Universität Leipzig

Institut für Informatik
Fakultät für Mathematik und Informatik
Abteilung Datenbanken

Relationale Abstraktion des EPGM unter Verwendung der Apache Flink Table-API

Masterarbeit

vorgelegt von:
Elias Saalmann

Matrikelnummer:
3731902

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

Kurzzusammenfassung

In dieser Arbeit wurde untersucht, wie sich EPGM-Implementierungen bestimmter Operatoren unter Verwendung der Table-API von Apache Flink hinsichtlich Laufzeit und Skalierbarkeit verhalten. Dazu wurden zunächst drei relationale Schemata erarbeitet: Zunächst das GVE-Schema, welches sich an der Referenzimplementierung orientiert. Darüber hinaus das vertikale Schema, in dem Eigenschaften als Tripel und Graphzugehörigkeit als Dupel in relationaler Art in eigenen Tabellen gehalten werden. Abschließend das horizontale Schema, das sich vom vertikalen darin unterscheidet, dass jede Eigenschaft eine eigene Tabelle erhält und die Werte als Dupel abgelegt werden. Es wurden die Operatoren Subgraph, Grouping sowie jeweils drei Mengenoperatoren auf logischen Graphen und Graph Collections für jedes der drei Schemata implementiert. Eine Implementierung besteht dabei i. A. aus einer Folge von relationalen Abfragen. Es wurde gezeigt, dass Operatoren technisch nicht unter ausschließlicher Verwendung der Table-API sinnvoll implementiert werden können. Darüber hinaus wurden von der Abstraktion bedingte geringere Ausdrucksstärke sowie weniger Kontrollmöglichkeiten über die verteilte Ausführung identifiziert. Die Evaluation der Implementierung hat ergeben, dass eine der relational abstrahierten Varianten bei maximaler Parallelität stets schneller als die Referenzimplementierung Gradoop ist. Dies ist insbesondere auf ein besseres Optimierungsverhalten zurückzuführen. Während das GVE-Schema in der Breite bessere Laufzeitergebnisse erzielt hat, erzeugt das horizontale Schema offensichtlich zu viel zusätzlichen E/A-Aufwand bei hoher Parallelität und zeigt so eine deutlich schlechtere Skalierbarkeit als die Referenzimplementierung.

Inhaltsverzeichnis

1 Einleitung

Motivation Die verteilte Analyse großer Graph-Datenmengen wurde und wird maßgeblich an der Universität Leipzig entwickelt und erforscht. Neben einem neuartigen Graphmodell und der Definition von darauf arbeitenden Operatoren wurde der Forschungsprototyp Gradoop entwickelt. Dieser enthält eine Open Source Referenzimplementierung des Datenmodells sowie eine verteilte Implementierung bestimmter Operatoren. Die verteilte Ausführung kann auf einem primär horizontal skalierten Shared-Nothing-Cluster stattfinden. Bei der Implementierung wurde auf das Datenfluss-Programmiermodell unter Verwendung von Apache Flink zurückgegriffen. In dieser Implementierung wurde die *DataSet-API* verwendet, über die ein Workflow an Transformationen erstellt und dessen Ausführung gesteuert werden kann. Zusätzlich zu endlichen verteilten Datensätze unterstützt Apache Flink die Verarbeitung von *Streams*, also von über die Zeit theoretisch unbegrenzt wachsenden Datensätzen. Zur Entwicklung von Workflows auf Streams wird die *DataStream*-API bereitgestellt. Als eine gemeinsame Abstraktion beider Schnittstellen (DataSet-API und DataStream-API) wurde inzwischen die *Table-API* bereitgestellt. Diese abstrahiert die übrigen auf ein relationales Modell und bietet entsprechende Operatoren auf abstrakten Tabellen an. Darüberhinaus wird die relationale Abfragesprache SQL unterstützt.

Ziel der Arbeit In dieser Arbeit soll untersucht werden, ob es hinsichtlich Laufzeit und Skalierbarkeit sinnvoll ist, die Table-API von Apache Flink zur Implementierung des Datenmodells und einiger grundlegender Operatoren heranzuziehen. Dafür müssen zunächst relationale Schemata für das Datenmodell entworfen werden. Darauf aufbauend müssen relationale Abfragen als Implementierung der Operatoren entwickelt werden. Beispielhaft sollen insgesamt acht Operatoren implementiert werden. Die Implementierungen der einzelnen Operatoren soll hinsichtlich Laufzeit und Skalierbarkeit untereinander und mit der Referenzimplementierung verglichen werden.

Aufbau der Arbeit Zunächst werden in den Grundlagen das Datenmodell sowie die Operatoren darauf formal definiert. Daraufhin wird die bestehende Referenzimplementierung konzeptionell zusammengefasst und eine Einführung in die Table-API bzw. die relationale Abstraktion mit Apache Flink gegeben. Nachdem in Kapitel 3 verwandte Arbeiten diskutiert werden, findet im anschließenden Kapitel ein Entwurf dreier relationaler Schemata statt. Daraufhin wird im Hauptteil auf die Implementierung der Operatoren in Form von relationalen Abfragen eingegangen. Vorangestellt sind einige grundlegende Aspekte der Implementierung. In der darauf folgenden Evaluation in Kapitel 6 wird eine Laufzeit- und Skalierbarkeitsanalyse der Implementierungen durchgeführt und die Ergebnisse diskutiert. Diese Arbeit endet mit einem Ausblick auf mögliche weitere Schritte in Forschung und Entwicklung der Thematik.

2 Grundlagen

Im folgenden Kapitel werden zunächst die theoretischen Grundlagen eingeführt, bevor die der Referenzimplementierung Gradoop zugrundeliegenden Konzepte erläutert werden. Anschließend werden die durch die Apache Flink bereitgestellten Schnittstellen und deren Anwendung besprochen. Im weiteren Verlauf seien grundlegendes graphentheoretisches Verständnis bzw. Kenntnis entsprechender Begriffe vorausgesetzt.

2.1 Extended Property Graph Model

Es existieren bereits Graphmodelle, die das theoretische Konzept der Graphen in einem anwendungsorientierten Datenmodell verwenden bzw. enthalten. Ziel ist es zumeist, Sachverhalte aus dem realen Leben möglichst gut abzubilden. Dabei ist es i. A. von Interesse, bestimmte Entitäten, deren Beziehungen untereinander und etwaige Attribute der Entitäten abbilden zu können. Ein graphbasiertes Model stellt das vom World Wide Web Consortium (W3C) entwickelte *Resource Description Framework* (RDF) dar [34]. Dabei werden Fakten als ⟨Subjekt, Prädikat, Objekt⟩-Tripel modelliert, wobei Subjekt und Prädikat eindeutig identifizierbare Ressourcen referenziert und Objekt entweder eine Ressource oder ein Literal sein kann. Fasst man das Tripel als Kante mit Start- und Zielknoten auf, ergibt sich eine Graphstruktur wie in Abbildung 2.1. Die dort gezeigte Instanz enthält z. B. die Fakten *Bob ist eine Person* oder *Die TU Dresden liegt in Dresden*. Das Modell ist sehr generisch - jedes Attribut einer Ressource wird als eigenes Tripel modelliert, wobei das Prädikat stets eine eigene Ressource ist (in der Abbildung vernachlässigt). Bei der Modellierung von möglicherweise weniger strukturierten Daten wird das Modell bzw. die Graphstruktur sehr komplex. Würde man im in der Abbildung gezeigten Bespiel an der Verbindung zweier Personen eine Jahreszahl hinzufügen, müsste man zunächst explizite Ressourcen für jede *Freundschaft* erzeugen und jeden der neuen Knoten mit einer Jahreszahl als Literalknoten verbinden. Als Format für die einheitliche Bereitstellung und den Austausch von Informationen hat sich RDF hingegen im *Semantic Web* durchgesetzt [9].

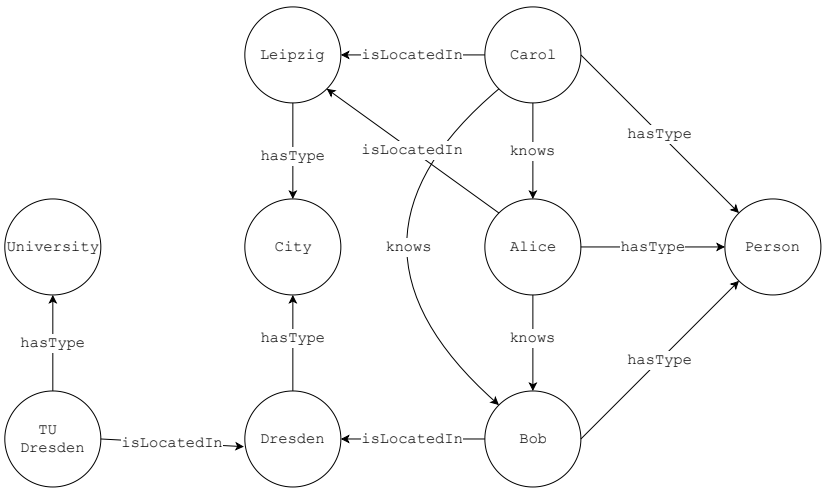


Abbildung 2.1: Beispiel einer RDF-artigen Modellierung

Ein grundlegend anderer Ansatz der Modellierung wurde im *Property Graph Model* (PGM) gewählt [39]. Im Gegensatz zum RDF werden darin die Attribute als Schlüssel-Wert-Paare direkt an den Entitäten gehalten. Diese auch als Eigenschaften (engl. *Property*) bezeichneten Attribute können darüber hinaus auch an den Kanten hinterlegt werden. Für die Modellierung von Attributen ist also nicht stets eine neue Kante erforderlich, die z. B. einen Literal-Knoten mit der Entität verbindet. Weiterhin enthält das PGM einen Kantenbezeichner, i. e. eine Typisierung der Kanten auf Basis von Zeichenketten. Das PGM ist als Multigraph definiert - zwischen zwei Knoten darf folglich mehr als eine Kante existieren. Aufgrund der Flexibilität und Ausdrucksmächtigkeit des PGM findet es in bestehenden Softwaresystemen breite Anwendung. Das populäre Graphdatenbankmanagement-System Neo4j [37] hat das PGM implementiert, wobei in Neo4j zusätzlich auch die Knoten typisiert sind.

Es sei angemerkt, dass es sich beim RDF und PGM formal nur um Modelle handelt - die Implementierung der Datenmodelle in den verschiedenen Softwaresystemen ist nicht vorgegeben. Theoretisch ist es möglich, dass die interne Datenstruktur zweier Implementierungen unterschiedlicher Modelle gleich ist. Vielmehr definiert das formale Modell die Schnittstellen zum Anwender bzw. zu anderen Systemen. Zur Abfrage von Daten in den verschiedenen Graphmodellen wurden verschiedene Abfragesprachen vorgeschlagen: Auf dem PGM erfreut sich z. B. die von Neo4j unterstützte Sprache Cypher [23] großer Verbreitung - für RDF-Systeme hat das W3C die Abfragesprache SPARQL vorgeschlagen [17].

An der Abteilung Datenbanken der Universität Leipzig wurde das PGM in [32] von Junghanns et. al. um *logische Graphen* erweitert. Das Extended Property Graph Model (EPGM) ist wie folgt formal definiert:

Definition Eine EPGM-Instanz ist ein 8-Tupel $\langle V, E, L, \Sigma_B, \sigma, \Sigma_A, A, \gamma \rangle$, wobei

- V eine Menge von Knoten $\{v_i\}$ ist, die durch ihren Index $i \in \mathbb{N}$ identifiziert werden
- E eine Menge von Kanten $\{e_k\}$ ist, die durch ihren Index $k \in \mathbb{N}$ identifiziert werden und für jede Kante $\{e_k = \langle v_i, v_j \rangle \in E\}$ gilt: $v_i, v_j \in V$
- L eine Menge von logischen Graphen $\{G_m\}$ ist, die durch ihren Index $m \in \mathbb{N}$ identifiziert werden und für jeden logischen Graphen $G_m = \langle V_m, E_m \rangle$ gilt: $V_m \subseteq V$, $E_m \subseteq E$ sowie für alle Kanten $e_k = \langle v_i, v_j \rangle \in E_m$ gilt: $v_i, v_j \in V_m$
- Σ_B ein Alphabet aller Bezeichner ist
- $\sigma : (V \cup E \cup L) \to \Sigma_L$ eine Abbildung ist, die jedem Knoten, jeder Kante und jedem logischen Graphen einen Bezeichner zuordnet
- Σ_A ein Alphabet von Schlüsseln für Eigenschaften ist
- A eine Menge aller möglichen Eigenschaftswerte A ist
- $\gamma : (V \cup E \cup L) \times \Sigma_A \to A$ eine Abbildung ist, die einem Tupel aus Knoten (oder Kante oder logischem Graph) und Eigenschaftsschlüssel einen Eigenschaftswert zuordnet

Offensichtlich entspricht eine EPGM-Instanz mit nur einem logischen Graphen im wesentlichen dem PGM. Folglich lässt sich jede PGM-Instanz in das EPGM überführen. Existieren mehrere logische Graphen, kann man sich ein PGM mit verschiedenen, möglicherweise überlappenden, logischen Bereichen vorstellen. Die logischen Graphen referenzieren jeweils Knoten und Kanten einer gemeinsamen Grundmenge. Darüber hinaus können logische Graphen sowohl Bezeichner als auch Eigenschaften haben. Betrachte dazu Abbildung 2.2. Die Instanz besteht aus einer Grundmenge von 9 Knoten und 12 Kanten. Der logische Graph G_1 enthält die Knoten v_1 bis v_6 sowie die Kanten $e_1, e_2, e_4, e_5, e_7, e_{11}$ und e_{12}. Einzeln und ohne den Bezeichner `Geo View` und die Eigenschaft `edgeCount` könnte G_1 auch eine PGM-Instanz darstellen. Die EPGM-Instanz enthält noch einen weiteren logischen Graphen (G_2). Wie in der Abbildung ersichtlich, überlappt sich dieser mit G_1 in den Elementen v_4, v_5, v_6, e_4, e_5 und e_8.

G_1 entspricht im Wesentlichen dem in Abbildung 2.1 gezeigten RDF-artig modellierten Sachverhalt. Offensichtlich genügen im EPGM bzw. PGM deutlich weniger Knoten und Kanten - darüber hinaus lässt sich die Eigenschaft `since` problemlos an den mit `knows` bezeichneten Kanten hinzufügen.

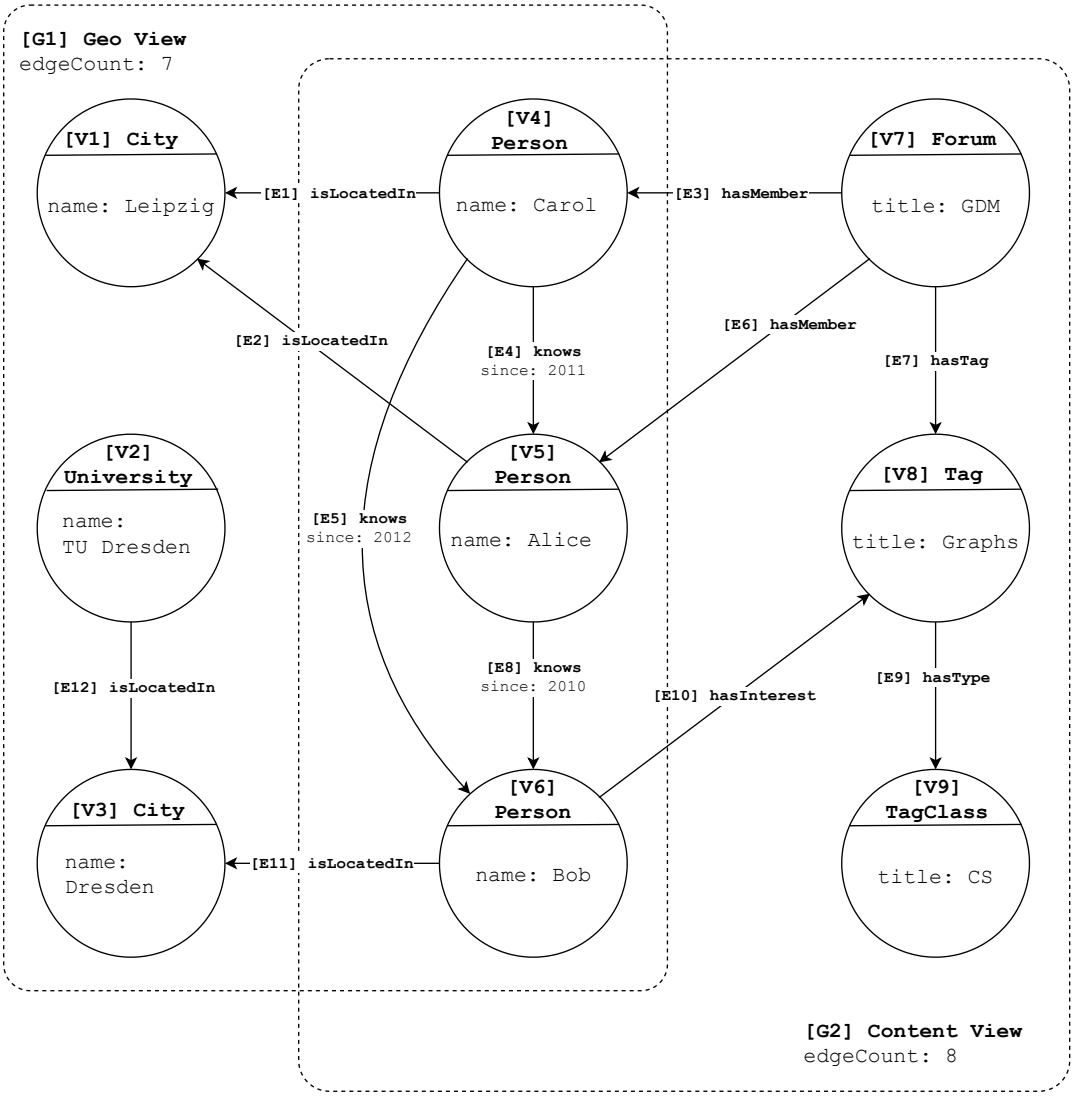


Abbildung 2.2: Beispiel-Instanz im EPGM

2.2 Operatoren

Wie bereits im vergangenen Abschnitt erläutert, kann auf Basis eines Graphmodells definiert werden, mit welchen Schnittstellen Anwender oder Systeme mit der Implementierung des Graphmodells interagieren bzw. die Instanz-Daten verarbeiten (vgl. Cypher). So ließe sich z. B. eine Schnittstelle definieren, mit der man die Überlappung zweier EPGM-Instanzen abfragen kann. In [32] haben Junghanns et. al. eine umfassende Sammlung an Schnittstellen definiert. Jede führt einen *Operator* aus, der z. B. auf einer gesamten EPGM-Instanz oder einem logischen Graphen definiert ist. Darüber hinaus wurden Schnittstellen bereitgestellt, mit denen nutzerdefinierte (domänenspezifische) Operatoren ausgeführt werden können. Im folgenden Abschnitt werden die im Rahmen dieser Arbeit behandelten Operatoren formal definiert, wobei zunächst jegliche Implementierungsaspekte ausgelassen werden.

2.2.1 Subgraph

Der Subgraph-Operator dient dazu, einen spezifischen Teilgraph G' aus einem logischen Graphen G zu gewinnen. In [32] wird der Operator basierend auf Prädikatsfunktionen

$$\Phi_v : \mathcal{V} \rightarrow \{true, false\}$$
$$\Phi_e : \mathcal{E} \rightarrow \{true, false\}$$

definiert:

$$V(G') = \{v \mid v \in V(G) \wedge \Phi_v(v) = true\}$$
$$E(G') = \{\langle v_i, v_j\rangle \mid \langle v_i, v_j\rangle \in V(G) \wedge \Phi_e(\langle v_i, v_j\rangle) = true \wedge v_i, v_j \in V(G')\}$$

Offensichtlich wird die Knoten- und Kantenmenge jeweils auf eine Teilmenge reduziert und sichergestellt, dass Start- und Zielknoten aller Kanten Teil der reduzierten Knotenmenge sind. Diese Mengen werden im weiteren Verlauf der Arbeit als *reduziert* bezeichnet. Die Prädikatsfunktionen Φ_v und Φ_e sind nutzerdefiniert.

Beispiel Sei G der in Abbildung 2.2 gezeigte logische Graph G_1. Wähle die Prädikatsfunktionen wie folgt:

- $\Phi_v(v) := \begin{cases} \texttt{true} & \sigma(v) = \texttt{person} \\ \texttt{false} & \text{sonst} \end{cases}$
- $\Phi_e(e) := \begin{cases} \texttt{true} & \sigma(e) = \texttt{knows} \\ \texttt{false} & \text{sonst} \end{cases}$

Dann besteht resultierend Teilgraph G' aus den Knoten v_4, v_5 und v_6 sowie den Kanten e_4, e_5 und e_8.

2.2.2 Mengenoperatoren auf logischen Graphen

In [32] werden drei mengentheoretische binäre Operatoren auf logischen Graphen definiert. Ergebnis dieser Operatoren ist jeweils ein logischer Graph G'. In den nachfolgenden Beispielen seien logische Graphen vereinfacht durch bezeichnete Knoten sowie einfache gerichtete Kanten dargestellt. Die in Abbildung 2.3 gezeigten Graphen G_1 und G_2 sind exemplarisch Eingaben für binäre Operatoren auf logischen Graphen.

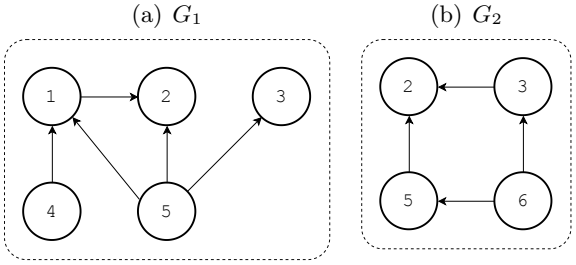


Abbildung 2.3: Beispiel-Graphen G_1 und G_2 als Eingabe der Mengenoperatoren auf logischen Graphen

Combination Der *Combination*-Operator beruht auf der Vereinigungsmenge. Für zwei logische Graphen G_1 und G_2 ist die Combination $G' = G_1 \cup G_2$ definiert als

$$V(G') = V(G_1) \cup V(G_2)$$
$$E(G') = E(G_1) \cup E(G_2)$$

Ergebnis des Operators ist ein logischer Graph, der alle Knoten und Kanten ohne Duplikate aus beiden Eingabegraphen enthält. Betrachte die beiden Eingabegraben aus Abbildung 2.3. Der Operator-Aufruf erzeugt den in Abbildung 2.4 gezeigten Graphen.

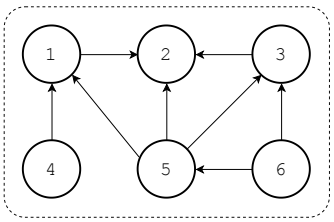


Abbildung 2.4: Ergebnis der Operation $G_1 \cup G_2$

Overlap Der *Overlap*-Operator beruht auf der Schnitt-Menge. Für zwei logische Graphen G_1 und G_2 ist der Overlap $G' = G_1 \cap G_2$ definiert als

$$V(G') = V(G_1) \cap V(G_2)$$
$$E(G') = E(G_1) \cap E(G_2)$$

Ergebnis des Operators ist ein logischer Graph, der nur Knoten und Kanten enthält, die sowohl in G_1 als auch in G_2 enthalten sind. Das vorangegangene Beispiel fortführend erzeugt der Overlap-Operator den in Abbildung 2.5 gezeigten Graphen.

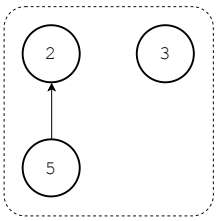


Abbildung 2.5: Ergebnis der Operation $G_1 \cap G_2$

Exclusion Der *Exclusion*-Operator beruht auf der Differenzmenge. Für zwei logische Graphen G_1 und G_2 ist die Exclusion $G' = G_1 \setminus G_2$ definiert als

$$V(G') = V(G_1) \setminus V(G_2)$$
$$E(G') = \{\langle u, v\rangle \in E(G_1) \mid u, v, \in V(G')\}$$

Der Operator berechnet die Differenzmenge der Knoten und den daraus induzierten Teilgraphen von G_1. Betrachte wieder das Beispiel in Abbildung 2.3. Die Knoten 2, 3 und 5 sind in der Schnittmenge enthalten und müssen somit aus der Knotenmenge entfernt werden. Alle mit diesen Knoten verbundenen Kanten ($\langle 1, 2\rangle$, $\langle 5, 1\rangle$, $\langle 5, 2\rangle$ und $\langle 5, 3\rangle$) müssen nun noch entfernt werden, um das in Abbildung 2.6 gezeigte Ergebnis zu erhalten.

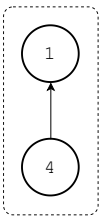


Abbildung 2.6: Ergebnis der Operation $G_1 \setminus G_2$

2.2.3 Mengenoperatoren auf EPGM-Instanzen

Ähnlich zu den Mengenoperatoren wurden in [32] entsprechende binäre Operatoren auf zwei EPGM-Instanzen $\mathcal{G}_1$ und $\mathcal{G}_2$ eingeführt. Ergebnis eines solchen Operators ist eine neue EPGM-Instanz $\mathcal{G}'$. Die Instanzen sind nachfolgend aus Gründen der Übersichtlichkeit ohne Eigenschaften und Kantenbezeichner dargestellt. Abbildung 2.7 zeigt beispielhaft zwei mögliche Eingabe-Instanzen.

