system“. In: *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*. Ieee. 2010, S. 1–10.

[38] L. G. Valiant. „A bridging model for parallel computation“. In: *Communications of the ACM* 33.8 (1990), S. 103–111.

[39] V. K. Vavilapalli u. a. „Apache hadoop yarn: Yet another resource negotiator“. In: *Proceedings of the 4th annual Symposium on Cloud Computing*. 2013, S. 1–16.

[40] M. Zaharia u. a. „Apache spark: a unified engine for big data processing“. In: *Communications of the ACM* 59.11 (2016), S. 56–65.

Erklärung

Ich versichere, dass ich die vorliegende Arbeit mit dem Thema:

„Evaluation des EPGM auf Basis von Apache Spark“

selbständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe, insbesondere sind wörtliche oder sinngemäße Zitate als solche gekennzeichnet. Mir ist bekannt, dass Zuwiderhandlung auch nachträglich zur Aberkennung des Abschlusses führen kann.

Leipzig, den 28.09.2020

_____

Timo Adameit