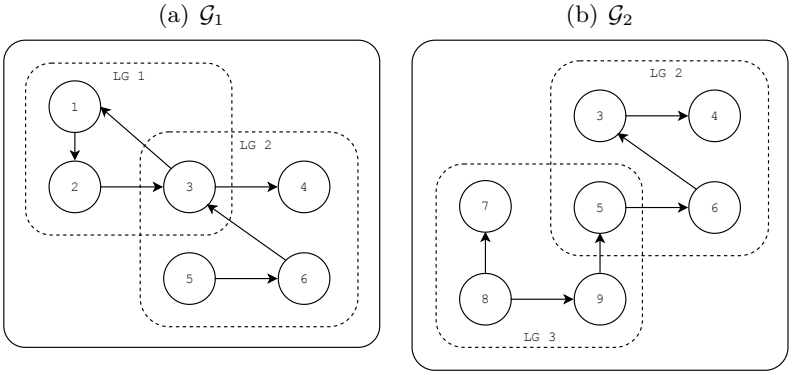


Abbildung 2.7: Beispiel Ein- und Ausgaben für Mengenoperatoren auf EPGM-Instanzen

Union Der *Union*-Operator beruht auf der Vereinigungsmenge. Für zwei Instanzen $\mathcal{G}_1$ und $\mathcal{G}_2$ ist die Vereinigung $\mathcal{G}' = \mathcal{G}_1 \cup \mathcal{G}_2$ folgendermaßen definiert:

$$L(\mathcal{G}') = L(\mathcal{G}_1) \cup L(\mathcal{G}_2)$$
$$V(\mathcal{G}') = V(\mathcal{G}_1) \cup V(\mathcal{G}_2)$$
$$E(\mathcal{G}') = E(\mathcal{G}_1) \cup E(\mathcal{G}_2)$$

Der Operator vereinigt sowohl logische Graphen, als auch Knoten und Kanten beider Instanzen. Die Vereinigung der in Abbildung 2.7 gezeigten Instanzen besteht folglich aus den Graphen $LG1$, $LG2$ und $LG3$, wie in Abbildung 2.8 gezeigt.

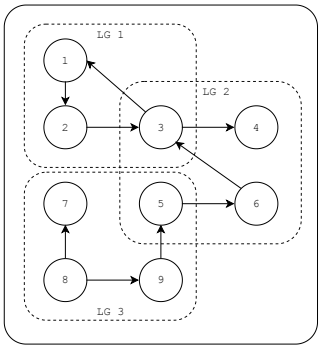


Abbildung 2.8: Ergebnis der Operation $\mathcal{G}_1 \cup \mathcal{G}_2$

Intersection Der *Intersection*-Operator beruht auf der Schnittmenge. Für zwei Instanzen $\mathcal{G}_1$ und $\mathcal{G}_2$ ist der Schnitt $\mathcal{G}' = \mathcal{G}_1 \cap \mathcal{G}_2$ folgendermaßen definiert:

$$L(\mathcal{G}') = L(\mathcal{G}_1) \cap L(\mathcal{G}_2)$$
$$V(\mathcal{G}') = \bigcup_{G \in L(\mathcal{G}')} V(G)$$
$$E(\mathcal{G}') = \bigcup_{G \in L(\mathcal{G}')} E(G)$$

Der Intersection-Operator selektiert alle logischen Graphen, die in beiden Eingaben enthalten sind. Anschließend wird die gemeinsame Grundmenge an Knoten und Kanten auf diejenigen reduziert, die in den verbliebenen logischen Graphen enthalten sind. Betrachte wieder das Beispiel in Abbildung 2.7. Offensichtlich ist $LG2$ der einzige in beiden Instanzen enthaltene logische Graph. Folglich besteht das Ergebnis aus $LG2$ bzw. den darin enthaltenen Knoten und Kanten (siehe Abbildung 2.9).

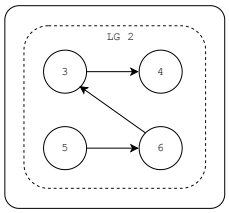


Abbildung 2.9: Ergebnis der Operation $\mathcal{G}_1 \cap \mathcal{G}_2$

Difference Der *Difference*-Operator beruht auf der Differenz. Für zwei Graph Collections $\mathcal{G}_1$ und $\mathcal{G}_2$ ist die Differenz $\mathcal{G}' = \mathcal{G}_1 \setminus \mathcal{G}_2$ folgendermaßen definiert:

$$L(\mathcal{G}') = L(\mathcal{G}_1) \setminus L(\mathcal{G}_2)$$
$$V(\mathcal{G}') = \bigcup_{G \in L(\mathcal{G}')} V(G)$$
$$E(\mathcal{G}') = \bigcup_{G \in L(\mathcal{G}')} E(G)$$

An dieser Stelle wird zunächst die Differenzmenge der logischen Graphen gebildet. Darauf aufbauend werden Knoten- und Kantenmenge analog zum Intersection-Operator reduziert. Das Beispiel fortführend enthält die Differenz den logischen Graphen $LG1$. Die resultierende Instanz ist in Abbildung 2.10 dargestellt.

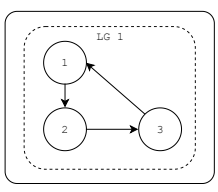


Abbildung 2.10: Ergebnis der Operation $\mathcal{G}_1 \setminus \mathcal{G}_2$

2.2.4 Grouping

Der Grouping-Operator [31] berechnet für einen (nicht-leeren) logischen Graphen G seinen *zusammenfassenden* Graph. Dieser wird nachfolgend formal definiert. Sei

- $\langle V, E, L, \Sigma_B, \sigma, \Sigma_A, A, \gamma \rangle$ eine EPGM-Instanz
- $G = \langle \tilde{V}, \tilde{E} \rangle \in L$ ein logischer Graph mit nicht-leerer Knotenmenge, dessen zusammenfassender Graph berechnet werden soll
- $K_v \subseteq \Sigma_A$ eine nicht-leere Menge an Eigenschaftenschlüsseln zur Gruppierung der Knoten
- $K_\epsilon \subseteq \Sigma_A$ eine Menge an Eigenschaftenschlüsseln zur Gruppierung der Kanten
- $\Lambda_v = \{\alpha_v : \mathcal{P}(\tilde{V}) \to A\}$ eine Menge von assoziativen und kommutativen Aggregatfunktionen, die jeweils einer Menge von Knoten einen Eigenschaftswert zuordnet
- $\Lambda_\epsilon = \{\alpha_\epsilon : \mathcal{P}(\tilde{E}) \to A\}$ eine Menge von assoziativen und kommutativen Aggregatfunktionen, die jeweils einer Menge von Kanten einen Eigenschaftswert zuordnet
- $f_\alpha : \Lambda_v \cup \Lambda_\epsilon \to \Sigma_A$ eine Abbildung, die jeder Aggregatfunktion einen Eigenschaftenschlüssel zuordnet

Dann ist $G' = \langle V', E' \rangle$ der zusammenfassende Graph von G, wenn gilt:

- $\forall u, v \in \tilde{V}$:

$$s_v(u) = s_v(v) \Leftrightarrow \forall k \in K_v : \gamma(u, k) = \gamma(v, k) \tag{2.1}$$

  wobei $s_v : \tilde{V} \to V'$ eine surjektive Abbildung ist, die jedem Konten aus G einen sogenannten Superknoten zuordnet

- $\forall v \in \tilde{V}, \forall k \in K_v$:

$$\gamma(v, k) = \gamma(s_v(v), k) \tag{2.2}$$

- $\forall v' \in V', \forall \alpha_v \in \Lambda_v$:

$$\gamma(v', f_\alpha(\alpha_v)) = \alpha_v(\{v \in \tilde{V} \mid s_v(v) = v'\}) \tag{2.3}$$

- $\forall \langle u, v \rangle, \langle s, t \rangle \in \tilde{E}$:

$$s_\epsilon(\langle u, v \rangle) = s_\epsilon(\langle s, t \rangle) \Leftrightarrow s_v(u) = s_v(s) \wedge s_v(v) = s_v(t) \tag{2.4}$$

$$\wedge \forall k \in K_\epsilon : \gamma(\langle u, v \rangle, k) = \gamma(\langle s, t \rangle, k) \tag{2.5}$$

  wobei $s_\epsilon : \tilde{E} \to E'$ eine surjektive Abbildung ist, die jeder Kante aus G eine sogenannte Superkante zuordnet

- $\forall e \in \tilde{E}, \forall k \in K_\epsilon$:

$$\gamma(e, k) = \gamma(s_\epsilon(e), k) \tag{2.6}$$

- $\forall e' \in E', \forall \alpha_\epsilon \in \Lambda_\epsilon$:

$$\gamma(e', f_\alpha(\alpha_\epsilon)) = \alpha_\epsilon(\{e \in \tilde{E} \mid s_\epsilon(e) = e')\}) \tag{2.7}$$

Anschaulich sind in einem zusammenfassenden Graph die Knoten nach bestimmten Eigenschaftswerten gruppiert (2.1), d. h. zwei Knoten sind genau dann ein und demselben Superknoten zugeordnet, wenn alle Werte der zur Gruppierung herangezogenen Eigenschaften (K_v) übereinstimmen. Die Menge der Superknoten entspricht dem Bild der surjektiven Funktion s_v. Diese Superknoten sind mit den jeweiligen Gruppierungs-Eigenschaftswerten (2.2) sowie beliebigen aggregierten Eigenschaftswerten (2.3) versehen. Die Kanten sind nach Superknoten der Start- und Zielknoten (2.4) sowie bestimmten Eigenschaftswerten (2.5) gruppiert. Analog zu den Superknoten werden die Superkanten mit den Gruppierungs-Eigenschaften (2.6) und ggf. weiteren Aggregaten (2.7) versehen.

In der Definition wurde aus Gründen der Übersichtlichkeit nicht vorgesehen, nach Bezeichnern zu gruppieren. Da dies in der Praxis jedoch häufig Anwendung findet, sei nachfolgend τ ein spezieller Eigenschaftsschlüssel, mithilfe dessen sich für alle Knoten und Kanten ihr jeweiliger Bezeichner als Eigenschaftswert gewinnen lässt, i. e. $\forall x \in (V \cup E)$ gelte: $\gamma(x, \tau) = \sigma(x) \in \Sigma_L$.

Beispiel Betrachte zur Veranschaulichung den logischen Graph G_1 aus Abbildung 2.11. Wähle $K_v = \{\tau\}$ und $K_\epsilon = \{\tau\}$, d.h. sowohl Knoten und Kanten sollen nach Bezeichner gruppiert werden. Weiterhin sollen die Elemente aller Gruppen gezählt werden, d. h. $\Lambda_v = \{\alpha_v : V \mapsto |V|\}$ und $\Lambda_e = \{\alpha_\epsilon : E \mapsto |E|\}$. Die von α_v bzw. α_e errechneten Aggregate sollen unter dem Eigenschaftsschlüssel `count` gespeichert werden, d. h. $f_\alpha(\alpha_v) = f_\alpha(\alpha_\epsilon) =$ `count`. Abbildung 2.11 zeigt den zusammenfassenden Graph von G_1, der sich aus den gewählten Parametern ergibt.

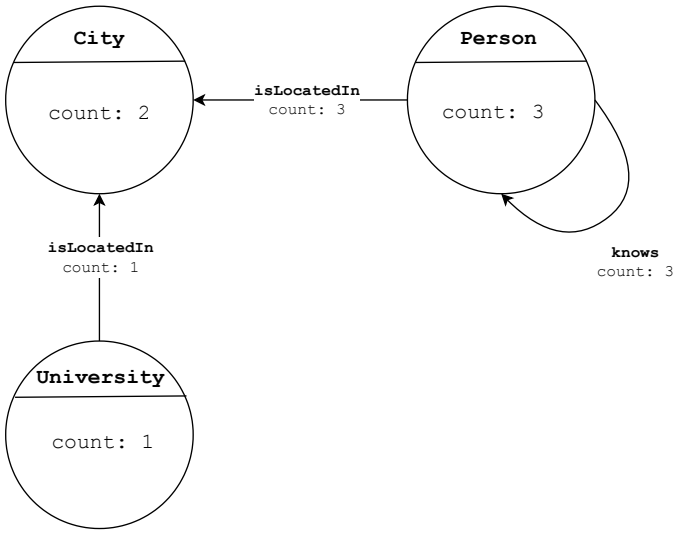


Abbildung 2.11: Exemplarischer zusammenfassender Graph

2.3 Referenzimplementierung Gradoop

In [32] haben Junghanns et. al. nicht nur das EPGM und die zugehörigen Operatoren bzw. Schnittstellen definiert, sondern auch eine Referenzimplementierung namens Gradoop (Graph analytics on Hadoop) als Forschungsprototyp vorgestellt. Unter der Annahme, dass Graph-Daten in der realen Welt i. A. sehr groß werden können, wurde eine Implementierung mit Technologien im Kontext von *Big Data* gewählt. Bevor auf die konkrete Implementierung des EPGM und der Operatoren eingegangen wird, werden nachfolgend zunächst die Grundlagen der verteilten Datenfluss-Programmierung erläutert.

2.3.1 Verteilte Datenfluss-Programmierung mit Apache Flink

Zur effizienten Verarbeitung von großen Datenmengen ist i. A. eine nebenläufige Ausführung mehrerer Prozesse notwendig. Rechenintensive Anwendungen können durch Erhöhung von Rechenleistung und/oder Hauptspeicher auf einer Maschine vertikal skaliert werden (engl. *scale-up*). Bei einfacheren Operatoren auf sehr großen Datenmengen können Anwendungen allerdings auch durch Hinzufügen weiterer Maschinen horizontal skaliert werden (engl. *scale-out*) [36]. Bei der horizontal skalierten Ausführung kann die bereits existierende Verteilung von Daten bzw. die dadurch entstehende Datenlokalität ausgenutzt werden. Dies hat sich *Google Inc.* zu Nutze gemacht und das verteilte Dateisystem *Google File System* [25] sowie das Programmiermodell *MapReduce* [20] entwickelt. Beide Konzepte wurden auch unter den Namen *Apache Hadoop* bzw. *Hadoop Distributed File System* (HDFS) quelloffen implementiert. Das MapReduce-Modell nutzt den Vorteil der Datenlokalität aus, in dem es möglichst viele Operationen lokal auf den einzelnen Cluster-Knoten ausführt. Da MapReduce zum damaligen Zeitpunkt noch viele Festplatten-Zugriffe mit sich brachte, wurde der Grundgedanke im Kontext von verteilter *Datenfluss*-Programmierung aufgegriffen, wobei dabei versucht wird, möglichst viele Daten im Hauptspeicher bzw. Festplattenzugriffe gering zu halten. Als Beispiel für solche Systeme sei neben Apache Flink [12] auch Apache Spark [47] genannt.

Apache Flink stellt Programmierschnittstellen zur Entwicklung von verteilten Datenanalyse-Workflows bereit. Grundprinzip ist dabei, dass dem Nutzer der Schnittstelle möglichst viele Aspekte und Details der Verteilung verborgen bleiben. Der Nutzer definiert einen Datenfluss, der stets bei (mindestens) einer Datenquelle beginnt und (mindestens) einer Datensenke endet. Auf den erhaltenen verteilten Datensätzen (in Flink jeweils als DataSet bezeichnet) kann mit einer Menge von Transformationen gearbeitet werden, wobei jede Transformation (mindestens) ein DataSet in ein neues DataSet überführt. Je nach Transformation kann die Ausführung nicht immer rein lokal ausgeführt werden, i. e. es müssen Daten zwischen den Ausführungsknoten ausgetauscht werden. Jedes DataSet kann mehrfach referenziert (wiederverwendet) und Transformationen miteinander verkettet werden. Aus der Workflow-Definition ergibt sich ein gerichteter, azyklischer Graph, der von Apache Flink optimiert und in einen konkreten Ausführungsplan übersetzt wird. Dieser Plan kann anschließend auf einem horizontal skalierten Cluster ausgeführt werden. Dabei wird versucht, lokal verkettete Operatoren möglichst innerhalb eines Threads auszuführen. Sämtlicher Datenaustausch zwischen den Knoten wird von Apache Flink

über das Netzwerk ausgeführt. Tabelle 2.1 beschreibt eine exemplarische Auswahl an Flink-Transformationen. Die Implementierung eines Workflows bzw. Verkettung von Transformation ist an das funktionale Paradigma angelehnt.

Einfache Operatoren wie Map oder Filter kommen i. A. ohne zusätzlichen Datenaustausch aus, während Operatoren wie Distinct, Join oder Group offensichtlich auf globales Wissen zurückgreifen und somit i. A. Daten ausgetauscht werden müssen. Üblicherweise bestehen die DataSets aus Tupeln bzw. serialisierten *Plain Old Java Objects* (POJO) - sämtliche nutzerdefinierte Funktionen lassen sich unter Verwendung der durch Tupel bzw. POJOs bereitgestellten Schnittstellen implementieren. Zwar sind die Details der verteilten Ausführung konzeptuell verborgen - ausgewählte Aspekte lassen sich aber steuern, z. B. der zu verwendende Verbund-Algorithmus oder die Aggregationsmethode in Folge einer Gruppierung. Darüber hinaus besteht die Möglichkeit, bestimmten Operatoren zur Laufzeit ein (kleines) DataSet vollständig zur Verfügung zu stellen (Broadcasting) - in bestimmten Szenarien kann dies gegenüber einem Verbund vorteilhaft sein. Weiterhin bietet die DataSet-API Möglichkeiten, die Datenverteilung zu steuern (z.B. Hash-Partition oder Range-Partition). Der Flink-Optimierer nimmt zudem Hinweise zur besseren Optimierung entgegen, die der Entwickler allerdings proaktiv geben muss.

Ein Anwendungsbeispiel der Transformationen wird im folgenden Abschnitt über die Implementierung von Operatoren in Gradoop präsentiert.

Transformation	Beschreibung
Distinct	Es werden alle Duplikate aus dem DataSet entfernt. Es kann ein nutzerdefinierte Identifikationsfunktion übergeben werden.
Map	Eine nutzerdefinierte Funktion überführt jedes Element eines DataSets in ein (neues) Datum (Projektion).
FlatMap	Wie Map, allerdings kann ein Element in kein, ein, oder mehrere neue Objekte überführt werden.
Filter	Es werden alle Elemente eines DataSets selektiert, die eine nutzerdefinierte Prädikatsfunktion erfüllen (Selektion).
Join	Es wird anhand einer Selektionsbedingung ein Verbund zweier DataSets gebildet. Die Verbundstrategie kann gewählt werden.
OuterJoin	Es wird anhand einer Selektionsbedingung ein äußerer Verbund zweier DataSets gebildet. Die Verbundstrategie kann gewählt werden.
Group	Die Elemente eines DataSets werden nach einem nutzerdefinierten Attribut gruppiert. Es existierenden verschiedene Transformationen, die zur Aggregation eines gruppierten Datensatzes herangezogen werden können (Reduce, GroupReduce, GroupCombine), je nachdem zu welchem Zeitpunkt und wie viele Werte aggregiert bzw. emittiert werden sollen.
CoGroup	Es werden zwei DataSets jeweils nach einem nutzerdefinierten Attribut gruppiert. Anschließend können die Gruppe des ersten und des zweiten DataSets mit gleichem Wert im Gruppierungsattribut gemeinsam verarbeitet und neue Elemente emittiert werden.
Union	Es werden die Elemente zweier DataSets vereinigt.

Tabelle 2.1: Exemplarische Auswahl von Flink-Transformationen

2.3.2 Implementierung des EPGM

In [32] wird eine Implementierung des EPGM in Form von drei verteilten DataSets vorgeschlagen. Dabei stellt je ein Datensatz die Knoten V, Kanten E und die (Existenz) der logischen Graphen L dar. Abbildung 2.12 zeigt die Eigenschaften der drei POJOs.

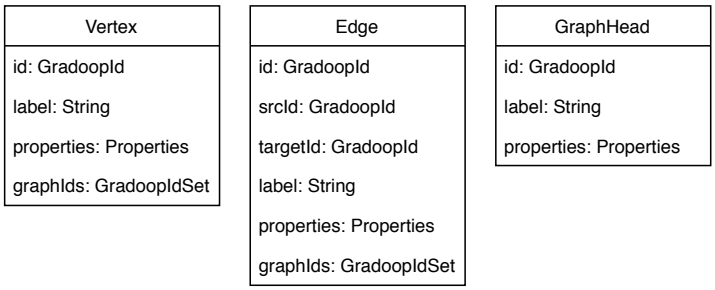


Abbildung 2.12: POJOs der EPGM-Implementierung in Gradoop

Jedes Element enthält eine ID, eine Zeichenkette als Bezeichner (σ) und seine Eigenschaften als Menge von Schlüssel-Wert-Paaren (γ). Die Graphzugehörigkeit der Knoten und Kanten wird an den Elementen in Form einer Liste von Graph-IDs gehalten. Die GraphHead-Datensätze enthalten keine Information über die in den Graphen enthaltenen Elemente. Start- und Zielknoten der Kanten werden in Form von IDs der jeweiligen Knoten dargestellt.

Bis auf `String` werden in den EPGM-POJOs keine einfachen bzw. von Apache Flink bereitgestellten Datentypen verwendet. Stattdessen wurden eigene Basis-Datentypen implementiert (siehe Abbildung 2.13). Diese werden im weiteren Verlauf der Arbeit wiederverwendet. `GradoopId` wird in Gradoop für jede Element-ID verwendet. Die Implementierung basiert auf dem `ObjectId`-Typen der *Binary JavaScript Object Notation* (BSON) Spezifikation [1] - dieser besteht aus 12 Byte. Bei Erzeugung einer neuen Instanz von `GradoopId` werden die aktuelle Systemzeit, ein Maschinenmerkmal, ein Prozessmerkmal und ein Zufallszähler berücksichtigt. So kann sichergestellt werden, dass die Wahrscheinlichkeit zweier identischer Instanzen hinreichend klein ist[1]. Ein `GradoopIdSet` besteht in den ersten vier Byte aus der Ganzzahl an enthaltenen `GradoopIds`, welche direkt darauf zu jeweils 12 Byte folgen. Ein `PropertyValue` enthält im ersten Byte einen Typen-Code, der bestimmt, welcher Datentyp folgt. Unterstützt werden unter anderem alle primitiven Java-Datentypen. Der `Properties`-Datentyp enthält eine Menge von Schlüssel-Wert-Zuordnungen, wobei der Schlüssel stets eine Zeichenkette und der Wert ein `PropertyValue` ist.

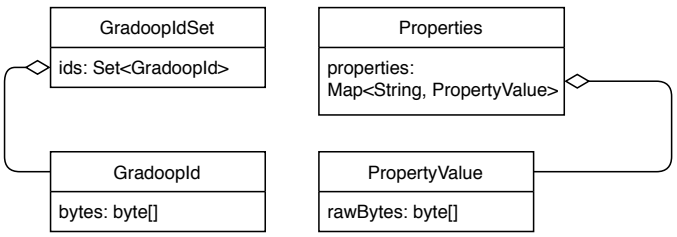


Abbildung 2.13: Basis Datentypen der EPGM-Implementierung in Gradoop

[1] https://www.mongodb.com/blog/post/quick-start-bson-data-types–objectid

2.3.3 Implementierung der Operatoren

Als Beispiel für eine Operator-Implementierung unter Verwendung der DataSet-API und dem im vergangenen Abschnitt erläuterten EPGM-Modell, wird nachfolgend die Implementierung der Intersection zweier EPGM-Instanzen in Gradoop erläutert. Zunächst sei vorausgesetzt, dass jeweils Schnittstellen zum Laden der drei DataSets der ersten und der drei DataSets der zweiten EPGM-Instanz bereitstehen (z.B. `firstVertices()` oder `secondGraphHeads()`). Zunächst müssen die GraphHeads der Ergebnis-Instanz (`outGraphHeads`) berechnet werden. Listing 2.2 zeigt die Implementierung in Pseudocode. Die GraphHead-DataSets der beiden Eingabe-Instanzen werden unter Zuhilfenahme des Union-Operators vereinigt (Zeile 2). Das Resultat enthält nun Duplikate. Um alle Elemente zu gewinnen, die in beiden DataSets enthalten sind, wird nach ID gruppiert (Zeile 3) und anschließend alle Gruppen vernachlässigt, die keine zwei Elemente halten (Zeile 5). Um ein duplikatfreies Ergebnis zu bekommen, wird jeweils eines der Gruppenergebnisse emittiert (Zeile 6).

```
1  DataSet<GraphHead> outGraphHeads = firstGraphHeads()
2    .union(secondGraphHeads())
3    .groupBy(graph -> { graph.id })
4    .reduceGroup(group -> {
5      if (group.size() == 2) {
6          emit(group.last())
7      }
8    });
```

Listing 2.1: Berechnung der resultierenden GraphHeads in DataSet-API Pseudocode

Nun kann das resultierende Knoten-DataSet berechnet werden. Dafür wird jeder Knoten zunächst in eine Menge von `<GraphId, Vertex>`-Tupeln überführt - und zwar für jeden Graph, in dem der Knoten enthalten ist (Zeilen 2 bis 6). Anschließend wird ein innerer Verbund mit den neuen GraphHeads gebildet, wobei die Graph-IDs übereinstimmen müssen (Zeilen 7 bis 9). Von jedem Verbund-Tupel wird jeweils nur der Knoten emittiert (Zeile 10). Da die Knoten im ersten Schritt ggf. dupliziert wurden, müssen abschließend alle Duplikate entfernt werden (Zeile 11). Die Berechnung der resultierenden Kanten kann analog erfolgen. Weitere Details zum Intersection-Operator folgen im Kapitel zur Implementierung.

```
1  DataSet<Vertex> outVertices = firstVertices()
2    .flatMap(vertex -> {
3      foreach (graph in vertex.graphIds) {
4        emit(<vertex, graph.id>)
5      }
6    })
7    .join(outGraphHeads)
8    .where(<vertex, graph_id> -> graph_id)
9    .equalTo(graph -> { graph.id })
10   .with(<<vertex, graph_id>, graph> -> { emit(vertex) })
11   .distinct();
```

Listing 2.2: Beispiel Operator Implementierung

2.4 Relationale Abstraktion in Apache Flink

Die bislang erläuterten Transformationen und Beispiele bezogen sich auf die DataSet-API von Apache Flink. Diese Schnittstelle ist zur Verarbeitung von statischen Daten geschaffen. Darüber hinaus ist Apache Flink in der Lage, kontinuierliche Datenströme zu verarbeiten - zu diesem Zweck wird die DataStream-API bereitgestellt. Apache Flink zeichnet sich unter anderem dadurch aus, dass es statische und kontinuierliche Daten innerhalb einer Implementierung verarbeitet [12]. Um auch eine gemeinsame Schnittstelle anbieten zu können, wurden DataSet- und DataStream-API in eine gemeinsame Schnittstelle abstrahiert: eine relationale und deklarative Table-API. Zusätzlich wurde diese Schnittstelle wiederum weiter abstrahiert, sodass relationale Abfragen auch mittels Sequel (SQL) [14] deklariert werden können. Zum Entstehungszeitpunkt dieser Arbeit wurde allerdings keine SQL-Spezifikation vollständig erfüllt. Abbildung 2.14 zeigt eine von der Apache Flink Community veröffentlichte Darstellung der Abstraktionslevel. Sämtliche Verarbeitung von Datenströmen bleibt in dieser Arbeit unbetrachtet. Im Folgenden wird ein grundlegendes Verständnis von relationaler Algebra [16] und SQL vorausgesetzt.

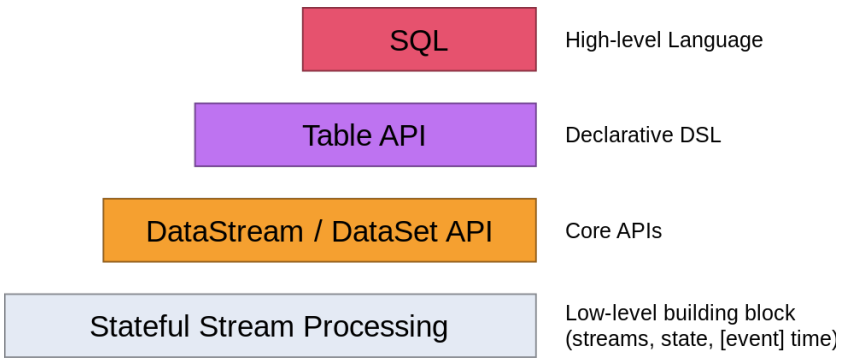


Abbildung 2.14: Abstraktionslevel in Apache Flink [2]

Im Gegensatz zu DataSets sind bei Verwendung der Table-API *Tabellen* das zentrale Element. Eingabe und Ausgabe einer jeden Abfrage ist eine Tabelle. Spaltennamen müssen über alle Tabellen hinweg eindeutig sein. Betrachte die beispielhafte Abfrage unter Verwendung der Java Table-API in Listing 2.3. Darin wird eine Knoten-Tabelle über eine einfache Selektion und Projektion in eine neue Tabelle überführt. Listing 2.4 zeigt dieselbe Abfrage als SQL-Abfrage, wobei in der `FROM`-Klausel nur der Flink-Laufzeitumgebung bekannte Tabellen referenziert werden können.

```
1  Table vertices = // ...
2  Table newVertices = vertices
3    .select("vertex_id AS id")
4    .where("vertex_label = 'person'");
```

Listing 2.3: Beispielhafte Abfrage unter Verwendung der Java Table-API

```
1  SELECT vertex_id AS id
2  FROM vertices
3  WHERE vertex_label = 'person'
```

Listing 2.4: Beispielhafte SQL-Abfrage

Analog zu den Transformationen der DataSet-API können Tabellen bzw. Abfragen verkettet werden. Jede Tabelle bzw. Abfrage wird intern als logischer Plan aufgefasst - analog zu logischen Relationen in Form von Sichten. Auf diese logischen Pläne werden Regeln der relationalen Abfrageoptimierung [15] angewandt. Apache Flink greift intern auf die Bibliothek Apache Calcite [8] zurück. Ergebnis ist ein optimierter relationaler Operatorbaum, der anschließend im Falle von statischen Daten in einen DataSet-Operatorbaum transformiert wird. Im Vergleich zur imperativen Verwendung der DataSet-API erfolgt die Optimierung automatisch und auf logischer Ebene und muss nicht vom Anwender unterstützt werden. Es kann beispielsweise weder eine Verbund-Strategie gewählt noch ein Broadcasting von Tabellen deklariert werden. Unter Verwendung der Table-API ist es zum Entstehungszeitpunkt dieser Arbeit nicht möglich, die Ergebnisse einer Tabelle bzw. eines logischen Plans wiederzuverwenden, da der logische Plan stets erneut ausgeführt wird.

Die relationalen Operatoren unterscheiden sich i. A. nicht konzeptionell von den durch die DataSet-API bereitgestellten Transformationen: Projektion, Selektion, Verbundoperationen, Gruppierung, Aggregation oder Duplikateleminierung können analog verwendet werden. In der relationalen Abfrage werden die Operationen allerdings stets auf Spalten der Tabellen deklariert - die Verwendung von POJOs oder nutzerdefinierten Funktionen zur Definition spezifischer Semantiken werden z. B. bei der Gruppierung nicht unterstützt. Die Funktionalität, die `flatMap` bereitstellt, kann ebenfalls nicht durch einen relationalen Standard-Operator deklariert werden. Unter Verwendung der Table-API wird dem Nutzer der Schnittstellen folglich grundsätzlich weniger Ausdrucksstärke geboten. Um dem Nutzer mehr Ausdrucksmöglichkeit zu bieten, können drei Arten von nutzerdefinierten Funktionen implementiert und zur Verwendung in relationalen Ausdrücken bereitgestellt werden [44]:

Skalarfunktionen Eine Skalarfunktion ordnet keinem, einem oder mehreren skalaren Werten einen neuen skalaren Wert zu. Betrachte beispielhaft eine Funktion `toUppercase`, die eine beliebige Zeichenkette entgegen nimmt, alle Buchstaben durch Großbuchstaben ersetzt und das Resultat zurückgibt. Ein solche Funktion lässt sich leicht in Java implementieren und in der Laufzeitumgebung von Flink registrieren. Listing 2.5 zeigt eine exemplarische Verwendung der Skalarfunktion innerhalb einer Projektion.

```
1 SELECT TO_UPPERCASE(vertex_label)
2 FROM vertices;
```

Listing 2.5: Beispielhafte Verwendung einer Skalarfunktion

Tabellenfunktionen Eine Tabellenfunktion ordnet keinem, einem oder mehreren skalaren Werten eine Tabelle, i. e. eine Menge von Zeilen mit mindestens einer Spalte, zu. Betrachte beispielhaft eine Funktion `split`, die eine Zeichenkette entgegen nimmt. Daraus erzeugt die Funktion eine Tabelle, die für jedes Wort der Zeichenkette eine Zeile erhält. Die Funktion bzw. die resultierenden Tabellen können nun als Verbundpartner verwendet werden. Listing 2.6 zeigt die Verwendung der Tabellenfunktion als Verbundpartner. Im Ergebnis der Abfrage sind für jeden

Knoten und jedes Wort des Knotenbezeichners jeweils eine Zeile bestehend aus Bezeichner und Wort enthalten.

```
1  SELECT vertex_label, word
2  FROM vertices
3  LEFT JOIN LATERAL TABLE(SPLIT(vertex_label)) AS T(word) ON TRUE;
```

Listing 2.6: Beispielhafte Verwendung einer Skalarfunktion

Aggregatsfunktionen Eine Aggregatsfunktion aggregiert eine Tabelle, i. e. eine Menge von Zeilen mit mindestens einer Spalte, zu einem skalaren Wert. Innerhalb einer Aggregatsfunktion werden ein Akkumulator (`ACC`) und eine Prozedur, die eine neue Zeile auf den Akkumulator anwendet, benötigt (`accumulate`). Soll die Aggregatsfunktion `count` z. B. die Zeilen zählen, genügt als Akkumulator eine Zahl. Bei jedem Eintreffen einer neuen Zeile kann der Akkumulator inkrementiert werden. Schließlich ist eine Prozedur erforderlich, die aus dem Akkumulator ein Aggregat berechnet (`getValue`). Im Fall von `count` genügt es, den Wert des Akkumulators zurückzugeben. Da die Aggregatsfunktion verteilt ausgeführt wird, muss darüber hinaus definiert werden, wie unabhängig erzeugte Akkumulatoren kombiniert werden sollen (`merge`). Das Beispiel fortführend genügt es, die Werte der Akkumulatoren zu addieren. Offensichtlich können die Implementierungen je nach Anforderung deutlich komplexer werden. Abbildung 2.15 veranschaulicht die interne Funktionsweise der `count`-Funktion. In Listing 2.7 ist eine exemplarische Verwendung der Funktion im Rahmen einer Gruppierung gezeigt.

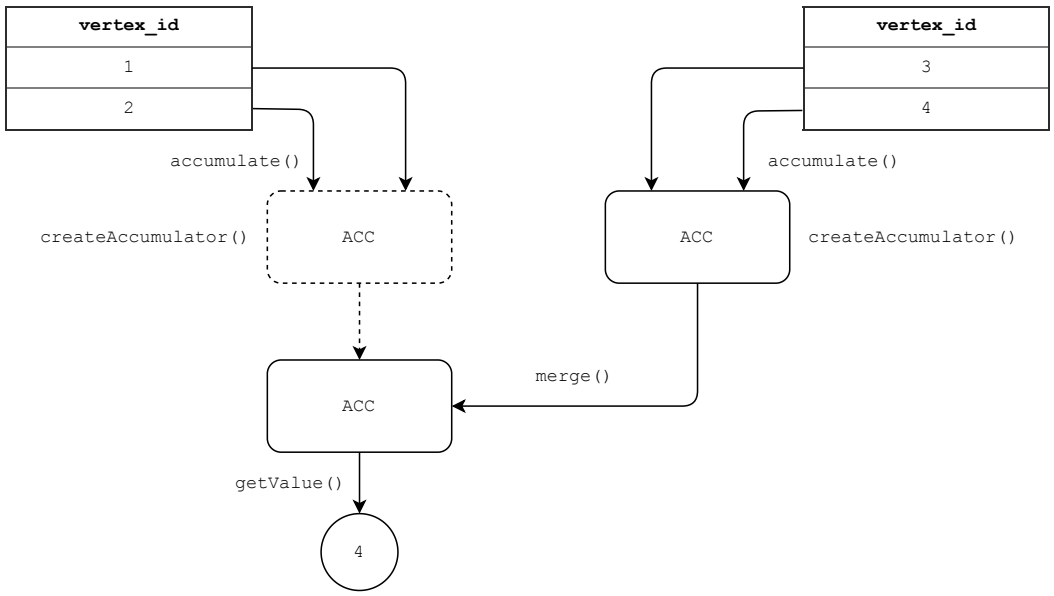


Abbildung 2.15: Funktionsweise einer exemplarischen Aggregatsfunktion

```
1  SELECT COUNT(vertex_id)
2  FROM vertices
3  GROUP BY vertex_label;
```

Listing 2.7: Beispielhafte Verwendung einer Aggregatsfunktion

3 Verwandte Arbeiten

Junghanns et. al. haben mit Gradoop ein Framework zur verteilten Graphdatenverarbeitung veröffentlicht. Dieses enthält wie in den Grundlagen erläutert eine EPGM-Implementierung. Während zum Veröffentlichkeitszeitpunkt dieser Arbeit keine weitere Implementierung bekannt ist, existieren bereits einige Frameworks zur verteilten Graphdatenverarbeitung. In [26] haben Gonzales et. al. zum Beispiel eine Graph-Erweiterung von Apache Spark namens *GraphX* vorgestellt. GraphX implementiert zur Modellierung von Graphdaten das Property Graph Model. Der Fokus liegt dabei auf iterativen Graphalgorithmen - die gebotenen Schnittstellen erlauben es Algorithmen Knoten-zentriert (engl. *vertex-centric*) zu definieren. Dabei wird auf das populäre Programmiermodell *Pregel* [35] zurückgegriffen, in dem Algorithmen aus Sicht der Knoten formuliert werden, welche Nachrichten untereinander austauschen. Die bekannteste quelloffene Implementierung des Pregel-Programmiermodells ist Apache Giraph [5]. Es existiert auch eine Grapherweiterung namens *Gelly* [4] für Apache Flink. Diese beruht auf einem simplen Datenmodell: Knoten und Kanten können jeweils nur einen Wert beinhalten. Neben der Bereitstellung von Schnittstellen zur Definition von iterativen Graphalgorithmen, wurde in Gelly auch der Subgraph-Operator implementiert. Im Gegensatz zu Systemen wie Gelly oder GraphX sind die Nutzerschnittstellen von Gradoop auf einem höheren Abstraktionslevel definiert. Vielmehr wird Gelly selber innerhalb von Operator-Implementierungen in Gradoop angewandt. Eine Verwendung von Gradoop ist möglich, ohne die Details des Graphmodells oder bestimmter Programmiermodelle zu kennen. Das EPGM unterscheidet sich von allen bereits bekannten Graphmodellen durch seine integrierte Unterstützung von Graph-Mengen. Eine detaillierte Analyse verwandter Arbeiten haben Junghanns et. al. auch in [30] durchgeführt. Auf einzelne Operatoren wird mit Blick auf den Umfang dieser Arbeit nicht weiter eingegangen. Da keiner der im Rahmen dieser Arbeit implementierten Operatoren iterativ implementiert wird, bleibt die iterative Verarbeitung mit Pregel im weiteren Verlauf unbetrachtet. Insbesondere Graph-Abfragen mittels Graph-Matching bleiben in dieser Arbeit unbetrachtet. Auch im RDF-Umfeld wurden bereits Systeme veröffentlicht: *FLINKer* von Azzam et. al. führt SPARQL-Abfragen auf RDF-Daten in Apache Flink aus. Die Abfrage wird innerhalb des Systems mit Gelly ausgeführt.

Die Idee, Abfragen auf horizontal verteilten Daten (i. e. im Kontext von Hadoop) mittels einer Abfragesprache zu abstrahieren, wurde nicht erst mit der Table-API von Apache Flink vorgeschlagen. Es existieren diverse Implementierungen, die deklarative Abfragen in verteilte Workflows übersetzen. Apache Pig transformiert beispielsweise Abfragen, die in der Sprache *Pig Latin* formuliert werden, in eine Folge von MapReduce-Jobs [38]. Apache Hive hingegen transformiert SQL-Abfragen in solche Folgen [43]. Ein mit Gradoop vergleichbares System namens *GLog* zur verteilten Graphanalyse wurde in diesem Kontext in [24] vorgeschlagen. Gao et. al. haben darin die logikbasierten Datenbanksprache *Datalog* [13] um Graph-Abfragen erweitert. Auf dem durch die Mischung von relationalen und Graph-Daten entstehenden *Relational-Graph*-Modell wurde die Abfragesprache *GLog* definiert. Jeder der auf dem *Relational-Graph*-Modell definierten Operatoren wird dann in eine Sequenz von MapReduce Jobs transformiert bzw. optimiert. Gao et. al. haben anhand beispielhafter Graphabfragen gezeigt, dass ein solches speziell für Graphen

optimiertes System im Vergleich zu generischen Lösungen wie Apache Pig oder Apache Hive performanter läuft. Im Vergleich zu Gradoop unterstützt *GLog* keine Graph-Mengen.

Ein weiteres System zur gleichzeitigen Verarbeitung von Graph- und relationalen Daten haben Dave et. al. mit *GraphFrames* vorgestellt [19]. Damit können innerhalb eines Workflows relationale und Graph-Abfragen gemischt werden. Die Graphabfragen werden vom *GraphFrames*-System in relationale Abfragen überführt und über die relationale SQL-Schnittstelle von Apache Spark ausgeführt. Die Graphdaten werden in Form von Knoten, Kanten und Triplets, i. e. Kanten mit zugehörigen Knoten, materialisiert. Dave et. al. haben die Verbund-Operationen als teuer identifiziert und sich intensiv mit deren Optimierung befasst.

Neben der Abstraktion von verteilten Datenfluss-Programmen, wurden auch Graphverarbeitungssysteme auf Basis von klassischen relationalen Datenbankmanagementsystemen (RDBMS) entwickelt. Jindal et. al. haben mit ein solches System vorgestellt [28]. Grundlage liefert dabei das spaltenorientierte RDBMS Vertica [33]. Jindal et. al. haben das populäre Programmiermodell Pregel relational implementiert, i. e. es gibt nicht nur Datenbanktabellen zur Speicherung von Knoten und Kanten, sondern z. B. auch für die Nachrichten, die zwischen den Knoten ausgetauscht werden. Im Gegensatz zu den meisten Big Data Systemen erfüllt ein solches System alle ACID-Eigenschaften. Mit dem System auf Basis von Vertica konnten mit Apache Giraph vergleichbare Laufzeitergebnisse erzielt werden. Die Autoren haben damit gezeigt, dass iterative Graphalgorithmen in Form von relationalen Abfragen ausgedrückt werden können. Gleiches konnten Fan et. al. mit *Grail* zeigen [22]. Dieses basiert ebenfalls auf einem relationalen Datenbankmanagementsystem und implementiert das Pregel-Modell. Im Unterschied zu der Implementierung von Jindal et. al. wird die Adjazenz der Knoten nicht in Form von Triplets, sondern als Liste an den jeweiligen Knoten bereitgehalten.

Wie bei der Beschreibung der bestehenden Systeme auffällt, unterscheiden sich neben den Technologien auch das zugrundeliegende Graphmodell und das Datenschema. Während zum Property Graph Modell wenig relationale Schemata veröffentlicht wurden, existieren im RDF-Umfeld viele Untersuchungen. Sakr et. al. klassifizieren in [40] einige relationale RDF-Schemata. Die grundlegenden Ideen dazu werden im nächsten Kapitel erläutert. Viele Konzepte aus relationalen Datenbankmanagement-System, wie z. B. Indizes, können allerdings nicht ohne weiteres auf verteilte Systeme wie z. B. Apache Flink übertragen werden. Eine der wesentlichen Herausforderungen beim Entwurf von Schemata ist die Abbildung der Adjazenz. Darüber hinaus ist es unter Umständen von Interesse, ein hinsichtlich der Eigenschaften-Menge flexibles Schema bereitzustellen.

Schätzle et. al. haben in [41] ein RDF-System namens *S2RDF* vorgestellt. Darin werden SPARQL-Abfragen in relationale Abfragen transformiert und über die SQL-Schnittstelle von Apache Spark ausgeführt. Hauptaugenmerk von Schätzle et. al. lag auf dem Entwurf eines Partitionierungsschemas namens *ExtVP*. Basis davon sind die nach Prädikat bzw. Eigenschaft vertikal partitionierten Eigenschaftswerte. Grundprinzip von *ExtVP* ist anschließend die Materialisierung von bestimmten Verbundoperationsergebnissen. Ziel ist es, sowohl den Eingabe- und Ausgabeaufwand als auch den Verbundaufwand zur Ausführungszeit gering zu halten.

Ein weiteres relationales Schema zur Speicherung von RDF-Daten haben Bornea et. al. in [10] vorgeschlagen. Dieses neuartige Schema *DB2RDF* zeichnet sich durch seine Flexibilität hinsichtlich der Prädikate bzw. Eigenschaften aus. Es wird nicht für jede der n in einem Datensatz enthaltenen Prädikat eine eigene Spalte hinzugefügt, sondern es existieren stets feste k Spalten. Es ist davon auszugehen, dass k i. A. kleiner als n ist. Jedes Prädikat wird über eine Hashfunktion einer der k Spalten zugeordnet. Sollte ein Datensatz Werte zu Prädikaten haben, die auf dieselbe Spalte abgebildet wurden, wird eine weitere Zeile in die Tabelle eingefügt. Die Grundidee besteht darin, die Zuordnung so zu wählen, dass häufig gemeinsam auftretende Prädikate nicht dieselbe Position zugeordnet bekommen und somit mehrfache Zeilen in der Tabelle verhindert werden. Bornea et. al. sprechen davon, die tatsächlichen Daten in ein festes relationales Modell zu *schreddern*. Um das *DB2RDF*-Schema herum haben die Autoren ein System entwickelt, welches SPARQL-Abfragen in SQL-Abfragen auf jenem Schema übersetzt.

Mit *SQLGraph* haben Sun et. al. in [42] die Idee von *DB2RDF* wieder aufgegriffen, um ein Schema zu entwickeln, in dem Instanzen des Property Graph Modells gespeichert werden können. *SQLGraph* übersetzt Graph-Abfragen, die in der Sprache Gremlin [3] formuliert wurden, in SQL-Abfragen und führt diese auf einem relationalen Datenbankmanagementsystem aus. Sun et. al. weisen darauf hin, dass im Kontext von RDF zwar viele Schemata veröffentlicht wurden, in Bezug auf Property Graphs hingegen wenig geforscht wurde. Die Autoren stellen insbesondere die Speicherung von Kanteneigenschaften als offenen Punkt heraus, da diese im RDF-Modell nicht vorgesehen sind (vgl. Grundlagen). In *SQLGraph* wurde die Idee des *Schredderns* nicht nur auf die Speicherung der Eigenschaften, sondern auch auf die Speicherung der Adjazenz angewendet. Die Adjazenz der Knoten wird dabei im Modell von Sun et. al. als Adjazenzliste an den Knoten gespeichert. Die Größe der Adjazenzliste ist offensichtlich dynamisch, wird in *SQLGraph* allerdings in eine feste Anzahl an Spalten *geschreddert*. Für die Zuordnung zu den Spalten wird eine Hashfunktion auf die Kantenbezeichner angewandt. Sun et. al. haben unter anderem untersucht, ob die SQL-Abfragen auf den *geschredderten* Tabellen effizienter als auf in einer relationalen Datenbank enthaltenen JSON-Daten ausgeführt werden können. Interessanterweise erweist sich das relationale Schema bei der Speicherung der Adjazenz als effizienter, während die Speicherung der Eigenschaften in einer JSON-Struktur bessere Ergebnisse lieferte.

Zum Zeitpunkt der Entstehung dieser Arbeit sind keine Schemata für EPGM-Instanzen veröffentlicht, die über die Referenzimplementierung hinausgehen - ebenso wurde insbesondere die Implementierung des Grouping-Operator nicht in relationaler Abstraktion zur verteilten Ausführung veröffentlicht. Ein Vergleich der relational abstrahierten Implementierung der EPGM-Operatoren mit einer weniger abstrakten Implementierung unter Verwendung der DataSet-API liegt ebenfalls noch nicht vor.

4 Entwurf

In diesem Kapitel werden verschiedene Schemata zur Darstellung von EPGM-Instanzen entwickelt sowie mögliche Vor- und Nachteile diskutiert.

4.1 Rahmenbedingungen

Bevor einzelne Schemata eingeführt werden, sei zunächst der Begriff *Schema* im Kontext dieser Arbeit definiert: Ein *Schema* bestehe im Folgenden aus eine Menge von mehreren Tabellenschemata, wobei ein Tabellenschema aus einer Menge von Paaren aus Spaltenbezeichner und Spalten-Datentyp besteht. Spaltenbezeichner seien darüber hinaus über die Menge aller Tabellen hinweg eindeutig (vgl. Kapitel 2.3). Eine Tabelle wiederum ist die logische Sicht auf die Daten, die im Rahmen dieser Arbeit in einem verteilten Dateisystem im CSV-Format gehalten werden. Die EPGM-Operatoren lassen sich anschließend auf dem Schema definieren und auf der Menge oder einer Teilmenge der Tabellen ausführen.

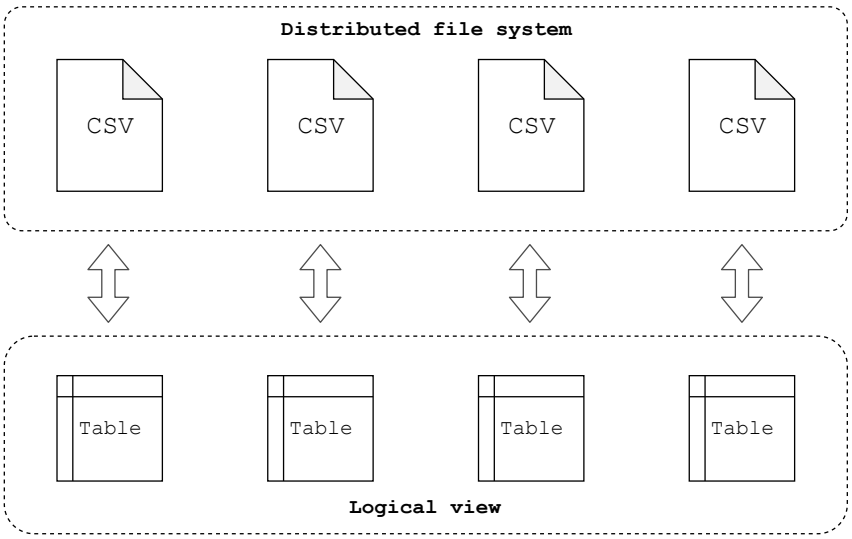


Abbildung 4.1: Logische Sicht auf verteilte CSV-Daten

Ein Table-Scan impliziert das vollständige Lesen und Parsen von CSV-Dateien. Dieser Ein- und Ausgabeaufwand (E/A) sollte beim Entwurf von Schemata berücksichtigt werden. Je nach Anwendungsgebiet kann es also sinnvoll sein, einzelne Daten in separate Tabellen auszulagern, um E/A-Aufwand, i. e. Lesen und Parsen von CSV-Daten, zu sparen.

Beim Entwurf von relationalen Datenbankschemata werden zusätzlich Integritätsbedingungen definiert, deren Einhaltung das Datenbankmanagementsystem garantiert. Derartige Bedingungen einzuhalten (wie z. B. Primärschlüssel- oder Fremdschlüsselbedingungen), wird im Kontext von Big Data bzw. OLAP nicht notwendigerweise gefordert. Nichtsdestoweniger sollte man beim Entwurf die Semantik einzelner Spalten beachten. Daher wird die Fremdschlüssel-Semantik in den nachfolgenden Veranschaulichungen von Schemata kenntlich gemacht.

4.2 Anforderungen

Betrachtet man die Definition des EPGM in Kapitel 1, muss ein Schema folgende Informationen abbilden:

- Eine Menge von identifizierbaren Knoten, Kanten und logischen Graphen
- Anfangs- und Endknoten einer Kante
- Bezeichner von Knoten, Kanten und logischen Graphen
- Zugehörigkeit von Knoten und Kanten zu logischen Graphen
- Eigenschaftwerte von Knoten, Kanten und logischen Graphen

In der bestehenden Referenzimplementierung werden Knoten und Kanten als zwei separate Datensätze aufgefasst, wobei ein Kantendatensatz jeweils eine Referenz auf Anfangs- und Endknoten enthält. Dieses grundlegende Schema soll auch im Rahmen dieser Arbeit Anwendung finden, i. e. jedes Schema enthält eine Tabelle für Knoten und eine für Kanten. Darüber hinaus wird eine Tabelle für logische Graphen benötigt.

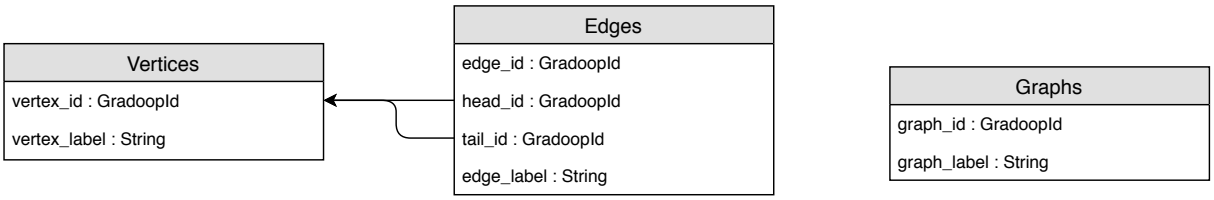


Abbildung 4.2: Grundlegendes Schema

Abbildung 4.2 zeigt das Basisschema, welches den im weiteren Verlauf dieses Kapitels erläuterten Schemata zugrunde liegt. Knoten, Kanten und (logische) Graphen werden anhand des Wertes in der jeweiligen `id`-Spalte identifiziert - die Bezeichner der Elemente werden in einer entsprechenden Spalte abgelegt. Quell- und Zielknoten einer Kante werden in Form von Referenzen auf Knoten an den Kanten gespeichert. Die in Gradoop enthaltenen Datentypen werden im weiteren Verlauf wiederverwendet (vgl. Kapitel 2.3.2).

Dieses grundlegende Schema berücksichtigt bereits einen Teil der notwendigen Informationen einer EPGM-Instanz. Noch nicht abgebildet sind die Graphzugehörigkeit und die Eigenschaftswerte. Insbesondere für die Modellierung von Eigenschaften existieren im RDF-Umfeld verschiedene Ansätze, da viele RDF-Systeme auf relationalen Datenbanksystemen aufbauen [46], [6], [45]. Sakr et al. geben in [40] einen Überblick über relationale Repräsentationen des RDF-Modells, indem sie die Schemata in drei Kategorien klassifizieren. Jede der drei Klassen dient als Inspiration für eines der nachfolgend vorgestellten Schemata.

4.3 Graphs Vertices Edges - Schema

Eine der drei vorgeschlagenen Kategorien stellen die *Property table stores* dar. Dabei werden alle zu einem Knoten oder einer Kante gehörigen Eigenschaftswerte direkt in der Knoten- bzw. Kanten-Tabelle gespeichert. Im Allgemeinen ist es möglich, für jede Eigenschaft bzw. jeden Eigenschaftsschlüssel eine Spalte anzulegen. Andere Ansätze wurden bereits in den verwandten Arbeiten erwähnt. Betrachtet man auch die Menge von Referenzen auf logische Graphen (in denen z. B. ein Knoten enthalten ist) als Eigenschaft, sind nun alle Aspekte des EPGM berücksichtigt. Das Halten von Graphzugehörigkeit und Eigenschaften an den Elementen selber entspricht der in Kapitel 1 vorgestellten bestehenden EPGM-Repräsentation. Abbildung 4.3 zeigt dieses Schema (relational betrachtet). Die Graphzugehörigkeit ist dabei in einer eigenen Spalte modelliert - alle übrigen Eigenschaftswerte werden in einem `Properties`-Typen zusammengefasst. Dies ermöglicht hinsichtlich der Eigenschaften die Freiheit, dass die Menge an Eigenschaftsschlüsseln nicht fest sein muss.

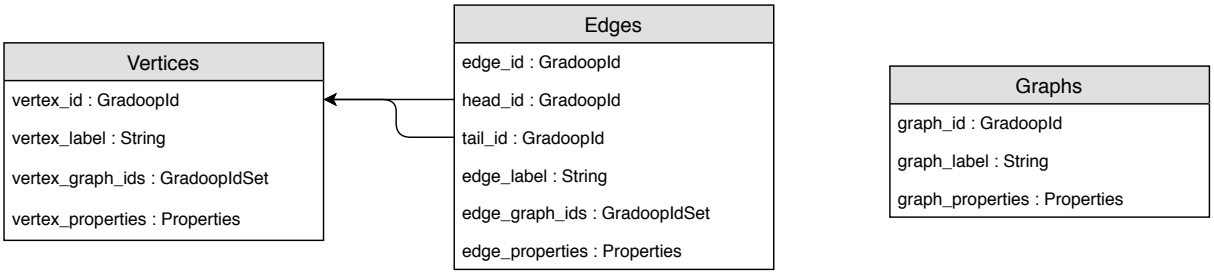


Abbildung 4.3: GVE-Schema

(a) `vertices`

vertex_id	vertex_label	vertex_graph_ids	vertex_properties
V4	Person	[1, 2]	{name: Carol}
V5	Person	[1, 2]	{name: Alice}
V6	Person	[1, 2]	{name: Bob}
...	...	...	...

(b) `edges`

edge_id	head_id	tail_id	edge_label	edge_graph_ids	edge_properties
E4	V4	V5	knows	[1, 2]	{since: 2011}
E5	V4	V6	knows	[1, 2]	{since: 2012}
E8	V5	V6	knows	[1, 2]	{since: 2010}
...	...	...	...	...	...

(c) `graphs`

graph_id	graph_label	graph_properties
G1	Geo View	{edgeCount: 7}
G2	Content View	{edgeCount: 8}
...	...	...

Tabelle 4.1: Exemplarische Daten im GVE Schema

Dieses Schema bietet eine optimale Vergleichsbasis zwischen der bestehenden Implementierung verschiedener Operatoren und einer Realisierung auf Basis einer relationalen Abstraktion. Tabelle 4.1 zeigt die relationale Sicht auf einen Ausschnitt der in Abbildung 2.2 veranschaulichten

EPGM-Instanz. Im Folgenden wird dieses Schema als *Graphs Vertices Edges*-Schema (GVE) bezeichnet.

Vorteilhaft an diesem Schema ist die direkte Verfügbarkeit aller zugehörigen Daten an den Knoten, Kanten und Graphen durch einmaliges Laden der Tabelle. Es sind keine weiteren (Verbund-) Operationen notwendig. Nachteilhaft ist es, dass die zugrundeliegende verteilte CSV-Datei stets vollständig durchlaufen werden muss - auch wenn nur Teile der Daten benötigt werden. Weiterhin ist es schwierig, mit relationalen Standard-Operatoren auf Datentypen wie `GradoopIdSet` oder `Properties` zu arbeiten.

4.4 Vertikales Schema

Alternativ zum GVE-Schema ist es möglicherweise sinnvoller, Eigenschaftswerte von der Knoten bzw. Kanten-Tabelle zu lösen. Zudem könnte es unter Umständen sinnvoller sein, die `n:m`-Beziehung zwischen Knoten (bzw. Kanten) und logischen Graphen mittels einer neuen Tabelle aufzulösen. Beide Ideen wurden in dem in Abbildung 4.4 gezeigten Schema umgesetzt. Die Graphzugehörigkeit wird jeweils in einer eigenen Tabelle bestehend aus zwei Referenzen umgesetzt. Die Eigenschaftswerte werden in grober Anlehnung an das RDF-Modell als Tripel abgelegt. Sakr et. al. fassen derartige relationale Sichten auf RDF-Daten in [40] als *Vertical table stores* zusammen. Das gezeigte Schema wird daher im weiteren Verlauf der Arbeit als *Vertikales* Schema bezeichnet.

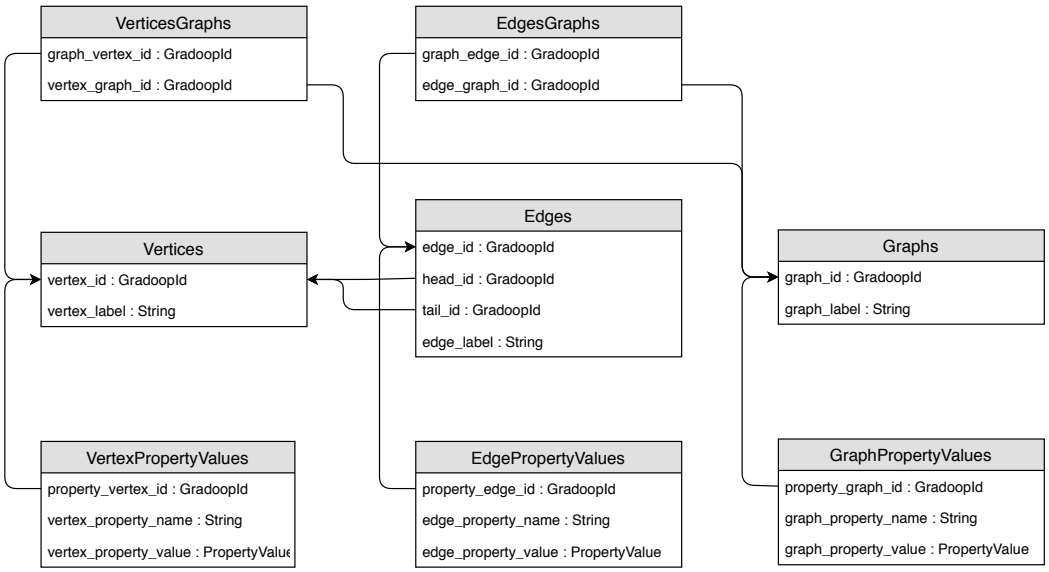


Abbildung 4.4: Vertikales Schema

Die Zugehörigkeit der Knoten zu den logischen Graphen in der normalisierten Form ist in Tabelle 4.1(a) ersichtlich. Offensichtlich steigt der Speicherbedarf im Vergleich zum GVE Schema, da statt einer Menge von `GraphIDs` eine Menge von `(KnotenID, GraphID)`-Tupeln gespeichert

(a) `VerticesGraphs`

graph_vertex_id	vertex_graph_id
V4	G1
V5	G1
V6	G1
V4	G2
V5	G2
V6	G2
...	...

(b) `VertexPropertyValues`

property_vertex_id	vertex_property_name	vertex_property_value
V4	name	Carol
V5	name	Alice
V6	name	Bob
...	...	...

(c) `EdgePropertyValues`

property_edge_id	edge_property_name	edge_property_value
E4	since	2011
E5	since	2012
E8	since	2010
...	...	...

(d) `GraphPropertyValues`

property_graph_id	graph_property_name	graph_property_value
G1	edgeCount	7
G2	edgeCount	8
...	...	...

Tabelle 4.2: Exemplarische Daten im vertikalen Schema

werden muss. Gleiches gilt für die Eigenschaften: In Tabelle 4.1(b) ist exemplarisch die Knoten-Eigenschaften-Tabelle gezeigt. Da jede Eigenschaft eines Knotens inklusive einer Referenz auf den jeweiligen Knoten in einer separaten Zeile gespeichert wird, entsteht im Vergleich zum GVE Schema ein zusätzlicher Speicheraufwand. Zwar enthält die Knoten- (und Kanten-) Tabelle eine deutlich geringe Datenmenge - insgesamt wird eine EPGM-Instanz im vertikalen Schema einen größeren Speicherbedarf als im GVE Schema aufweisen.

Das Schema enthält nun weder mehrwertige Daten noch komplexe Datentypen. Dies vereinfacht die Formulierung von Operatoren mithilfe von Standard-SQL. Ein offensichtlicher Nachteil der Auslagerung in separate Tabellen ist die Erfordernis, einen Verbund zu bilden, wenn z. B. Knoten zusammen mit all ihren Eigenschaften gemeinsam benötigt werden. Je nach Verteilung der Daten müssen dabei unter Umständen Daten über das Netzwerk ausgetauscht werden. Gleiches gilt für die Graphzugehörigkeit. Nichtsdestotrotz kann es effizienter sein, zunächst einen Knoten-Datensatz mit geringer Selektivität zu filtern und anschließend einen Verbund mit den Eigenschaften zu bilden, anstatt den gleichen Knoten-Datensatz im GVE Schema vollständig durch zu laufen und zu filtern.

4.5 Horizontales Schema

Eine weitere Möglichkeit, Eigenschaften in separate Tabellen auszulagern, besteht darin, für jede Eigenschaft bzw. jeden Eigenschaftsschlüssel eine eigene Tabelle zu führen. Sakr et. al. bezeichnen diese Variante als *Horizontal table store* [40]. Abbildung 4.5 zeigt dieses *horizontale* EPGM-Schema, wobei eine zugehörige EPGM-Instanz Eigenschaften mit n verschiedenen Eigenschaften-Schlüsseln beinhalte. Offensichtlich ist bei diesem Schema die Anzahl an Tabellen nicht fest. Zum Laden bzw. Speichern einer solchen Instanz sind Meta-Informationen über verwendeten Eigenschaften-Schlüssel erforderlich.

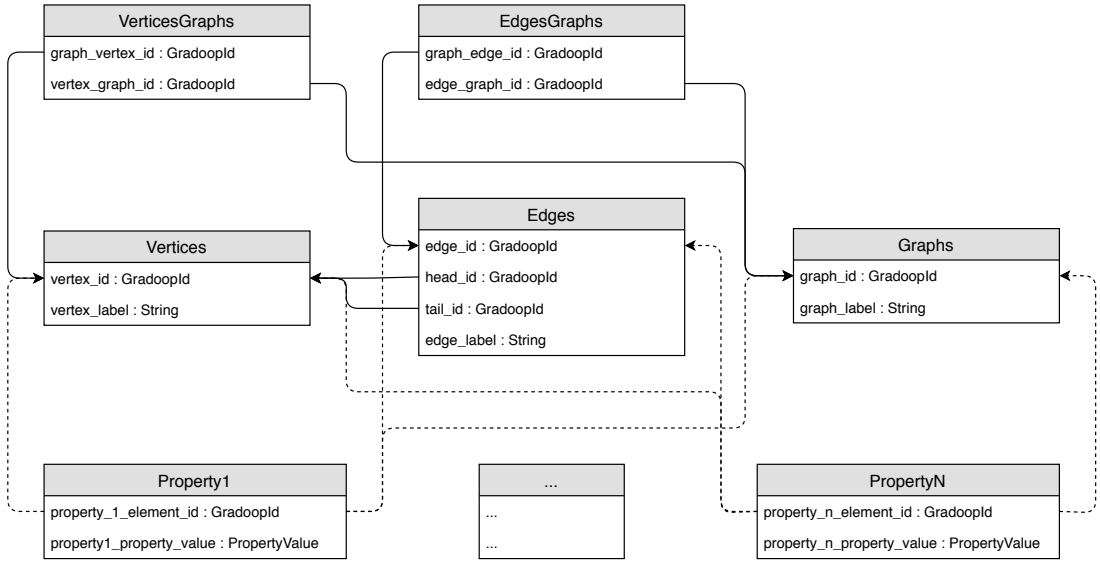


Abbildung 4.5: Horizontales Schema

Wie in der Abbildung ersichtlich, gibt es keine Separation der Eigenschaften in Knoten-, Kanten- und Graphen-Eigenschaften. Die in einem der Eigenschaften-Tupel enthaltene `ElementId` referenziert also polymorph entweder einen Knoten, eine Kante oder einen logischen Graphen. Diese Umsetzung ist möglich, da innerhalb von Gradoop ein Wert des Typs `GradoopId` global eindeutig ist (vgl. Kapitel 2.3.2) und es somit maximal einen Verbund-Partner (Knoten, Kante oder logischen Graphen) zu einem solchen Tupel geben kann. Diese Polymorphie erhöht i. A. allerdings die Anzahl an notwendigen Vergleichen bei der Bildung eines solchen Verbunds: Sollen alle zu einer Menge von Eigenschaftswerten gehörigen Elemente gefunden werden, müssen sowohl Knoten als auch Kanten und logische Graphen berücksichtigt werden. Soll hingegen ein Verbund von einer Menge von Knoten oder Kanten und der zugehörigen Eingeschaftswerte gebildet werden, müssen alle Eigenschafts-Tabellen in Betracht gezogen werden.

Im Vergleich zum vertikalen Schema weist eine EPGM-Instanz im horizontalen Schema eine geringere Datengröße auf, da die Tripel auf ein Dupel (2-Tupel) reduziert werden. Dennoch ist zu berücksichtigen, dass es bei einer Anzahl von mehr als 3 Eigenschafts-Schlüsseln mehr Tabellen als im vertikalen Schema gibt und eine Operator-Implementierung somit i. A. mehr Verbund-Operationen benötigt.

Die in Abbildung 2.2 gezeigte EPGM-Instanz enthält z. B. die Eigenschaften-Schlüssel `name`, `title`, `since` und `edgeCount`. Folglich muss das Schema für diese Instanz 4 Eigenschaften-Tabellen enthalten. Drei dieser Tabellen sind entsprechend dem Ausschnitt aus den Beispieldaten in Tabelle 4.3 im horizontalen Schema aufgeführt.

(a) `Name`

name_element_id	name_property_value
V4	Carol
V5	Alice
V6	Bob
...	...

(b) `Since`

since_element_id	since_property_value
E4	2011
E5	2012
E8	2010
...	...

(c) `EdgeCount`

edgecount_element_id	edgecount_property_value
G1	7
G2	8
...	...

Tabelle 4.3: Exemplarische Daten im horizontalen Schema

Die Notwendigkeit von Metadaten im horizontalen Schema widerspricht in gewisser Weise der im Kontext von Big Data häufig geforderten Schemalosigkeit von Daten. Derzeit ist eine Unterstützung der Graph-Abfragesprache OpenCypher [27] im *Pattern Matching*-Operator in Gradoop geplant - diese wiederum basiert auf einer Übersetzung von Cypher-Abfragen in relationale Operatoren.[2] Das geforderte Schema der Graph-Daten ähnelt dabei in seiner Grundidee dem horizontalen Schema. Im Gegensatz zum horizontalen Schema wird dabei zusätzlich für alle Knoten bzw. Kanten gleichen Bezeichners eine separate Eigenschaftentabelle gefordert. Um die Anzahl der Tabellen klein zu halten, wurde diese Forderung in dieser Arbeit nicht erfüllt.

4.6 Vergleich

Die nachfolgende Übersicht stellt die drei eingeführten Schemata hinsichtlich einiger Aspekte grob gegenüber:

	GVE Schema	Vertikales Schema	Horizontales Schema
Anzahl Tabellen	3	8	$5 + n$
Graphzugehörigkeit	mengenwertig	relational	relational
Eigenschaften	mengenwertig	ausgelagerte Tripel	ausgelagerte 2-Tupel
Eigenschaften-Flexibilität	ja	ja	nein
Datengröße	+	- -	-

Tabelle 4.4: Vergleich der Schemata

Wie sich die Schemata hinsichtlich der Operator-Implementierungen sowie Laufzeit und Skalierung der Operatoren unterscheiden, wird im weiteren Verlauf der Arbeit analysiert.

[2] https://github.com/opencypher/morpheus

5 Implementierung

In diesem Kapitel wird die Implementierung der in den Grundlagen eingeführten EPGM Operatoren erläutert, wobei diese mithilfe von relationalen Abfragen und für jedes der drei im vergangenen Kapitel eingeführten Schemata erfolgt. Bevor auf die konkreten relationalen Abfragen eingegangen wird, werden einige Vorüberlegungen und Vorbereitungen behandelt, die dem Gesamtverständnis dienen.

5.1 Grundlagen

Modell Elementarer Bestandteil des EPGM ist die enthaltene Menge an logischen Graphen. Nach Definition besteht eine Instanz des EPGM also stets aus einer Menge von Knoten, Kanten und logischen Graphen. Bei der formalen Definition der Operatoren in Kapitel 2.2 wurden diese hingegen nicht immer auf einer Menge an Graphen $\mathcal{G}$ definiert, sondern auch auf einzelnen Graphen G. Für die Implementierung gilt es nun zu modellieren, wie die verschiedenen Operatoren in das EPGM eingebettet werden können. In Gradoop wurde ein Modell implementiert, welches EPGM-Instanzen mit Mengen von logischen Graphen und EPGM-Instanzen mit 1-elementiger Graph-Menge unterscheidet (siehe Abbildung 5.1). Diese Modell wurde in dieser Arbeit beibehalten.

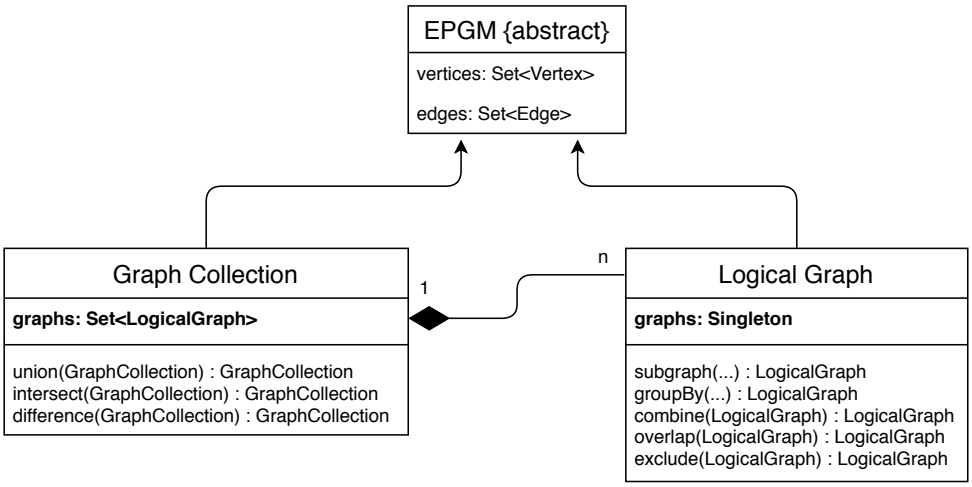


Abbildung 5.1: Modell auf Basis des EPGM

Demnach ist ein *logischer Graph* nicht mehr nur Bestandteil einer EPGM-Instanz, sondern kann selber eine EPGM-Instanz gemäß Definition darstellen. Einzelne EPGM-Operatoren sind dann i. A. entweder auf einem logischen Graphen oder einer *Graph Collection* definiert (siehe Abbildung 5.1). Der Begriff *Graph* wird nachfolgend teilweise synonym zu *logischer Graph* verwendet. Im Falle von logischen Graphen ist die Menge $L = \{G\}$ im EPGM stets einelementig - alle Knoten und Kanten sind in G enthalten. Diese Zuordnung wird im weiteren Verlauf der Arbeit auch als *triviale Graphzugehörigkeit* bezeichnet. Beim Einlesen einer EPGM-Instanz übergibt der Nutzer der Datenquelle, ob er die EPGM-Instanz als Graph Collection oder als logischen

Graphen betrachten möchte. Im weiteren Verlauf eines Workflows ist es i. A. möglich, aus einer Graph Collection einzelne logische Graphen (oder umgekehrt) zu gewinnen. Bei der Implementierung von Operatoren ist daher zu beachten, dass die ursprüngliche Herkunft keinen Einfluss auf die Funktionalität hat. Die Konvertierungs-Operatoren hingegen werden im Rahmen dieser Arbeit nicht betrachtet - sowohl Graph Collections als auch logische Graphen sind im weiteren Verlauf hinsichtlich des Typs abgeschlossen unter Anwendung der jeweiligen Operatoren (siehe Abbildung 5.1).

Surjektivität der Bezeichner- und Eigenschaftenabbildung In Kapitel 2.2 wurden die Operatoren stets über die Knoten- und Kantenmenge bzw. Graphenmenge definiert (eine Ausnahme bildet der Grouping-Operator, da dieser der einzige in dieser Arbeit betrachtetet Operator ist, dessen Ergebnis eine EPGM-Instanz mit vollständig neu erstellen Elementen ist). Sowohl die Bezeichnermenge Σ_B mit zugehöriger Abbildung σ als auch die Schlüsselmenge Σ_A und Wertmenge A mit Abbildung γ der resultierenden EPGM-Instanzen wurde formal nicht definiert. Bei den behandelten Operatoren genügt es in der Theorie, den Definitionsbereich von σ und γ an die Ergebnismengen von Knoten, Kanten und Graphen anzupassen. Das Ergebnis wäre wohldefiniert - allerdings enthielten die resultierenden Instanzen Elemente in Σ_B, Σ_A und A, die nicht im Bildbereich der jeweiligen Funktion liegen. Dies wurde zum Anlass genommen, um in der tatsächlichen Implementierung eine Surjektivität von σ und γ zu fordern, i. e. ungenutzte Elemente dürfen im Folgenden nicht in den Ergebnissen verbleiben. Im weiteren Verlauf wird auch von *induzierten* Eigenschaften bzw. *induzierter* Graphzugehörigkeit gesprochen.

Verwendung der Java Table-API Wie bereits in den Grundlagen (Kapitel 2.4) erläutert, bietet Apache Flink die Java Table-API an. Aufgrund der Nutzer-Eingaben, die erst zur Laufzeit feststehen und kein fester Bestandteil einer Operatorimplementierung sein können, bietet es sich aus softwaretechnischen Gründen an, diese zu verwenden. Im Rahmen der Arbeit wurden darüber hinaus Hilfsklassen zur Erzeugung von relationalen Ausdrücken implementiert, sodass eine tatsächliche Implementierung wie in Listing 5.1 erfolgen kann.

```
1  Table vertices = // ...
2  String label = "person";
3  vertices
4    .select(new ExpressionSeqBuilder()
5      .field(FIELD_VERTEX_LABEL).as("id")
6      .field(FIELD_VERTEX_GRAPH_IDS)
7      .buildSeq()
8    )
9    .where(new ExpressionBuilder()
10     .field(FIELD_VERTEX_LABEL)
11     .equalTo(label)
12     .toExpression()
13   )
```

Listing 5.1: Beispielhafte SQL-Abfrage

Sei die Zeichenkette `label` ein vom Nutzer definiertes Argument, dann kann dieses offensichtlich sehr einfach in der Operatorimplementierung verwendet werden, ohne auf Zeichenkettenoperationen zurückzugreifen. Da sich diese Variante semantisch nicht von einer SQL-Abfrage unterscheidet, wird im weiteren Verlauf der Arbeit nicht mehr auf die Java-Implementierung eingegangen. Alle Implementierungen werden anhand von SQL-Abfragen und Datenflussdiagrammen erläutert.

Implementierung von Datenquelle und -senke Wie in den Grundlagen erläutert, beginnt ein Gradoop-Workflow mit dem Laden einer EPGM-Instanz aus einer Datenquelle. Für die relationale Implementierung wurde (wie bereits in Kapitel 4 erwähnt) angenommen, dass die EPGM-Instanz in CSV-Dateien in einem verteilten Dateisystem abgelegt ist, wobei auf jede Datei eine logische Sicht als Tabelle projiziert wird. Im Zuge dessen wurde eine Datenquelle implementiert, die auf Grundlage eines ebenfalls im Dateisystem abgelegten Schemas alle CSV-Dateien mittels der `CsvTableSource` von Apache Flink lädt und in einer Menge von Tabellen zusammenfasst. Analog wurde eine Datensenke implementiert: Diese nimmt eine Menge von Tabellen entgegen und schreibt die in den Tabellen enthaltenen Daten mithilfe der `CsvTableSink` in verteilte CSV-Dateien. Darüber hinaus wird das Schema in das verteilte Dateisystem geschrieben. Diese Metadaten sind insbesondere für das horizontale Schema erforderlich, da die Menge an Tabellen nicht fest ist (siehe Kapitel 4). `PropertyValue` und `Properties`-Instanzen werden beim Schreiben Base64[3] kodiert bzw. beim Lesen entsprechend dekodiert. Da die gleichzeitige Verarbeitung mehrerer EPGM-Instanzen unterstützt werden muss, wurde ein Abstraktion implementiert, die es erlaubt, mehrere Instanzen gleichzeitig zu halten - dabei werden die tatsächlichen Tabellennamen verborgen (vgl. Eindeutigkeit der Tabellennamen in Apache Flink). Für den Zugriff wurden entsprechende Funktionen bereitgestellt (z. B. `graph.getVertices()`). Im folgenden Kapitel zur Implementierung wurden in den Abfragen stets exemplarische Tabellennamen angegeben.

Wiederverwendung von Zwischenergebnissen Da eine EPGM-Instanz in jedem der in Kapitel 4 eingeführten Schemata durch mehrere Tabellen repräsentiert wird, kann eine neue EPGM-Instanz als Ergebnis eines Operators i. A. nicht mittels einer einzigen relationalen Abfrage gewonnen werden. Vielmehr besteht ein Operator aus mehreren Abfragen, die jeweils einen Teil der resultierenden EPGM-Instanz abfragen, z. B. die resultierende Knotentabelle. Darüber hinaus können Abfragen miteinander verkettet und Zwischenergebnisse in mehreren Abfragen wiederverwendet werden. Verkettungen lassen sich relational durch Unterabfragen ausdrücken. Die SQL-Syntax wurde u. a. zur besseren Strukturierung um das `WITH ... AS`-Konstrukt erweitert, mit dessen sich ganze Tabellenausdrücke bzw. Abfragen benennen und wiederverwenden lassen. Betrachte dazu Listing 5.2: Zunächst wird eine Zwischentabelle T_1 als Selektion auf der Knoten-Tabelle definiert, die anschließend innerhalb der Abfrage der finalen Knoten- und Kantentabellen (`new_vertices` bzw. `new_edges`) referenziert wird. Diese Art der Strukturierung bzw. Darstellung von Abfragen wird aus Gründen der Übersichtlichkeit im weiteren Verlauf der Arbeit herangezogen.

[3] https://tools.ietf.org/html/rfc4648

```
1  WITH T1 AS (
2    SELECT * FROM vertices WHERE vertex_label = 'city'
3  ),
4  new_vertices AS (
5    SELECT * FROM T1
6  ),
7  new_edges AS (
8    SELECT * FROM edges JOIN T1 ON head_id = vertex_id
9  )
```

Listing 5.2: Beispielhafte Verkettung von SQL-Abfragen

Alle üblichen relationalen Datenbanksysteme bzw. die darin enthaltenen Optimierer würden die Zwischentabelle T_1 nur einmal auswerten. Da die relationale Abstraktion in Apache Flink allerdings nur auf logischer Sicht erfolgt und eine Tabellenabfrage einem abgeschlossenen, logischen Plan entspricht (vgl. Kapitel 2.4), führt Apache Flink diesen logischen Plan unter Verwendung der Table-API bei jeder Referenzierung erneut aus. Es findet keine Optimierung über die einzelnen Pläne hinaus statt. Da eine erneute Ausführung i. A. nicht wünschenswert ist, wurde für diese Arbeit eine Prozedur implementiert, welche eine gegeben Tabelle (i. e. das Ergebnis einer Abfrage) entgegen nimmt, in einen verteilten Datensatz (`DataSet`) und wieder zurück in eine Tabelle konvertiert. Dadurch wird sichergestellt, dass das Ergebnis der Tabelle zwischengespeichert und wiederverwendet wird.

Integration in die Referenzimplementierung Innerhalb der Referenzimplementierung bilden die Typen `LogicalGraph` und `GraphCollection` die für den Nutzer sichtbare zentrale Schnittstelle. Da deren Implementierungen sehr stark auf das in Gradoop verwendete GVE-Schema angepasst sind, wurden im Rahmen dieser Arbeit zwei separate Schnittstellen `TableLogicalGraph` und `TableGraphCollection` geschaffen, die dem in Abbildung 5.1 gezeigten Modell entsprechen. Um eine Interoperabilität der beiden Implementierungsarten zu ermöglichen, wurden in jede Richtung Konvertierungsfunktionen implementiert, d. h. eine herkömmliche Gradoop-Instanz lässt sich in eine relational abstrahierte Instanz beliebigen Schemas konvertieren - und umgekehrt. Das gewünschte Zielschema kann dabei vom Nutzer gewählt werden.

5.2 Subgraph

In Kapitel 2 wurde der Subgraph-Operator bereits definiert. Wie an jener Stelle erwähnt, bietet die Referenzimplementierung dem Nutzer die Möglichkeit, eine beliebige Prädikatsfunktion anzugeben. Darin kann z. B. auf das `Vertex`-POJO zugegriffen werden. Die Table-API von Flink bietet keine Möglichkeit, neben den in den Grundlagen vorgestellten Typen von nutzerdefinierten Funktionen, eine spezielle *Prädikatsfunktion* zu definieren. Die relationale Selektion hingegen nimmt nur einen relationalen booleschen Ausdruck entgegen. Theoretisch ist es nun möglich, dem Nutzer eine Menge an vordefinierten Ausdrucks-Möglichkeiten bereitzustellen - im Rahmen dieser Arbeit wurde allerdings nur eine feste Operatorsignatur implementiert, die eine Selektion basierend auf Bezeichnern ermöglicht. Dabei nimmt der Operator jeweils eine Liste aus Bezeichnern für Knoten und Kanten entgegen. Ein Knoten bzw. eine Kante ist in der jeweiligen reduzierten Menge enthalten, wenn sein bzw. ihr Bezeichner in jener Liste enthalten ist. Es sei angemerkt, dass diese Wahl die Ausdrucksmächtigkeit des Operators einschränkt.

Diese gewählte Logik lässt sich mit dem relationalen booleschen `IN`-Operator umsetzen. So lassen sich die reduzierte Knoten- und Kantenmenge über eine einfache relationale Selektion berechnen. Wie in der Definition des Operators gefordert, muss abschließend sichergestellt werden, dass es keine inkonsistenten Kanten gibt. Dies lässt sich über einen zweifachen (Semi-) Verbund der reduzierten Kantenmenge mit der Knotenmenge realisieren. Listing 5.3 zeigt die relationale Implementierung des Subgraph-Operators im GVE Schema in Form von SQL-Abfragen, wobei `new_vertices` und `new_edges` die Knoten- bzw. Kantenmengen des Ergebnisgraphen sind. Um eine gültige EPGM-Instanz zu erzeugen, muss innerhalb des Operators zusätzlich eine neue ID für den Ergebnis-Graphen erzeugt werden. Diese sei hier beispielhaft `G_NEW`. Zusätzlich zu den Knoten und Kanten muss die EPGM-Instanz (i. e. die Menge von Tabellen) also noch eine 1-elementige `Graphs`-Tabelle erhalten. Da sich neue Tabellen über die Table-API nicht erzeugen lassen, muss dafür zunächst ein `DataSet` erzeugt und anschließend konvertiert werden. Darüberhinaus muss die Graphzugehörigkeit der einzelnen Elemente zum neuen logischen Graphen in Form von 1-elementigen `GradoopIdSet`'s an den Elementen selber hinzugefügt werden (triviale Graphzugehörigkeit). Diese werden mit der Skalarfunktion `PARSE_GRADOOP_ID_SET` erzeugt (siehe Zeilen 5 und 16).

Das Erzeugen einer neuen Graph-ID bzw. die Integration dieser in eine neue EPGM-Instanz ist bei allen Operatoren auf logischen Graphen identisch und kein elementarer Bestandteil der Operator-Implementierung. Um einen Überblick über die relationale Funktionsweise der Operators zu verschaffen, ist der Datenfluss in Abbildung 5.2 visualisiert. Dabei werden nur die Knoten- und Kanten-Tabelle berücksichtigt. Die Tabelle T_1 entspricht `new_vertices` und ist markiert dargestellt, da sie zweifach referenziert wird und daher aus Performance-Gründen zwischengespeichert werden sollte (vgl. Grundlagen dieses Kapitels).

```
WITH new_vertices AS (
  SELECT
    vertex_id,
    vertex_label,
    PARSE_GRADOOP_ID_SET('G_NEW') AS vertex_graph_ids,
    vertex_properties
  FROM vertices
  WHERE vertex_label IN ('...')
),
T2 AS (
  SELECT
    edge_id,
    tail_id,
    head_id,
    edge_label,
    PARSE_GRADOOP_ID_SET('G_NEW') AS edge_graph_ids,
    edge_properties
  FROM edges
  WHERE edge_label IN ('...')
),
new_edges AS (
  SELECT edge_id, head_id, tail_id, edge_label, edge_graph_ids, edge_properties
    FROM T2
    JOIN T1 ON tail_id = vertex_id
    JOIN T1 ON head_id = vertex_id
)
```

Listing 5.3: Subgraph Operator als SQL-Abfragen im GVE-Schema

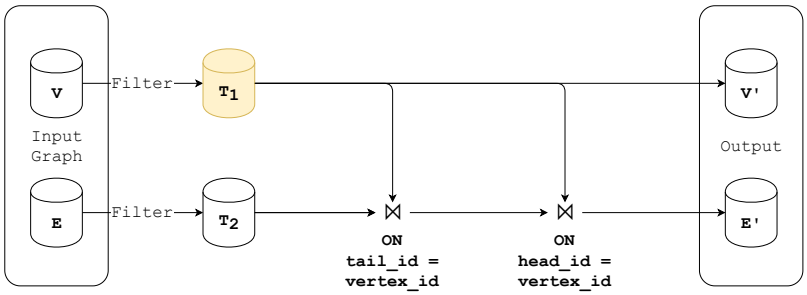


Abbildung 5.2: Datenfluss des grundlegenden Subgraph-Operators

Die Selektion der reduzierten Mengen im vertikalen Schema unterscheidet sich vom GVE Schema nur darin, dass die Spalten `vertex_graph_ids` und `vertex_properties` bzw. `edge_graph_ids` und `edge_properties` nicht vorhanden sind und somit auch nicht projiziert werden. Stattdessen muss nach Bildung der reduzierten Knoten- und Kanten-Tabelle ein (Semi-) Verbund dieser mit den entsprechenden Eigenschaften-Tabellen gebildet werden. Damit wird erreicht, dass die EPGM-Instanz nur noch Eigenschaften enthält, die tatsächlich verwendet bzw. referenziert werden (Surjektivität). Listing 5.4 beinhaltet die dafür zusätzlich benötigten Abfragen.

```
new_vertex_properties AS (
  SELECT property_vertex_id, vertex_property_name, vertex_property_value
  FROM vertex_property_values
  JOIN new_vertices ON property_vertex_id = vertex_id
),
new_edge_properties AS (
  SELECT property_edge_id, edge_property_name, edge_property_value
  FROM edge_property_values
  JOIN new_edges ON property_edge_id = edge_id
)
```

Listing 5.4: Zusätzliche SQL-Abfragen im vertikalen Schema für Eigenschaften

Abschließend muss die Graphzugehörigkeit der Knoten und Kanten zu dem neu erzeugten logischen Graphen mit ID `G_NEW` gebildet werden. Da diese sowohl im vertikalen als auch im horizontalen Schema in der Tabelle `vertices_graphs` bzw. `edges_graphs` abgelegt wird, können die in Listing 5.5 gezeigten Abfragen in den Implementierungen auf beiden Schemata angewandt werden. Dabei wird die neue Graph-ID mittels der Skalarfunktion `PARSE_GRADOOP_ID` in eine `GradoopId` konvertiert. Die 1-elementige `graphs`-Tabelle wird analog zum GVE-Schema erzeugt, wobei sie offensichtlich keine `graph_properties`-Spalte enthalten darf. Somit ist die vollständige Menge an Tabellen erzeugt und stellt eine EPGM-Instanz dar.

```
new_vertices_graphs AS (
  SELECT
    vertex_id AS graph_vertex_id,
    PARSE_GRADOOP_ID('G_NEW') AS vertex_graph_id
  FROM new_vertices
),
new_edges_graphs AS (
  SELECT
    edge_id AS graph_edge_id,
    PARSE_GRADOOP_ID('G_NEW') AS edge_graph_id
  FROM new_edges
)
```

Listing 5.5: Zusätzliche SQL-Abfragen im vertikalen/horizontale Schema für Graphzugehörigkeit

In Abbildung 5.3 ist der Datenfluss der Implementierung im vertikalen Schema dargestellt, wobei die Erzeugung der Graphzugehörigkeit wieder vernachlässigt sei. In der Abbildung ist ersichtlich, dass die Implementierung auf der GVE-Implementierung basiert (mit Umrandung gekennzeichnet). Des Weiteren werden die neu berechneten Kanten in der Berechnung der neuen Kanten-Eigeschaften referenziert und sollten daher zwischengespeichert werden (Tabelle T_3 ist entsprechend markiert).

Wie in Kapitel 4.5 erläutert, sind die Eigenschaften im horizontalen Schema in Tabellen je Eigenschaften-Schlüssel ausgelagert, wobei die zu den Eigenschaftswerten gehörenden Knoten, Kanten oder Graphen polymorph referenziert werden. Daraus folgt, dass ein (Semi-) Verbund

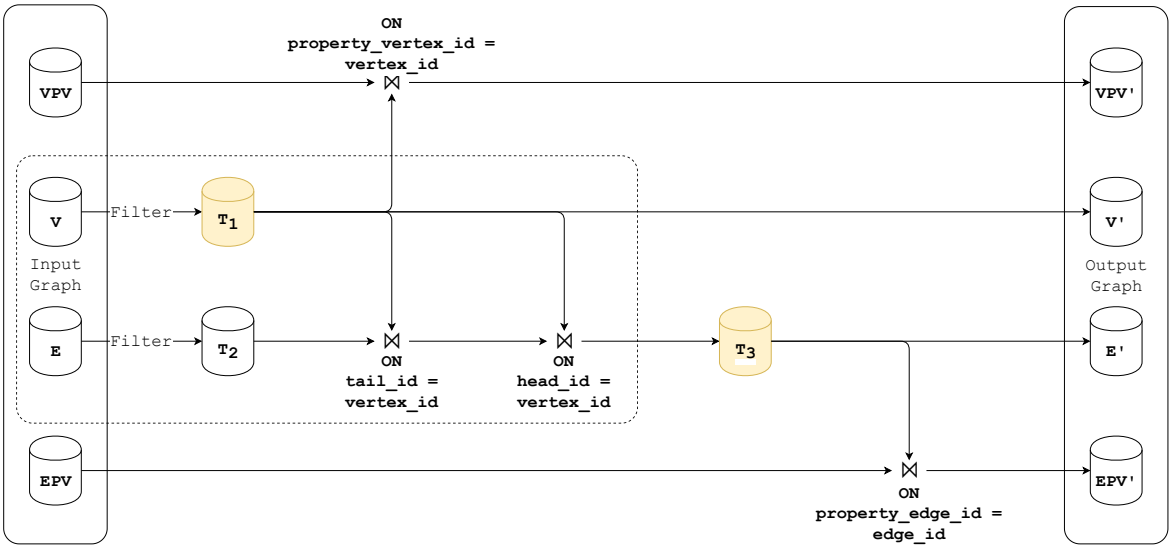


Abbildung 5.3: Datenfluss des Subgraph-Operators im vertikalen Schema

jeder der Eigenschaftstabellen sowohl mit der reduzierten Kanten- als auch mit der (reduzierten) Knotenmenge gebildet werden muss. Um die Eigenschaften-Tabellen nicht mehrfach lesen zu müssen, bietet es sich an, zunächst eine Vereinigung der Knoten- und Kanten-IDs zu bilden (siehe Zeilen 1 bis 7 in Listing 5.6). Anschließend kann für jede Eigenschaften-Tabelle ein (Semi)-Verbund mit der vereinigten Menge gebildet werden. Eine entsprechende SQL-Abfrage ist am Beispiel der Eigenschaft `property_1` in den Zeilen 8 bis 12 von Listing 5.6 gezeigt.

```
1  T4 AS (
2    SELECT vertex_id AS element_id
3    FROM new_vertices
4    UNION ALL
5    SELECT edge_id AS element_id
6    FROM new_edges
7  ),
8  new_property_1 AS (
9    SELECT property_1_element_id, property_1_property_value
10   FROM property_1
11   JOIN T4 ON property_1_element_id = element_id
12 )
```

Listing 5.6: Zusätzliche SQL-Abfragen im horizontalen Schema für Eigenschaften

Da die reduzierten Knoten- bzw. Kantentabellen zusätzlich in der Bildung der Vereinigung referenziert werden, sollten sie analog zur Implementierung im vertikalen Schema zwischengespeichert werden. Da die Vereinigung bei n Eigenschaften-Schlüsseln n-fach referenziert wird, sollte diese ebenfalls zwischengespeichert werden. Abbildung 5.4 veranschaulicht den Datenfluss der Implementierung basierend auf dem horizontalen Schema. Alle zwischengespeicherten Tabellen sind markiert - die zugrundeliegende GVE-Implementierung ist abermals mit Umrandung gekennzeichnet. Zusammen mit der analog zu den anderen Implementierungen gebildeten Graphen-Tabelle sowie der trivialen Graphzugehörigkeits-Tabellen (siehe Listing 5.5) ergibt sich eine vollständige EPGM-Instanz im horizontalen Schema.

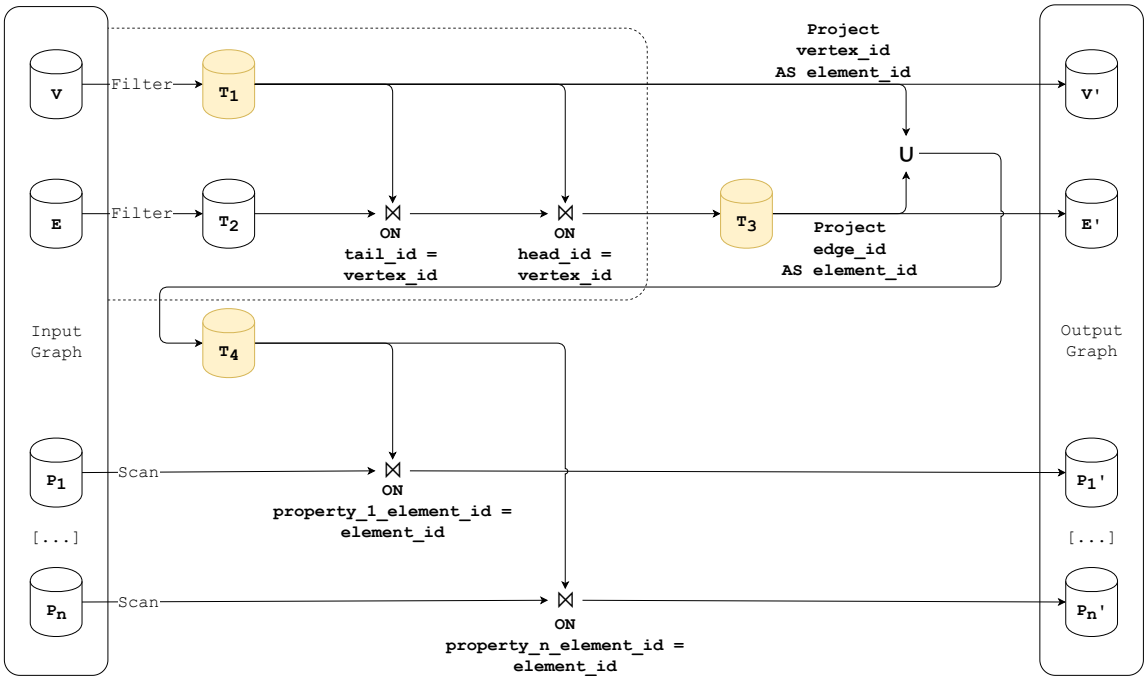


Abbildung 5.4: Datenfluss des Subgraph-Operators im horizontalen Schema

5.3 Mengenoperatoren auf logischen Graphen

Im folgenden Abschnitt wird die Implementierung der Mengenoperatoren auf logischen Graphen (Combination, Overlap und Exclusion) erläutert. Wie zu Beginn des Kapitels eingeführt, wird als Ausgabe der Operatoren unabhängig vom zugrundeliegenden Schema stets eine von den Knoten und Kanten induzierte EPGM-Instanz gefordert. Die Berechnungen der Knoten- und Kantenmengen in den verschiedenen Schemata unterscheiden sich indessen häufig nicht. Viele Implementierungen basieren auf der GVE-Implementierung und ergänzen diese um eine Berechnung der von der Surjektivität geforderten induzierten Eigenschaften und Graphzugehörigkeit. Im vorangegangen Abschnitt wurden diese Berechnung bereits am Beispiel des Subgraph-Operators erläutert. Listing 5.4 beinhaltet die Abfrage der induzierten Eigenschaftswerte im vertikalen Schema; in Listing 5.6 werden die induzierten Eigenschaftstabellen im horizontalen Schema abgefragt während Listing 5.5 die Berechnung der trivialen Graphzugehörigkeit im vertikalen und horizontalen Schema zeigt. Diese Erweiterungen der Implementierung im GVE-Schema werden in den nachfolgenden Implementierungen wieder aufgegriffen.

Die in Abschnitt 2.2.2 definierten Mengenoperatoren auf logischen Graphen beruhen auf mengentheoretischen Operatoren. Für die Implementierung gilt es zu definieren, wann für zwei Elemente der Mengen (z. B. zwei Knoten) Gleichheit gilt. Analog zur Referenzimplementierung wird dazu i. A. die ID herangezogen, i. e. es wird davon ausgegangen, dass Bezeichner und Eigenschaften durch die ID eindeutig bestimmt sind. Bei Kanten wird zusätzlich davon ausgegangen, dass Quell- und Zielknoten durch die Kanten-ID eindeutig bestimmt sind. Die Graphzugehörigkeit bleibt hingegen unberücksichtigt, da die zwei logischen Graphen i. A. aus unterschiedlichen Datenquellen bzw. Graph Collections stammen können.

In realen Daten ist es vorstellbar, dass zu einem Eigenschaftenschlüssel in beiden Eingabegraphen unterschiedliche Eigenschaftwerte existieren. Dieser Fall wird mit Blick auf den Umfang der Arbeit nicht weiter behandelt

5.3.1 Combination

Der Combination-Operator berechnet einen logischen Graphen, der alle Knoten und Kanten der beiden Eingabe-Graphen vereinigt. Zur Berechnung der mengentheoretischen (duplikatfreien) Vereinigung existiert der relationale Vereinigungsoperator (`UNION`). In Listing 5.7 ist exemplarisch aufgeführt, wie die gewünschte Knotenmenge relational abgefragt werden kann. Dabei sei `vertices_1` die Knoten-Tabelle des ersten und `vertices_2` die Knoten-Tabelle des zweiten Eingabe-Graphen. Die beiden Knotenmengen werden dabei zunächst auf ID, Bezeichner und Eigenschaften sowie eine konstante 1-elementige Menge von Graph-IDs projiziert. Dadurch werden die Knoten bereits dem Ausgabegraphen mit ID `G_NEW` zugeordnet. Dieses Vorgehen wurde bereits im Kontext der Subgraph-Implementierung erläutert. Gemäß Definition werden die Kantenmengen analog zu den Knotenmengen vereinigt - somit ist die EPGM-Instanz im GVE-Schema vollständig. Der Datenfluss unter Vernachlässigung der Graphzugehörigkeit ist in Abbildung 5.5 veranschaulicht.

```
1  SELECT
2    vertex_id,
3    vertex_label,
4    PARSE_GRADOOP_ID_SET('G_NEW') AS vertex_graph_ids,
5    vertex_properties
6  FROM vertices_1
7  UNION
8  SELECT
9    vertex_id,
10   vertex_label,
11   PARSE_GRADOOP_ID_SET('G_NEW') AS vertex_graph_ids,
12   vertex_properties
13 FROM vertices_2;
```

Listing 5.7: SQL-Abfrage der neuen Knoten-Menge im GVE-Schema

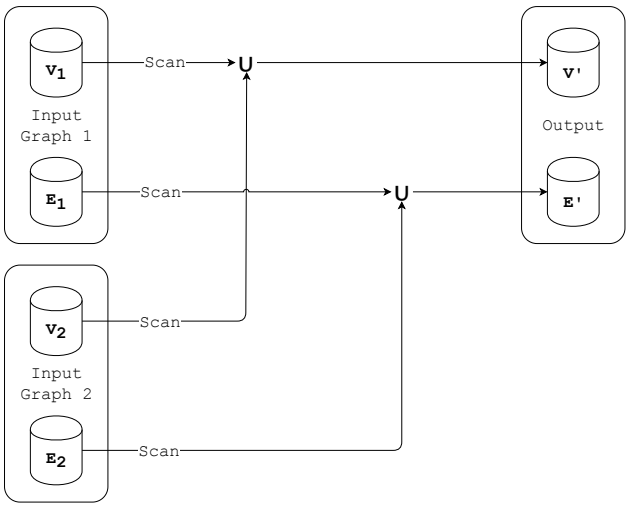


Abbildung 5.5: Datenfluss des Combination-Operators im GVE-Schema

Im vertikalen und horizontalen Schema weicht die Berechnung der Knoten- und Kantenmenge geringfügig ab. Da die Tabellen in diesen Schemata weniger Spalten haben, kann vor der Vereinigung folglich auf eine kleinere Menge an Attribute projiziert werden. Im Falle von Knoten muss z. B. nur noch auf ID und Bezeichner projiziert werden. Die Zeilen 4 und 5 bzw. 11 und 12 aus Listing 5.7 entfallen also für die Implementierung im vertikalen und horizontalen Schema.

Die triviale Graphzugehörigkeit, i. e. die Tabellen `vertices_graphs` und `edges_graphs`, wird analog zum Subgraph-Operator gebildet. Aus allen anderen Tabellen wird jeweils die Vereinigung gebildet. Existieren im horizontalen Schema in einem der Eingabegraphen Eigenschafts-Tabellen, die im anderen nicht existieren, werden diese ohne Vereinigung übernommen - existiert für einen Eigenschaftenschlüssel in jedem der Eingabegraphen eine Tabelle, wird aus diesen augenscheinlich die Vereinigung gebildet.

5.3.2 Overlap

Der Overlap-Operator berechnet alle Knoten und Kanten, die in beiden Eingabe-Graphen enthalten sind. Jedes Element des Ausgabegraphen ist somit per Definition auch im ersten Eingabegraphen enthalten. Bildet man einen Semi-Verbund der Knoten- bzw. Kantenmengen des ersten Eingabegraphen mit den entsprechenden Mengen des Zweiten, erhält man das gewünschte Ergebnis im GVE-Schema. Die Graphzugehörigkeit zum Ausgabegraphen wird wieder analog zum Subgraph-Operator gebildet. Die grobe Funktionsweise des Operators im GVE-Schema ist in Abbildung 5.6 visualisiert.

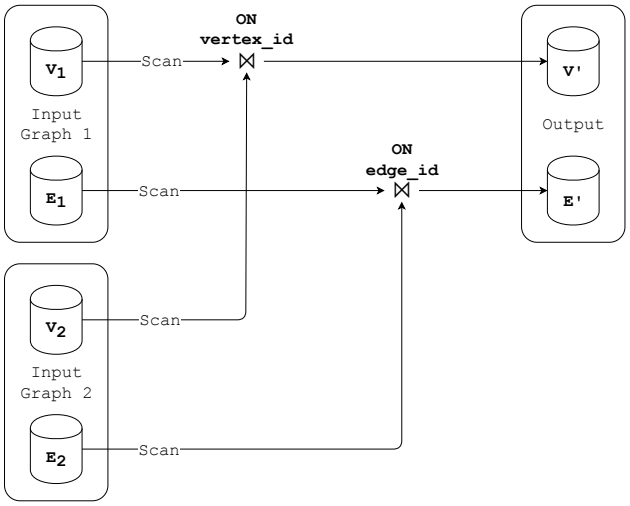


Abbildung 5.6: Datenfluss des Overlap-Operator

Wie zu Beginn des Abschnitts angekündigt, kann für die Berechnung der induzierten Eigenschaften im vertikalen bzw. horizontalen Schema auf dieselben Abfragen zurückgegriffen werden, die innerhalb der Subgraph-Implementierung verwendet werden. Dies ist möglich, da jede Eigenschaft bzw. jeder Eigenschaftswert des Ergebnisgraphen bereits im ersten Eingabegraphen enthalten sein muss. Es müssen also nur die Eigenschaften eines der Eingabegraphen geladen und ein Verbund gebildet werden - die Tabellen des anderen Eingabegraphen müssen nicht eingelesen werden und bleiben gänzlich unberührt. Der Operator wurde so implementiert, dass stets die Eigenschaften des ersten Eingabegraphen verwendet werden. Die Graphzugehörigkeit im vertikalen und horizontalen Schema wird analog zur Subgraph-Implementierung gebildet. Somit ist die Implementierung in diesen beiden Schemata auf Basis der GVE-Implementierung vollständig.

5.3.3 Exclusion

Der Exclusion-Operator berechnet den durch die Differenzmenge der Knotenmengen induzierten Graphen. Dieser induzierte Graph kann nur Knoten und Kanten aus dem ersten Eingabegraphen enthalten - es muss aus dem zweiten Graphen folglich lediglich die Knoten-Tabelle eingelesen werden, um die Differenzmenge zu bilden. Dabei genügt es die Differenzmenge der Knoten-IDs zu bilden. Diese kann mit dem entsprechenden relationalen Operator (`MINUS`) gebildet werden.

Daraufhin kann ein Semi-Verbund der Knotenmenge des ersten Graphen mit den berechneten Knoten-IDs gebildet werden (siehe Listing 5.8).

```
1  SELECT
2    vertex_id,
3    vertex_label,
4    PARSE_GRADOOP_ID_SET('G_NEW') AS vertex_graph_ids,
5    vertex_properties
6  FROM vertices_1
7  JOIN (
8    SELECT vertex_id AS tmp
9    FROM vertices_1
10   MINUS
11   SELECT vertex_id AS tmp
12   FROM vertices_2
13 ) ON vertex_id = tmp;
```

Listing 5.8: SQL-Abfrage der neuen Knoten-Menge im GVE-Schema

Die induzierte Kantenmenge kann abschließend über einen zweifachen Semi-Verbund mit der neuen Knotentabelle berechnet werden. Abbildung 5.7 veranschaulicht den vollständigen Datenfluss der Implementierung im GVE-Schema. Die Tabelle T_1 (entspricht dem Ergebnis der Abfrage aus Listing 5.8) sollte aufgrund der zweifachen Referenzierung zwischengespeichert werden.

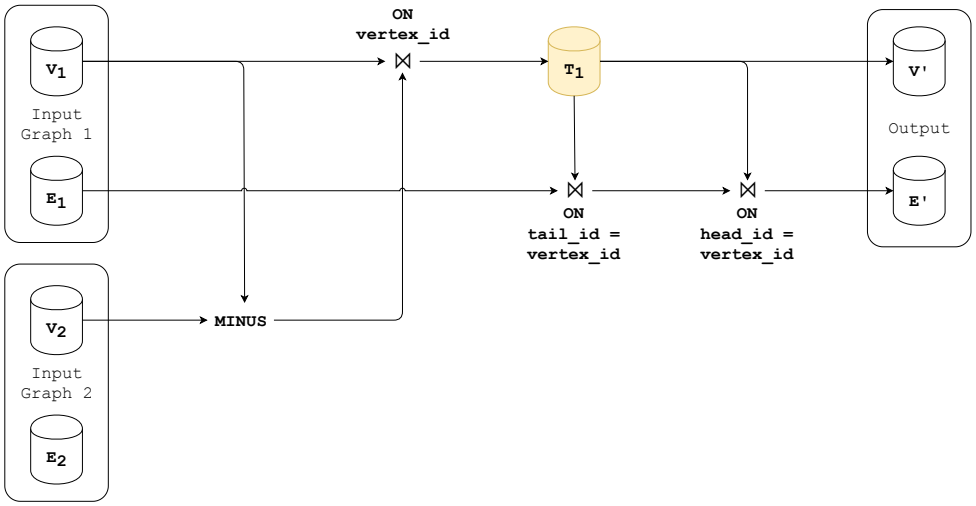


Abbildung 5.7: Datenfluss des Exclusion-Operators

Die Implementierung im vertikalen und horizontalen Schema erfolgt analog zur Implementierung des Overlap-Operators basierend auf der GVE-Implementierung. Da der Exclusion-Operator im Gegensatz zum Overlap nicht kommutativ ist, ist es zwingend erforderlich, die Eigenschaften des ersten Eingabegraphen zu verwenden. Da diese bereits bei der Overlap-Implementierung herangezogen wurden, ergeben sich keine Änderungen.

5.4 Mengenoperatoren auf Graph Collections

In diesem Abschnitt wird die Implementierung der Mengenoperatoren auf Graph Collections besprochen. Diese beruhen auf mengentheoretischen Operationen auf den in zwei EPGM-Instanzen enthaltenen Mengen an logischen Graphen. In diesem Abschnitt wird also erstmalig die Graphzugehörigkeit auf nicht-triviale Weise verarbeitet. Als Ausgabe der Operatoren wird eine von Graphen, Knoten und Kanten induzierte EPGM-Instanz gefordert, i. e. es müssen jeweils die induzierten Eigenschaften berechnet werden.

Hinsichtlich der Mengenoperatoren gilt es einige Voraussetzungen festzuhalten, die über die formale mengentheoretische Definition hinausgehen. Diese Voraussetzungen wurden in der Referenzimplementierung gewählt und sollen für eine optimale Vergleichbarkeit auch im Rahmen dieser Arbeit gelten:

1. Die Gleichheit von logischen Graphen sei auf Basis der ID definiert. Es wird also implizit angenommen, dass die gesamte Struktur eines logischen Graphen (Knoten und Kanten) sowie Bezeichner und Eigenschaften durch seine ID eindeutig bestimmt sind - dies gelte auch über mehrere Datenquellen hinweg.

2. Analog zu den Mengenoperatoren auf logischen Graphen hängen auch alle Attribute der Knoten und Kanten funktional von ihrer ID ab.

3. Aus Implementierungsgründen soll zwar die Graph-Menge gemäß der Mengenoperation neu bestimmt werden - die Graphzugehörigkeit eines Knotens bzw. einer Kante soll allerdings nicht verändert werden, i. e. die resultierende EPGM-Instanz enthält weiterhin die Information, dass ein Knoten V in Graph G enthalten ist, obwohl G selber nicht mehr in der Menge L an logischen Graphen enthalten ist. Dies hat den Vorteil, dass im weiteren Verlauf des Workflows mit dieser Information gearbeitet werden kann, ohne die Graph Collection vollständig neu einlesen zu müssen. Aus formaler Sicht ist die Graphzugehörigkeit allerdings nicht mehr konsistent.

5.4.1 Union

Der Union-Operator berechnet eine Graph Collection, die alle logischen Graphen der beiden Eingabe-Collection vereinigt. Wie der Definition in Kapitel 2.2.3 zu entnehmen ist, werden neben den Mengen an Graphen auch die Knoten- und Kantenmengen vereinigt. Die Implementierung des Union-Operators unterscheidet sich also nur minimal von der des Combination-Operators auf zwei logischen Graphen. Es werden für jede Tabelle T (unabhängig vom Schema) die Vereinigung der Tabellen T_1 und T_2 aus den Eingabe-Collections gebildet. Dazu wird der relationale Vereinigungsoperator (`UNION`) herangezogen. Hierbei findet die Voraussetzung Anwendung, dass alle Attribute eines Datensatzes über beide Eingabedatensätze hinweg eindeutig durch die ID bestimmt sind.

5.4.2 Intersection und Difference

Die beiden Operatoren Intersection und Difference sind über die Schnitt- bzw. Differenzmenge der Mengen an logischen Graphen definiert. Da sich die Definitionen und Implementierungen darüberhinaus nicht unterscheiden, werden sie hier gemeinsam abgehandelt. Sei `graphs_1` die Graphen-Tabelle in der ersten Eingabe-Collection und `graphs_2` die der zweiten, dann lässt sich die Menge der Graph-IDs mithilfe der entsprechenden relationalen Operatoren (`INTERSECT` bzw. `MINUS`) unabhängig vom zugrundeliegenden Schema berechnen (siehe Listing 5.9). Da die Zwischentabelle T_1 nachfolgend mehrfach referenziert wird, sollte sie zwischengespeichert werden.

```
1  WITH T1 AS (
2    SELECT graph_id FROM graphs_1
3    INTERSECT -- MINUS
4    SELECT graph_id FROM graphs_2
5  )
```

Listing 5.9: SQL-Abfrage der neuen Graph-IDs

Da T_1 nur Graph-IDs enthält, bedarf es einer weiteren Abfrage, um die finale Graphen-Tabelle der Ergebnis-Instanz zu erhalten. Dazu genügt es, einen Semiverbund der Graphen-Tabelle der ersten Eingabe-Collection mit T_1 zu bilden. Die nötige Abfrage im GVE-Schema zeigt Listing 5.10.

```
1  SELECT graph_id, graph_label, graph_properties
2  FROM graphs_1
3  JOIN T1 ON graph_id = graph_id;
```

Listing 5.10: SQL-Abfragen im GVE-Schema

Ausgehend von Tabelle T_1 müssen nun noch die Knoten- und Kantenmengen bestimmt werden. Dazu muss die Graphzugehörigkeit herangezogen werden, i. e. es ergeben sich grundlegende Unterschiede zwischen der Implementierung im GVE-Schema und derjenigen im vertikalen bzw. horizontalen Schema.

Für die Implementierung im GVE-Schema muss eine Tabellenfunktion definiert werden (siehe Kapitel 2.4). Diese wurde so implementiert, dass sie ein mengenwertiges `GradoopIdSet` in einzelne `GradoopId`s zerlegt. Betrachte zur Verdeutlichung die Tabellen 5.1. Während die Knoten-Tabelle der ersten Eingabe-Collection (a) noch dem GVE-Schema entspricht, existieren nach Aufruf der Tabellenfunktion für einen Knoten jeweils mehrere Zeilen (b). Für Knoten $V5$ enthält das Ergebnis z. B. drei Zeilen, die sich nur anhand der Werte in der `vertex_graph_id`-Spalte unterscheiden.

Abschließend müssen ein Verbund dieser Zwischentabelle mit T_1 gebildet, auf die Knoten-Attribute projiziert und Duplikate entfernt werden. Listing 5.11 beinhaltet die vollständige Abfrage. Wie bereits in Kapitel 2.4 erläutert, werden die nutzerdefinierten Tabellenfunktionen über einen Verbund in eine relationale Abfrage integriert. In diesem konkreten Fall wird formal ein Verbund mit (dem Ergebnis) der Tabellenfunktion `EXPAND_GRADOOP_ID_SET` gebildet, wobei die Funktion auf `vertex_graph_ids` arbeitet und die neue Spalte mit `vertex_graph_id` bezeichnet

(a) Beispielhafte Knotentabelle

vertex_id	vertex_label	vertex_graph_ids	vertex_properties
V4	Person	[1, 2]	{name: Carol}
V5	Person	[2, 3, 4]	{name: Alice}
...	...	...	...

(b) Ergebnis der Tabellenfunktion

vertex_id	vertex_label	vertex_graph_ids	vertex_properties	vertex_graph_id
V4	Person	[1, 2]	{name: Carol}	1
V4	Person	[1, 2]	{name: Carol}	2
V5	Person	[2, 3, 4]	{name: Alice}	2
V5	Person	[2, 3, 4]	{name: Alice}	3
V5	Person	[2, 3, 4]	{name: Alice}	4
...	...	...	...	...

Tabelle 5.1: Beispieltabellen zur Erklärung der Tabellenfunktion

wird. Die Graphzugehörigkeit `vertex_graph_ids` wird gemäß der Voraussetzungen unverändert projiziert.

```
1  SELECT DISTINCT vertex_id, vertex_label, vertex_properties, vertex_graph_ids
2  FROM vertices_1
3  LEFT JOIN LATERAL TABLE(EXPAND_GRADOOP_ID_SET(vertex_graph_ids))
4    AS T(vertex_graph_id) ON TRUE;
5  JOIN T1 ON vertex_graph_id = graph_id
```

Listing 5.11: SQL-Abfragen im GVE-Schema

Analog kann die neue Kanten-Menge abgefragt werden. Abbildung 5.8 zeigt den gesamten Datenfluss der Operator-Implementierung im GVE-Schema. Offensichtlich besteht die Implementierung aus der Mengenoperation und drei Verbund-Operationen. Es genügt, einzig die Graphen-Tabelle der zweiten Eingabe-Collection einzulesen - Knoten und Kanten dieser Collection können per Definition nicht im Ergebnis enthalten sein.

Im Gegensatz zum GVE-Schema muss weder im vertikalen noch im horizontalen Schema eine Tabellenfunktion zur Auflösung der Graphzugehörigkeit herangezogen werden. Diese wird bereits normalisiert in der `vertices_graphs` bzw. `edges_graphs` vorgehalten. Um alle Knoten bzw. Kanten zu identifizieren, die in den in T_1 identifizierten logischen Graphen enthalten sind, genügt es einen Verbund zwischen T_1 und jener `vertices_graphs` bzw. `edges_graphs`-Tabelle zu bilden. Ergebnis sei z. B. eine Tabelle `new_vertex_ids`, die alle entsprechenden Knoten-IDs enthalte. Um zusätzlich die Knotenbezeichner zu erhalten, muss ein weiterer Verbund mit der Knotentabelle der ersten Eingabe-Collection gebildet werden. Listing 5.12 zeigt die entsprechende SQL-Abfrage der neuen Knoten. Die Tabellen `vertices_graphs` bzw. `edges_graphs` werden gemäß Voraussetzung unverändert in der Ergebnis-Instanz übernommen.

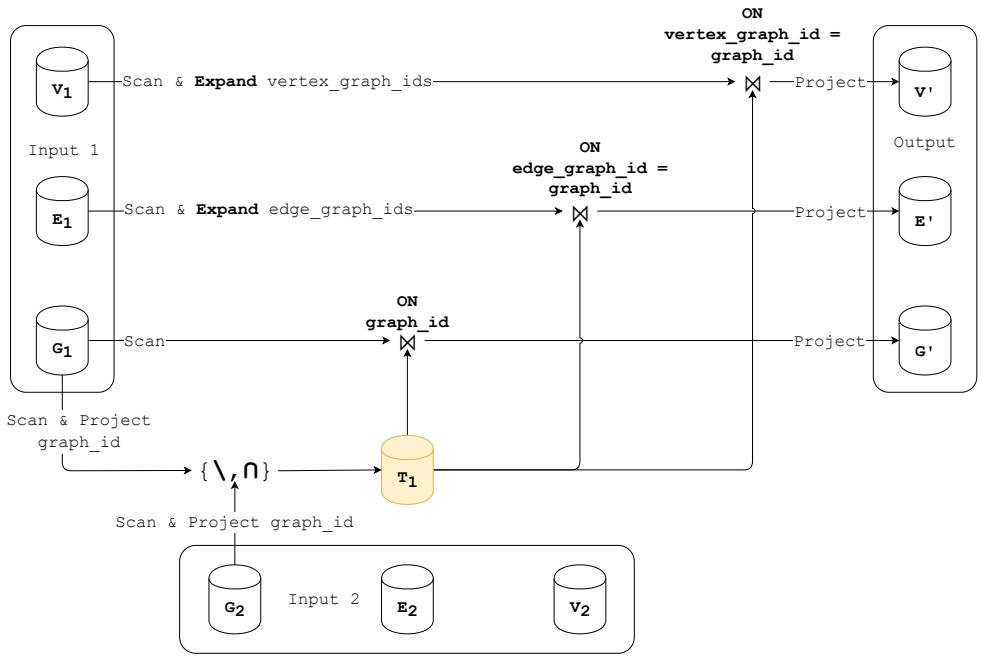


Abbildung 5.8: Datenfluss des Difference- bzw. Intersection-Operators im GVE-Schema

```
1  WITH new_vertex_ids AS (
2    SELECT graph_vertex_id
3    FROM vertices_graphs
4    JOIN T1 ON vertex_graph_id = graph_id
5  )
6  SELECT vertex_id, vertex_label
7  FROM vertices_1
8  JOIN new_vertex_ids ON vertex_id = graph_vertex_id
```

Listing 5.12: SQL-Abfrage im vertikalen und horizontalen Schema

Im Vergleich zur Implementierung im GVE-Schema muss also ein zusätzlicher Verbund-Operator verwendet werden, um die neue Knotentabelle zu bestimmen. Gleiches gilt für die Abfrage der neuen Kanten. Abbildung 5.9 zeigt den Datenfluss der Implementierung des Difference bzw. Intersection-Operators im vertikalen und horizontalen Schema. Auch in diesem Schema muss einzig die Graphen-Tabelle der zweiten Eingabe-Collection gelesen bzw. weiterverarbeitet werden.

Bislang unberücksichtigt ist die Berechnung der induzierten Eigenschaften. Diese erfolgt analog zu den Überlegungen im Abschnitt über die Implementierung des Subgraph-Operators. Im Unterschied zu den Operatoren auf logischen Graphen können nun auch den Graphen Eigenschaftswerte zugeordnet sein. Listing 5.13 zeigt die SQL-Abfrage der induzierten Graph-Eigenschaften bzw. Eigenschaftswerte im vertikalen Schema, wobei die Tabelle `graph_property_values` die Graph-Eigenschaften der ersten Eingabe-Collection beinhalte.

```
1  SELECT property_graph_id, graph_property_name, graph_property_value
2  FROM graph_property_values
3  JOIN T1 ON property_graphs_id = graph_id
```

Listing 5.13: SQL-Abfrage der induzierten Graph-Eigenschaften im vertikalen Schema

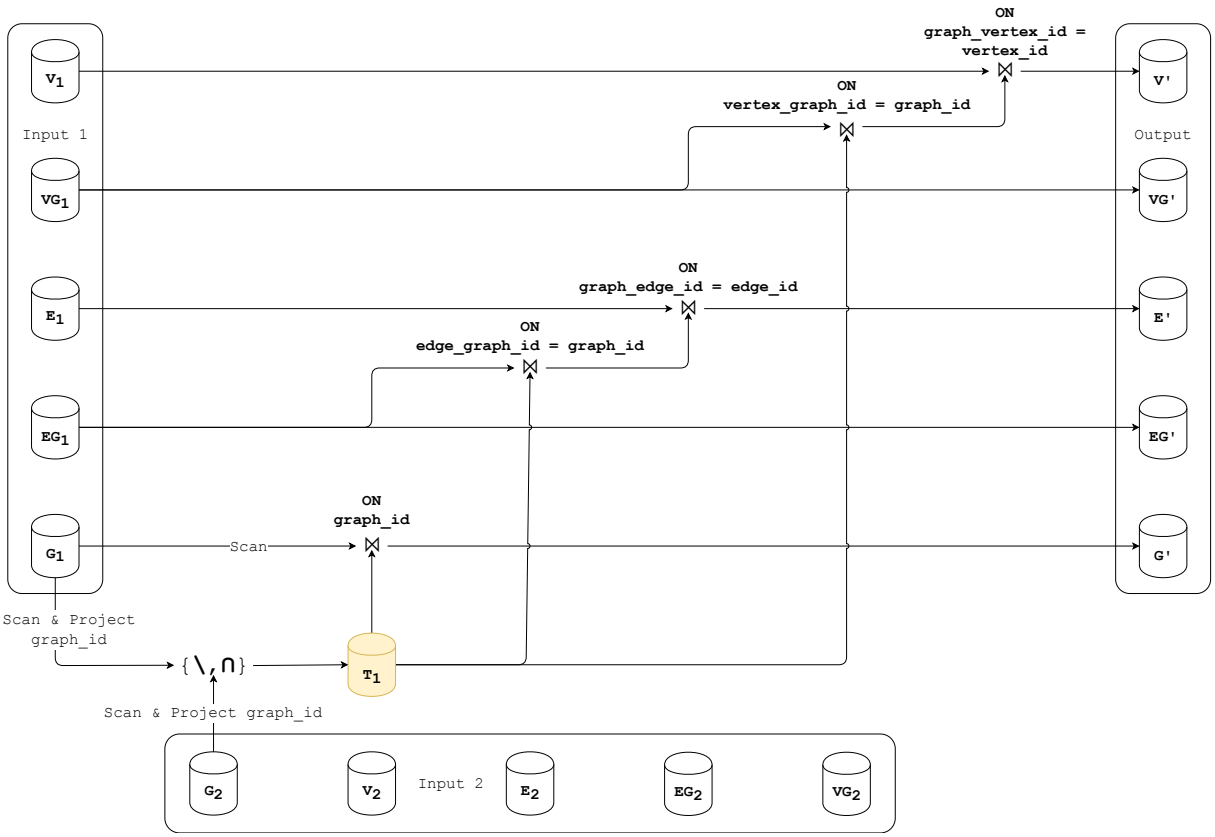


Abbildung 5.9: Datenfluss des Difference- bzw. Intersection-Operators im vertikalen bzw. horizontalen Schema

Aufgrund der polymorphen Referenzierung der Elemente in den Eigenschaftstabellen im horizontalen Schema, müssen zur Berechnung der induzierten Eigenschaftswerte zunächst die ID-Mengen der möglichen referenzierten Elemente vereinigt werden (siehe Listing 5.4 im Kapitel über die Implementierung des Subgraph-Operators). Um auch die Graph-Eigenschaften abzudecken, müssen also zusätzlich zu den Knoten- und Kanten-IDs auch die Graph-IDs herangezogen werden. Listing 5.14 zeigt die dafür notwendige SQL-Abfrage.

```
1  SELECT vertex_id AS element_id
2  FROM new_vertex_ids
3  UNION ALL
4  SELECT edge_id AS element_id
5  FROM new_edge_ids
6  UNION ALL
7  SELECT graph_id AS graph_id
8  FROM T1
```

Listing 5.14: SQL-Abfrage aller Element-IDs im horizontalen Schema

5.5 Grouping

In diesem Abschnitt wird die Implementierung des in Kapitel 2.2.4 definierten Grouping-Operators erläutert. Dabei beziehen sich alle nachfolgend gezeigten Abfragen und Beispieltabellen auf das in Kapitel 2.2.4 eingeführte Beispiel: Darin wird der Graph G_1 aus Abbildung 2.11 nach Knoten- und Kantenbezeichnern gruppiert - zur Aggregation wird die `count`-Funktion sowohl auf Knoten als auch auf Kanten herangezogen. Mit Blick auf den Umfang dieser Arbeit wird auf die Darstellung von komplexeren Varianten in den Listings verzichtet. Nachfolgend werden die fünf grundlegenden Schritte zur relationalen Abfrage des zusammenfassenden Graphen erläutert.

1. Vorbereitung der Knotentabelle Unabhängig vom zugrunde liegenden Schema sei nun eine Knoten-Tabelle gegeben, welche die Knoten-ID und alle Eigenschaften (in separaten Spalten) enthält, nach denen die Knoten gruppiert werden sollen (ggf. inklusive Knoten-Bezeichner). Darüber hinaus seien separate Spalten für all diejenigen Eigenschaften enthalten, die von einer oder mehreren Aggregatfunktionen verwendet bzw. aggregiert werden sollen. Auf das soeben referenzierte Beispiel angewandt, wird nun also eine projizierte Knotentabelle vorausgesetzt, die ID und Bezeichner der Knoten enthält (siehe Tabelle 5.2).

vertex_id	vertex_label
V1	City
V2	University
V3	City
V4	Person
V5	Person
V6	Person

Tabelle 5.2: Vorbereitete Knotentabelle (T_1)

2. Gruppierung der Knotentabelle Die Gruppierung der vorbereiteten Knotentabelle kann mittels des relationalen Gruppierungs-Operators (`GROUP BY`) durchgeführt werden. Listing 5.15 zeigt gemäß des Beispiels eine SQL-Abfrage zur Gruppierung nach Bezeichner und Bildung eines `count`-Aggregats. Den neu gewonnenen Superknoten muss eine neue global eindeutige ID zugeordnet werden - dazu wurde eine Skalarfunktion `NEW_GRADOOP_ID` implementiert, welche auf die Referenzimplementierung zurückgreift. In der tatsächlichen Implementierung hängen die zur Gruppierung und Aggregation herangezogenen Spalten augenscheinlich von der Benutzereingabe bzw. den gewählten Parametern des Gradoop-Operators ab.

```
1  T2 AS (
2    SELECT
3      NEW_GRADOOP_ID() AS super_vertex_id,
4      vertex_label AS super_vertex_label,
5      COUNT(vertex_id) AS vertex_count,
6    GROUP BY
7      vertex_label,
8  )
```

Listing 5.15: Gruppierung und Aggregation der Knoten zu Superknoten

Tabelle 5.3 zeigt das Ergebnis der voranstehenden Abfrage - angewandt auf die Beispieldaten aus Tabelle 5.2. Offensichtlich existieren in dem Ausgangsgraph ein `University`-, zwei `City`- und drei `Person`-Knoten - für jeden dieser Bezeichner existiert nun eine Superknoten-Zeile mit jeweiligem `count`-Aggregat.

super_vertex_id	super_vertex_label	vertex_count
SV1	City	2
SV2	University	1
SV3	Person	3

Tabelle 5.3: Superknotentabelle (T_2)

3. Zuordnung von Knoten zu Superknoten Da die Kanten stets nach den zu Start- und Zielknoten zugehörigen Superknoten gruppiert werden müssen (siehe (2.4) in der Definition), wird eine Zuordnung der ursprünglichen Knoten zu ihren Superknoten benötigt. Tabelle T_2 enthält allerdings keine IDs der in einer Gruppe zusammengefassten Knoten. Die gewünschte Zuordnung kann durch einen Gleichverbund der ursprünglichen Knotentabelle T_1 mit T_2 gebildet werden, wobei als Selektionsbedingung eine Gleichheit aller Attribute, die zur Aggregation herangezogen wurden, gefordert ist. Listing 5.16 zeigt Abfragen zur Bestimmung dieser Zuordnung von Knoten zu Superknoten im Kontext des bekannten Beispiels. Die Zwischentabellen T_3 und T_4 in den Zeilen 1 und 2 dienen der Reduzierung der Anzahl an Attributen der Ausgangstabellen - dies ist insbesondere dann sinnvoll, wenn z. B. T_1 viele Eigenschafts-Spalten enthält, die zwar von Aggregatfunktionen verwendet, allerdings nicht zur Gruppierung herangezogen werden. Mit T_5 wird schließlich der Verbund zwischen Knoten und Superknoten gebildet, wobei in diesem Beispiel als Selektionsbedingung die Übereinstimmung der Bezeichner genügt (Zeile 5).

```
1  T3 AS ( SELECT vertex_id, vertex_label FROM T1 ),
2  T4 AS ( SELECT super_vertex_id, super_vertex_label FROM T2 ),
3  T5 AS (
4    SELECT vertex_id, super_vertex_id
5    FROM T3 JOIN T4 ON vertex_label = super_vertex_label
6  )
```

Listing 5.16: Zuordnung von Knoten zu Superknoten

Für den Beispieldatensatz sind T_3 und T_1 identisch - T_4 enthält bis auf das `count`-Aggregat alle Spalten der ursprünglichen Superknotentabelle T_4. Tabelle 5.4 beinhaltet das Ergebnis der soeben erläuterten Verbundoperation von Knoten und Superknoten (das Beispiel fortführend).

vertex_id	super_vertex_id
V1	SV1
V2	SV2
V3	SV1
V4	SV3
V5	SV3
V6	SV3

Tabelle 5.4: Beispiel: Zuordnung von Knoten zu Superknoten ($T_5 = T_3 \bowtie T_4$)

4. Zuordnung von Superknoten zu Start- und Zielknoten der Kanten Wie bereits im vorangegangen Schritt vorbereitet, müssen nun die IDs der Start- und Zielknoten der Kanten durch die jeweiligen IDs der Superknoten ersetzt werden. Dazu sei zunächst die Kantentabelle analog zu der Knotentabelle im ersten Schritt präpariert und mit T_6 bezeichnet - siehe beispielhaft Tabelle 5.5 (a). Anschließend können die Superknoten-IDs über einen zweifachen Verbund mit Tabelle T_5 abgefragt werden. Aufgrund des doppelten Verbundes (für je Start- und Zielknoten) ist es erforderlich und hilfreich, T_5 jeweils auf eine Tabelle mit umbenannten Spaltennamen zu projizieren. Siehe beispielhaft die Abfrage in Listing 5.17. Darin wird T_5 in den Zeilen 8 bis 11 bzw. 14 bis 17 zunächst projiziert, bevor anschließend der Verbund in den Zeilen 12 bzw. 18 gebildet wird. Da die individuellen Kanten-IDs im weiteren Verlauf der Gruppierung i. A. nicht mehr benötigt werden, können diese aus der Projektion ausgeschlossen werden.

```
1  T7 AS (
2    SELECT
3      head_super_vertex_id AS head_id,
4      tail_super_vertex_id AS tail_id,
5      edge_label
6    FROM T6
7    JOIN (
8      SELECT
9        vertex_id AS tail_vertex_id,
10       super_vertex_id AS tail_super_vertex_id
11     FROM T5
12   ) tmp_table_1 ON tail_id = tail_vertex_id
13   JOIN (
14     SELECT
15       vertex_id AS head_vertex_id,
16       super_vertex_id AS head_super_vertex_id
17     FROM T5
18   ) tmp_table_2 ON head_id = head_vertex_id
19 )
```

Listing 5.17: Zuordnung von Superknoten zu Start- und Zielknoten der Kanten

Tabelle 5.5 (b) zeigt das Ergebnis der Abfrage auf der exemplarischen Kantentabelle: Start- und Zielknoten zeigen auf die jeweiligen Super-Knoten.

(a) T_6

edge_id	head_id	tail_id	edge_label
E1	V4	V1	isLocatedIn
E2	V5	V1	isLocatedIn
E4	V4	V5	knows
E5	V4	V6	knows
E8	V5	V6	knows
E11	V6	V3	isLocatedIn
E12	V2	V3	isLocatedIn

(b) $T_7 = T_6 \bowtie T_5 \bowtie T_5$

head_id	tail_id	edge_label
SV3	SV1	isLocatedIn
SV3	SV1	isLocatedIn
SV3	SV3	knows
SV3	SV3	knows
SV3	SV3	knows
SV3	SV1	isLocatedIn
SV2	SV1	isLocatedIn

Tabelle 5.5: Beispiel: Zuordnung von Superknoten zu Start- und Zielknoten der Kanten

5. Gruppierung der Kantentabelle Abschließend muss die aufbereitete Kantentabelle nach Start- und Zielknoten sowie den notwendigen Attributen gruppiert und alle Aggregate gebildet werden. Dazu kann erneut der relationale Gruppierungs-Operator herangezogen werden - neue IDs für die Superkanten können abermals mithilfe der entsprechenden Skalarfunktion gewonnen werden. Listing 5.18 zeigt die beispielhafte Abfrage zur Gruppierung nach Bezeichner und Berechnung des `count`-Aggregats. Tabelle 5.6 beinhaltet das Ergebnis bezogen auf die Beispieldaten.

```
1  T8 AS (
2    SELECT
3      NEW_GRADOOP_ID() AS super_edge_id,
4      tail_id,
5      head_id,
6      edge_label AS super_edge_label,
7      COUNT(edge_id) AS edge_count
8    FROM T7
9    GROUP BY
10     tail_id,
11     head_id,
12     edge_label
13 )
```

Listing 5.18: Gruppierung und Aggregation der Kanten zu Superkanten

super_edge_id	head_id	tail_id	super_edge_label	edge_count
SE1	SV3	SV1	isLocatedIn	3
SE2	SV3	SV3	knows	3
SE3	SV2	SV1	isLocatedIn	1

Tabelle 5.6: Superkantentabelle (T_8)

Die grundlegende Herangehensweise zur Abfrage des zusammenfassenden Graphen wurde nun in den voranstehenden fünf Schritten abgehandelt. Dieses Vorgehen ist i. A. mit allen Schemata kompatibel, sofern jeweils ein Vorbereitungs- und Nachbereitungs-Schritt erfolgt. Bevor auf die einzelnen Schemata eingegangen wird, seien nachfolgend noch einige allgemeine Aspekte besprochen.

Der Grouping-Operator ist auf einem logischen Graphen definiert und nimmt jeweils eine Menge an Gruppierungs-Eigenschaften und Aggregatfunktionen für Knoten (K_v, Λ_v) und Kanten entgegen. Um dem Benutzer eine Gruppierung nach Bezeichner zu ermöglichen, wurde (ähnlich zur formalen Definition) ein spezieller Schlüssel eingeführt, der innerhalb der Implementierung gesondert behandelt wird. Derzeit stellt die Referenzimplementierung vier Aggregatfunktionen zur Verfügung: `count`, `min`, `max` und `sum`. Jede Aggregatfunktion enthält einen nutzerdefinierten Schlüssel, unter welchem das berechnete Aggregat der Superelemente abgelegt wird (z. B. `vertex_count`). Darüber hinaus muss der Nutzer einen weiteren Eigenschaftenschlüssel definieren, wenn die Aggregatfunktion auf bestehenden Eigenschaftswerten arbeitet, i. e. bei `min`, `max` und `sum`. Als Bestandteil der Gradoop-Aggregatsfunktionen wurden im Rahmen dieser Arbeit entsprechend vier neue Flink Table Aggregatsfunktionen (siehe Kapitel 2.4) implementiert.

In den vorangegangen Kapiteln zur Implementierung von EPGM-Operatoren wurde bereits mehrfach auf das Zwischenspeichern von Tabellen eingegangen. Ursächlich für die Verwendung war bislang stets eine Laufzeiteinsparung (siehe Grundlagen dieses Kapitels). Innerhalb der Implementierung des Grouping-Operators erlangt das Zwischenspeichern eine tatsächliche funktionale Notwendigkeit: Da die Superknoten-Tabelle (T_2) mehrfach referenziert wird, würde die Skalarfunktion `NEW_GRADOOP_ID` bei jeder Referenzierung neu ausgewertet - i. e. die IDs der neuen Superknoten wären nicht stabil und das Resultat falsch. Eine Implementierung des Grouping-Operators unter Verwendung der Flink Table API ist folglich ohne Zwischenspeichern nicht möglich.

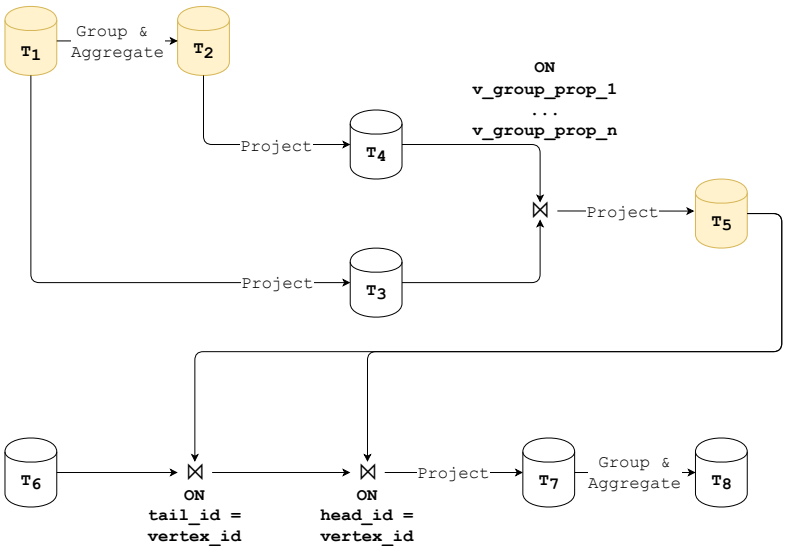


Abbildung 5.10: Datenfluss der generischen Grouping-Implementierung

Abbildung 5.11 zeigt den grundlegenden Datenfluss des Grouping-Operators. Darin wird ersichtlich, dass die Tabellen T_1, T_2 und T_5 zwischengespeichert werden sollten, wobei dies für T_2 wie bereits erläutert obligatorisch ist. Offensichtlich fehlt noch der Bezug zu den Tabellen des Eingabegraphen und Ausgabegraphen, da sich sowohl die Erzeugung der präparierten Knoten- (T_1) und Kantentabelle T_6 als auch die weitere Verarbeitung der Superknoten- (T_2) und Superkantentabelle (T_8) hinsichtlich der Eigenschaftenspalten je nach Schema unterscheiden.

Für den nachfolgenden Abschnitt sei $K_v = \{\tau\}$ ersetzt durch $K_v = \{\texttt{city}\}$. Die Knoten sollen demzufolge nach den zum Schlüssel `city` gehörenden Eigenschaftswerten gruppiert und weiterhin die Anzahl der Knoten in einer Eigenschaft mit dem Schlüssel `count` gespeichert werden. Gemäß Definition des Operators soll der `city`-Eigenschaftswert unter Verwendung desselben Schlüssels an den Superknoten erhalten bleiben.

GVE-Schema Die Vorbereitung der Knoten- bzw. Kantentabelle im GVE-Schema erfolgt über eine einfache Projektion. Um einen einzelnen Eigenschaftswert aus dem `Properties`-Objekt zu erhalten wurde eine Skalarfunktion `EXTRACT_PROPERTY_VALUE` implementiert, die das entsprechende Objekt und einen Eigenschaftenschlüssel entgegen nimmt. Listing 5.19 zeigt exemplarisch eine Abfrage der präparierten Knotentabelle - darin wird gemäß der Beispielkonfiguration der Eigenschaftswert zum Schlüssel `city` extrahiert. Der benutzerdefinierte Eigenschaftenschlüssel `city` steht vor Ausführung fest und kann daher als Literal in die Abfrage eingefügt werden. Analog wird bei der Vorbereitung der Kantentabelle vorgegangen.

```
1  T1 AS (
2    SELECT vertex_id, EXTRACT_PROPERTY_VALUE('city', vertex_properties) AS city
3    FROM vertices
4  )
```

Listing 5.19: Vorbereitung der Knotentabelle im GVE-Schema

Die Verarbeitung der Superknoten- bzw. Superkantentabelle kann im GVE-Schema ebenfalls mittels einer Projektion und dem Aufruf spezieller Skalarfunktionen erfolgen. Listing 5.20 zeigt eine entsprechende SQL-Abfrage - die Erzeugung der trivialen Graphzugehörigkeit (Zeile 5) sollte hinlänglich bekannt sein. Die Funktion `TO_PROPERTIES` wurde implementiert, um auf einzelne Spalten aufgeteilte Eigenschaftswerte in ein `Properties`-Objekt umzuwandeln. Die Eigenschaftenschlüssel stehen wiederum vor Ausführung fest und können als Literal zur Verfügung gestellt werden. Die Funktion nimmt nur eine gerade Anzahl an Parametern entgegen, wobei der i-te Parameter den Eigenschaftenschlüssel zum Wert in Paramter $i + 1$ enthält, falls i gerade ist. Nach dem gleichen Verfahren werden die Superkanten verarbeitet.

```
1  SELECT
2    super_vertex_id AS vertex_id,
3    super_vertex_label AS vertex_label
4    PARSE_GRADOOP_ID_SET('G_NEW') AS vertex_graph_ids,
5    TO_PROPERTIES('city', vertex_city, 'count', vertex_count) AS vertex_properties
6  FROM T2
```

Listing 5.20: Erzeugung der Knotentabelle im GVE-Schema

Die generische Grouping-Implementierung kann also ohne Einlesen bzw. Schreiben weiterer Tabellen allein durch Projektion und Skalarfunktionen im GVE-Schema verwendet werden.

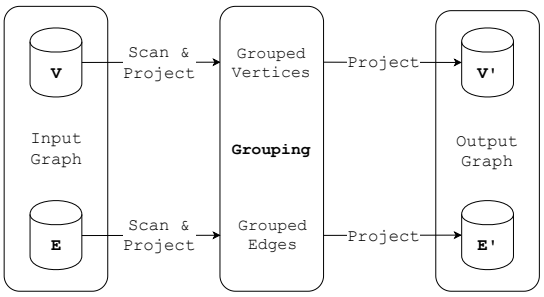


Abbildung 5.11: Verwendung der Grouping-Implementierung im GVE-Schema

Vertikales Schema Da die Eigenschaften im vertikalen Schema in separaten Tabellen ausgelagert sind, genügt es nicht mehr, einzig die Knoten- bzw. Kantentabelle einzulesen. Eine Spalte für eine bestimmte Eigenschaft kann durch einen linken äußeren Verbund (`LEFT OUTER JOIN`) mit der Knoten-Eigenschaften-Tabelle gewonnen werden, wobei diese zunächst auf alle Eigenschaftswerte des jeweiligen Eigenschaftenschlüssels eingeschränkt werden sollte. Für n verschiedene Eigenschaften müssen n Verbünde gebildet werden. Listing 5.21 zeigt, wie die Knoten inklusive einer separaten Spalte für die Eigenschaft `city` abgefragt werden können. Der äußere Verbund wird benötigt, da ein Null-Wert genauso wie andere Werte behandelt werden soll, i. e. jeder Knoten muss durch genau einen Superknoten abgedeckt sein. Insbesondere bei mehreren Gruppierungs-Eigenschaften erschließt sich die Verwendung des äußeren Verbundes. Die Vorbereitung der Kanten erfolgt analog.

```
1  T1 AS (
2    SELECT vertex_id, city
3    FROM vertices LEFT OUTER JOIN (
4      SELECT vertex_property_value AS city, property_vertex_id
5      FROM vertex_property_values
6      WHERE vertex_property_key = 'city'
7    ) tmp ON vertex_id = property_vertex_id
8  )
```

Listing 5.21: Erzeugung der Knotentabelle im vertikalen Schema

Die Graphzugehörigkeit zu einem neuen logischen Graphen kann im vertikalen Schema auf die bekannte triviale Art erzeugt werden. Ebenso trivial erfolgt die Abfrage der neuen Knoten- bzw. Kantentabelle im vertikalen Schema. Dazu muss z. B. die Superknotentabelle nur auf ID und Bezeichner mit entsprechender Umbenennung der Spalten projiziert werden. Die Abfrage der Knoten- bzw. Kanten-Eigenschaften-Tabellen der Ergebnis-Instanz muss für n Eigenschaften über die Vereinigung von n Tripel-Tabellen erfolgen. Siehe dazu beispielhaft Listing 5.22: Die `city`-Tripel werden mit den `count`-Tripeln vereinigt. Offensichtlich wird nicht nur die Superknotentabelle (T_2) mehrfach referenziert, sondern analog auch die Superkantentabelle (T_8) - daher ist es obligatorisch, diese im vertikalen Schema zwischenzuspeichern.

```
1  SELECT
2    super_vertex_id AS property_vertex_id
3    'city' AS vertex_property_name
4    vertex_city AS vertex_property_value
5  FROM T2
6  UNION
7  SELECT
8    super_vertex_id AS property_vertex_id
9    'count' AS vertex_property_name
10   vertex_count AS vertex_property_value
11 FROM T2
```

Listing 5.22: Abfrage der Superknoten-Eigenschaften im vertikalen Schema

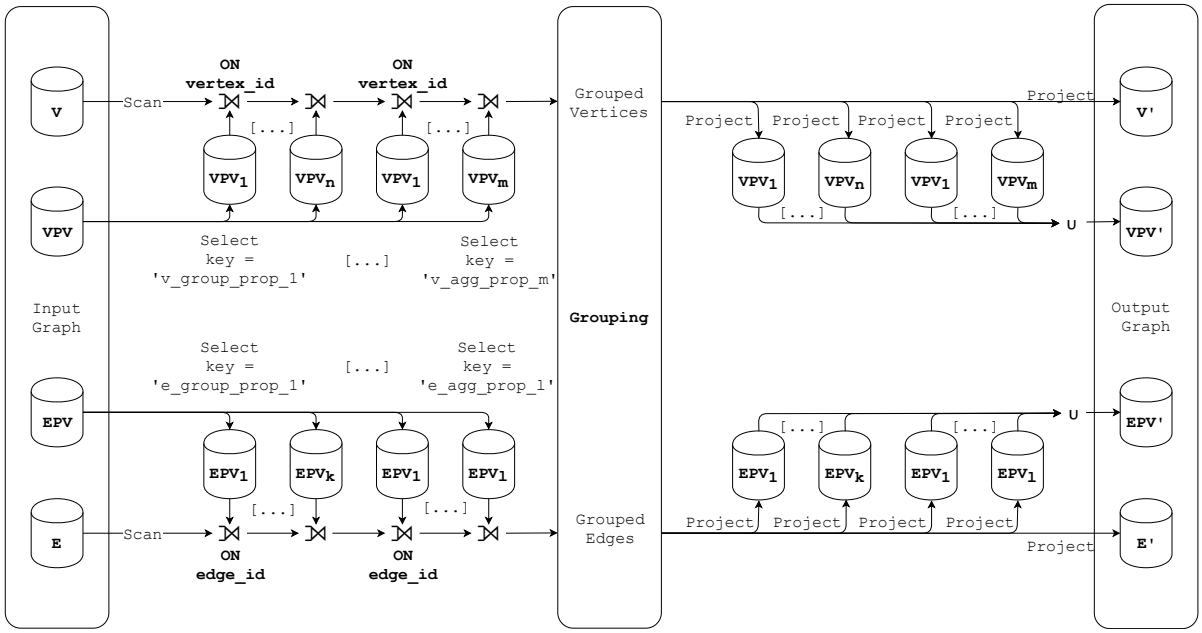


Abbildung 5.12: Verwendung der Grouping-Implementierung im vertikalen Schema

Abbildung 5.12 zeigt den vollständigen Datenfluss im vertikalen Schema, wobei davon ausgegangen wird, dass n bzw. k Eigenschaften zur Gruppierung der Konten bzw. Kanten herangezogen werden und m bzw. l Eigenschaften der Knoten bzw. Kanten aggregiert werden.

Horizontales Schema Im horizontalen Schema können die Eigenschaftswerte ebenfalls über einen linken äußeren Verbund mit der jeweiligen Eigenschaftentabelle (z. B. der Tabelle `city`) in die präparierte Knotentabelle integriert werden. Im Vergleich zu den Tripeln im vertikalen Schema muss allerdings keine Selektion stattfinden, sondern eine Eigenschaftentabelle kann direkt verwendet werden, ohne zu prüfen, ob der Eigenschaftenschlüssel übereinstimmt. Die resultierenden Eigenschaftentabellen können abermals über eine Projektion der Supertabellen abgefragt werden und für jede Eigenschaft direkt in eine separate Tabelle geschrieben werden - eine Vereinigung ist nicht mehr erforderlich. Aufgrund der mehrfachen Referenzierung muss die Superkantentabelle analog zum vertikalen Schema zwingend zwischengespeichert werden.

6 Evaluation

Im folgenden Kapitel wird die EPGM-Implementierung unter Verwendung der Apache Flink Table API evaluiert.

6.1 Allgemeine Beurteilung

Ein wesentlicher Vorteil der Verwendung der Table-API von Apache Flink ist, dass fast alle Aspekte der Verteilung bei der Formulierung von relationalen Abfragen verborgen sind. Der Nutzer kann SQL-Abfragen formulieren und von der verteilten Ausführung profitieren, ohne die Konzepte hinter Apache Flink oder die DataSet-Operatoren kennen zu müssen. Da SQL unter Softwareentwicklern eine überaus verbreitete und bekannte Sprache ist, werden somit eine breite Menge an Entwickler angesprochen und die Hemmschwellen gesenkt. Im Kontext von Gradoop ist vorgesehen, dass die Anwender selber Operatoren implementieren können. Dafür müssen sie unter Verwendung der Table-API das zugrundeliegende Schema kennen und verstehen. Da es keine POJO-Typen und die damit einhergehenden softwaretechnischen Annehmlichkeiten gibt, ist es möglicherweise als weniger komfortabel anzusehen, eigene Operatoren unter Verwendung der Table-API zu implementieren.

Am Beispiel des Subgraph-Operators wurde deutlich, dass die relationale Abstraktion teilweise auch zu Vereinfachung führt. Dies ist hauptsächlich auf die geringe Ausdrucksstärke im Vergleich zur DataSet-API zurückzuführen. Da es deutlich weniger Möglichkeiten gibt, vom Endanwender der Gradoop-Schnittstellen nutzerdefinierte Funktionen entgegenzunehmen, verliert die Gradoop-API möglicherweise an Flexibilität. Während der Implementierung der EPGM-Operatoren ist die unpraktische Handhabe von SQL-Zeichenketten aufgefallen. Dies ist jedoch kein spezifisches Problem von Apache Flink, sondern eine generelle softwaretechnische Herausforderung. Im Kapitel zur Implementierung wurde bereits angerissen, wie dieser Herausforderung mittels spezieller Hilfsprozeduren begegnet werden konnte.

Da für die Wiederverwendung von Zwischenergebnissen kurzzeitig auf die DataSet-API gewechselt werden muss (vgl. Grundlagen der Implementierung), disqualifiziert sich die Table-API praktisch als alleinstehende Lösung für komplexere Workflows, da zu viele Abfragen vielfach ausgewertet würden. Am Beispiel des Grouping-Operators konnte gezeigt werden, dass es gar keine Implementierung dieses Operators geben dann, die einzig die Table-API verwendet.

6.2 Evaluationsumgebung und -datensätze

Die experimentelle Evaluation der relational abstrahierten Implementierung des EPGMs bzw. der verschiedenen Operatoren erfolgt anhand von Ausführungen auf einem horizontal skalierbaren Big Data Cluster der Universität Leipzig. Die Ausführung der Workflows erfolgt auf maximal 16 Knoten, wobei jeder folgende Spezifikationen erfüllt:

- Intel Xeon E5-2430 v2 (6 Kerne zu je 2,5GHz)
- 48 GByte Hauptspeicher, davon 32 GByte zugeteilt
- 1 GBit/s Ethernet

Zusätzlich stehen zwei Master-Knoten für die Verwaltung der Jobs zur Verfügung. Alle Systeme sind auf Basis eines openSUSE Betriebssystems konfiguriert. Im Rahmen dieser Arbeit wurde Apache Flink in der Version 1.6.0 verwendet. Das verteilte Dateisystem beruht auf Hadoop in der Version 2.6.

Die Evaluation wurde ausgehend von einem synthetisch generierten Datensatz durchgeführt, der bereits in einer Vielzahl weiterer Veröffentlichungen der Abteilung Datenbanken herangezogen wurde ([32] [31] [29]). Zur Erstellung dieses Datensatzes wurde ein Datengenerator des *Linked Data Benchmark Council* (LDBC) verwendet, der ein synthetisches soziales Netzwerk liefert, welches hinsichtlich üblicher Graph-Metriken repräsentativ für ein soziales Netzwerk ist. Der Generator wurde als Teil des *Social Network Benchmark* (SNB) veröffentlicht. [21] Der Graph besteht unter anderem aus *Personen*, die sich ggf. gegenseitig *kennen* und sich in bestimmten *Foren* austauschen - dies erfolgt über *Posts* und *Kommentare*. Der LDBC-Datensatz liegt in verschiedenen Skalierungen (1, 10 und 100) in einem mit Gradoop kompatiblen CSV-Format vor. Die drei Datensätze, die jeweils einen logischen Graphen enthalten, wurden mit den im Rahmen dieser Arbeit implementierten Konvertierungsfunktionen in jedes der drei relationalen Schemata überführt und über die entsprechenden Datensenken erneut in das verteilte Dateisystem geschrieben. Tabelle 6.1 listet einige Metriken der drei Datensätze auf - hinsichtlich der Kantenanzahl sind sie offensichtlich annähernd proportional zu 1, 10 und 100 skaliert. Von Interesse ist zudem der benötigte Speicherplatz im HDFS: Zwischen den verschiedenen Schemata ergeben sich teilweise deutliche Unterschiede. Die auf das Schema zurückzuführenden Unterschiede wurden bereits im konzeptionellen Entwurf (Kapitel 4) vorhergesagt. Dass dieselbe EPGM-Instanz im GVE-Schema deutlich größer als im Gradoop-Format ist, lässt sich allerdings nicht durch Schema-Unterschiede erklären. Vielmehr wurde in der Referenzimplementierung auf eine weniger speicherplatzintensive Kodierung des Eigenschaften-Objekts gesetzt. Bei der weiteren Interpretation der Evaluationsergebnisse ist folglich zu beachten, dass im GVE-Schema eine größere Rohdatenmenge verarbeitet wird, als in Gradoop.

Um die konzeptionellen Unterschiede der Schemata genauer zu untersuchen, sei darüber hinaus ein detaillierterer Vergleich der Datensatzgrößen am Beispiel des LDBC100 in Abbildung 6.1 dargestellt. Von Interesse ist dabei i. A. die Aufteilung des Speicherbedarfs im horizontalen und vertikalen Schema: Im Vergleich zum GVE-Schema konnten die Kanten- und Knotentabelle gemeinsam um mehr als die Hälfte reduziert werden. Insbesondere die Knoten, die in

Datensatz	Anzahl		Datengröße			
	Knoten	Kanten	Gradoop	GVE-Schema	Vertikales Schema	Horizontales Schema
LDBC1	3,2 M	17,3 M	3,1 GB	3,5 GB	4,9 GB	4,7 GB
LDBC10	30,0 M	176,6 M	25,7 GB	35,6 GB	49,7 GB	47,6 GB
LDBC100	282,6 M	1.775,5 M	253,4 GB	356,3 GB	495,9 GB	475,0 GB

Tabelle 6.1: Datensätze und deren Größen

diesem Datensatz vergleichsweise viele Eigenschaften zugeordnet haben, benötigen deutlich weniger Speicherplatz (weniger als 10%). Dies ist z.B. von Vorteil, wenn nur die Knoten-Bezeichner eingelesen bzw. verarbeitet werden sollen. Darüber hinaus beansprucht die Graphzugehörigkeit in diesem Beispieldatensatz einen großen Teil der Gesamtgröße - es sei darauf hingewiesen, dass die Instanz einen einzigen logischen Graphen und somit nur die triviale Graphzugehörigkeit enthält. Es lässt sich folgern, dass die Bezeichner von Knoten und Kanten durchschnittlich weniger Speicherplatz als eine `GradoopId` benötigen. Gemäß den in Kapitel 4 angestellten Überlegungen, beanspruchen die Eigenschaften-Tripel im vertikalen Schema mehr Platz als die Dupel im horizontalen.

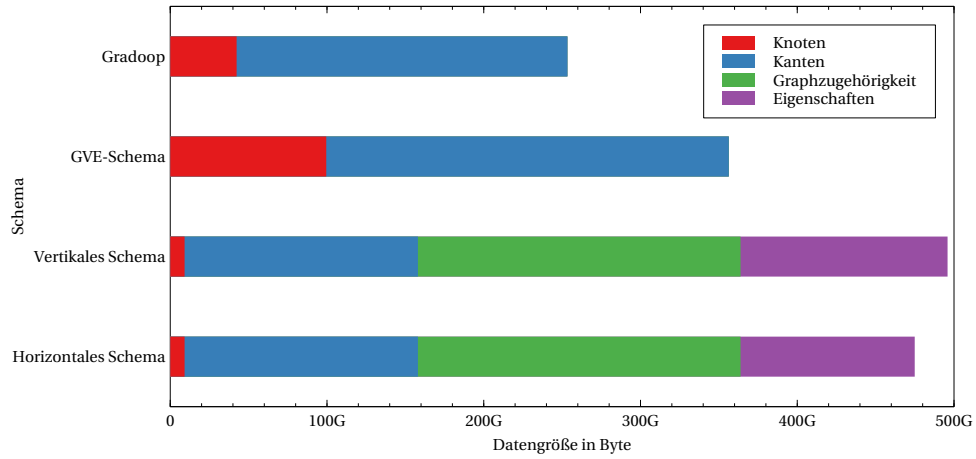


Abbildung 6.1: Vergleich der Datensatzgröße des LDBC100

Die zur Evaluation herangezogenen Datensätze sind stets auf alle 16 Datenknoten verteilt - unabhängig davon, mit welcher Parallelität p das jeweilige Programm ausgeführt wird. Die Daten müssen also im Falle von geringerer Parallelität zu Beginn jeder Ausführung über das Netzwerk auf die Ausführungsknoten verteilt werden. Diese Datenumverteilung übernimmt Apache Flink zwar automatisch, dennoch beeinflusst sie die Vergleichbarkeit der Ausführungen von unterschiedlicher Parallelität. Darüber hinaus wird p bei der Ausführung eines Workflows nicht in der Anzahl an Ausführungsknoten, sondern an Taskslots definiert. Dabei ist nicht deterministisch, auf welche und wie viele Ausführungsknoten eine Ausführung mit $p = 6$ verteilt wird - obwohl ein einziger Ausführungsknoten (mit 6 CPU-Kernen) gewünscht ist. Im weiteren Verlauf der Evaluation wird analog zu [32] [31] [29] $p = 6$ mit 1 Knoten gleichgesetzt. Bei der Interpretation der nachfolgenden Angaben zur Skalierbarkeit sollten die vorangehend erläuterten Aspekte jedoch stets berücksichtigt werden. Die Einflüsse der Mischung von vertikaler und horizontaler Skalierung werden im Verlauf der Arbeit nicht weiter thematisiert.

6.3 Experimentelle Evaluation von Laufzeit und Skalierbarkeit

6.3.1 Allgemeine Ergebnisse

Alle im Rahmen dieser Arbeit implementierten Operatoren wurden hinsichtlich Laufzeit und Skalierbarkeit evaluiert. Dazu wurde jeder Operator in jedem Schema (und ggf. zusätzlich die Gradoop-Implementierung) auf 1, 2, 4, 8 und 16 Knoten ausgeführt. Jede Ausführung wurde dreimal wiederholt, um Unterschiede hinsichtlich des HDFS Caching-Verhaltens und der Mischung von vertikaler und horizontaler Skalierung annähernd auszugleichen. Die drei gemessenen Laufzeiten wurden dazu arithmetisch gemittelt. Von Interesse sind bei der Interpretation dieser Werte zwei Vergleiche: Die Referenzimplementierung im Vergleich zur relational abstrahierten Implementierung im GVE-Schema und der Vergleich der verschiedenen relationalen Implementierungen untereinander. Bevor die Ergebnisse der einzelnen Operatoren dargestellt und analysiert werden, seien nachfolgend allgemeine Beobachtungen angestellt.

Zunächst lässt sich beobachten, dass die Implementierung aller Operatoren im GVE-Schema bei maximaler Parallelität stets schneller als die Referenzimplementierung ist. Die Ursachen sind vielfältig - nachfolgend werden die wichtigsten Gründe erläutert:

1. **Optimierung des Operatorbaums** Aufgrund der zumeist explizit deklarierten Abfragen, kann der Operatorbaum offensichtlich besser optimiert werden. Dabei werden einzelne Operatoren effizienter verkettet - insbesondere mit Blick auf den Hauptspeicherverbrauch. Zu beachten ist dabei, dass die Implementierung im GVE-Schema mehr Rohdaten zu verarbeiten hat - allerdings ein besseres Speicherverhalten aufweist. Zudem kann der `Row`-Datentyp besser serialisiert und optimiert werden. Nutzerdefinierte Funktionen, deren enthaltene Logik vom Optimierer schlecht berücksichtigt werden kann, werden deutlich weniger verwendet.

2. **Optimierung der CSV-Datenquelle** Die im Rahmen der Arbeit herangezogene Datenquelle der Table-API ist dahingehend optimiert, nur Daten in den Speicher zu laden, die auch tatsächlich im weiteren Verlauf des Workflows verwendet werden. Falls möglich wird nicht nur die Projektion, sondern auch die Selektion bereits in der Datenquelle ausgeführt.

3. **Komplexere Datenquelle und -senke in Gradoop** Im Vergleich zu der naiven Variante in den Implementierungen unter Verwendung der Table-API wird in Gradoop eine komplexere Datenquelle und -senke verwendet: Dabei werden zusätzlich Metadaten über die existierenden Eigenschaften inklusive der entsprechenden Datentypen gelesen bzw. geschrieben.

Weiterhin lässt sich beobachten, dass die Implementierungen im horizontalen Schema ein deutlich höheres Netzwerkaufkommen innerhalb des Clusters erzeugen. In der Standard-Konfiguration ließen sich diese oftmals nicht ohne Erhöhung das Netzwerkpuffers auf bis zu 6 GiB erfolgreich ausführen. Entsprechend weniger Hauptspeicher steht den Ausführungsknoten folglich für lokale Daten bzw. Operationen zur Verfügung. Generell lassen sich unter den relational abstrahierten Implementierungen deutliche Unterschiede hinsichtlich des E/A-Verhaltens feststellen.

6.3.2 Subgraph

Zur Evaluation des Subgraph-Operators wurden folgende Prädikatsfunktionen gewählt:

- $\Phi_v(v) := \begin{cases} \texttt{true} & \sigma(v) = \texttt{person} \\ \texttt{false} & \text{sonst} \end{cases}$
- $\Phi_e(e) := \begin{cases} \texttt{true} & \sigma(e) = \texttt{knows} \\ \texttt{false} & \text{sonst} \end{cases}$

Anschaulich werden aus dem sozialen Netzwerk alle Personen-Knoten und deren Freundschaften untereinander selektiert. Da eine `knows`-Kante stets als Start- und Zielknoten eine Person hat und alle `person`-Knoten selektiert sein sollen, kann auf eine Konsistenzprüfung verzichtet werden. Ein entsprechender Parameter (`unverified`) kann sowohl in Gradoop also auch in der in Kapitel 5 erläuterten Implementierung gesetzt werden. Auf einen Verbund von Knoten und Kanten wird in dieser Version verzichtet. Gleichzeitig bietet der Operator in dieser Konfiguration eine gute Basis zur Beurteilung des E/A-Verhaltens. Der Vollständigkeit halber ist in Tabelle 6.2 die Selektivität der Selektion auf den drei Datensätzen aufgeführt. Zwar skaliert die Selektivität nicht exakt mit den Datensätzen, bewegt sich jedoch insgesamt recht stabil im Bereich um 1 Prozent.

Quelldatensatz	Gesamtselektivität (Knoten und Kanten)	Knotenselektivität	Kantenselektivität
LDBC1	0,93 %	0,31 %	1,05 %
LDBC10	0,97 %	0,22 %	1,10 %
LDBC100	0,99 %	0,16 %	1,12 %

Tabelle 6.2: Selektivität der skalierten LDBC Datensätze unter der gewählten Konfiguration

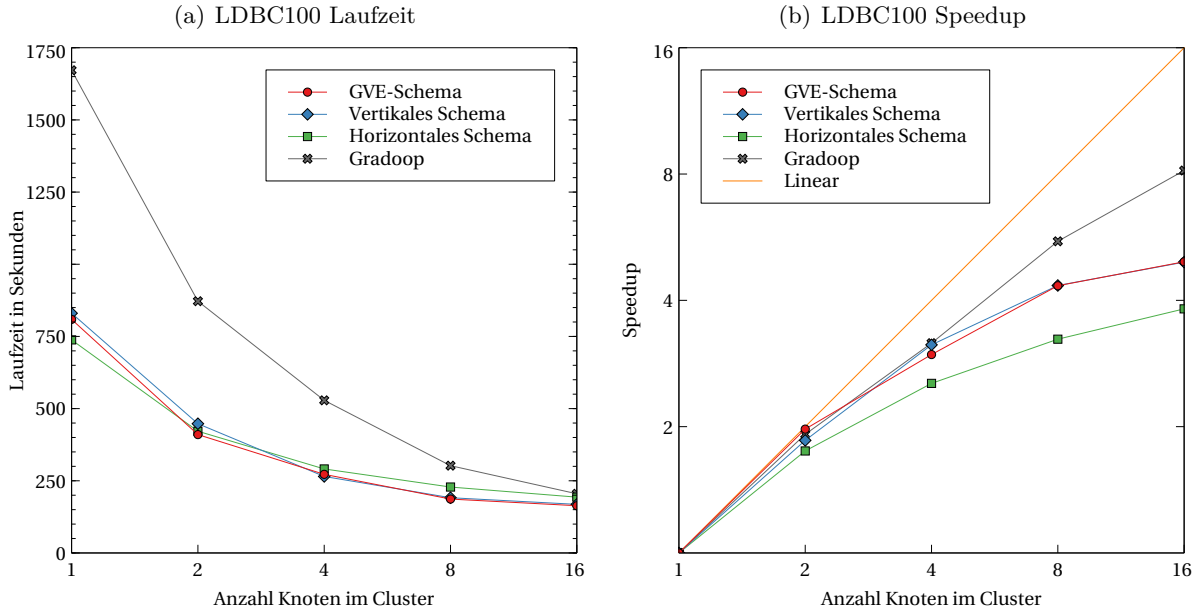


Abbildung 6.2: Laufzeit und Speedup des Subgraph-Operators auf LDBC100

Abbildung 6.2(a) zeigt die Laufzeiten der Subgraph-Ausführung auf dem LDBC100 in Abhängigkeit der Anzahl der Knoten im Cluster. Es zeigen sich die im vergangenen Abschnitt erläuterten

Laufzeitunterschiede zwischen Referenzimplementierung und den übrigen Implementierungen. Besonders groß sind die Unterschiede bei niedrigerer Parallelität - dabei kann offensichtlich auch von der effizienteren Speichernutzung der Datenquelle profitiert werden. Betrachtet man die Laufzeiten im horizontalen Schema, kann die große Menge von Tabellen augenscheinlich schneller eingelesen und geschrieben werden - das E/A-Verhalten ist aufgrund besserer Ausnutzung von Parallelität besser. Bei hoher Parallelität scheint hingegen der Verbundaufwand (für jede Eigenschaftentabelle) beim Erzeugen der induzierten EPGM-Instanz zu viel Zusatzaufwand und Netzwerknutzung zu erzeugen. Im theoretisch ungünstigsten Fall müssen für jede Eigenschaftentabelle zwischen allen Knoten Daten ausgetauscht werden. Betrachtet man den Speedup (Abbildung 6.2(b)) überzeugt die Referenzimplementierung, während die Implementierung im horizontalen Schema aus genannten Gründen schlechtere Werte liefert.

Der Zusatzaufwand, den die Verbund-Berechnungen im horizontalen Schema erzeugen, wird besonders an den Ausführungszeiten auf dem kleineren LDBC10-Datensatz deutlich (Abbildung 6.3): Die Ausführung auf 16 ist langsamer als diejenige auf zwei Knoten. Ebenso erscheint das vertikale Schema hinsichtlich einer einfachen Subgraph-Operation auf einer solch geringen Datenmenge nicht praktikabel. Da die verteilte Graphanalyse allerdings stets im Kontext von Big Data zu betrachten ist, sind solche negativen Effekte auf kleinen Datenmengen bei der Bewertung zu vernachlässigen.

(a) LDBC10 Laufzeit

(b) LDBC10 Speedup

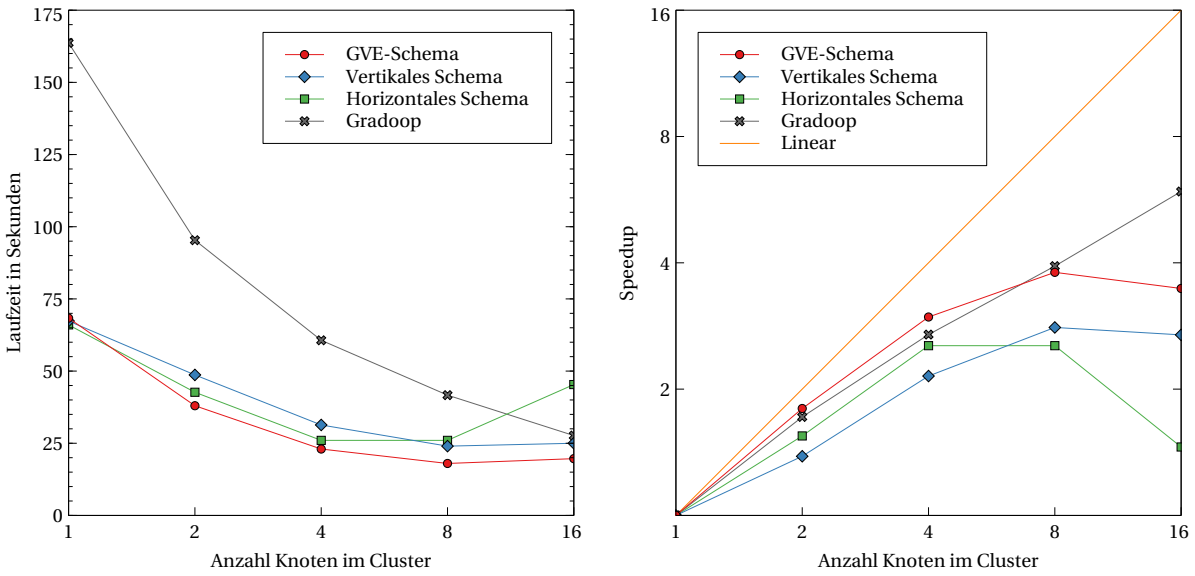


Abbildung 6.3: Laufzeit und Speedup des Subgraph-Operators auf LDBC10

6.3.3 Mengenoperatoren auf logischen Graphen

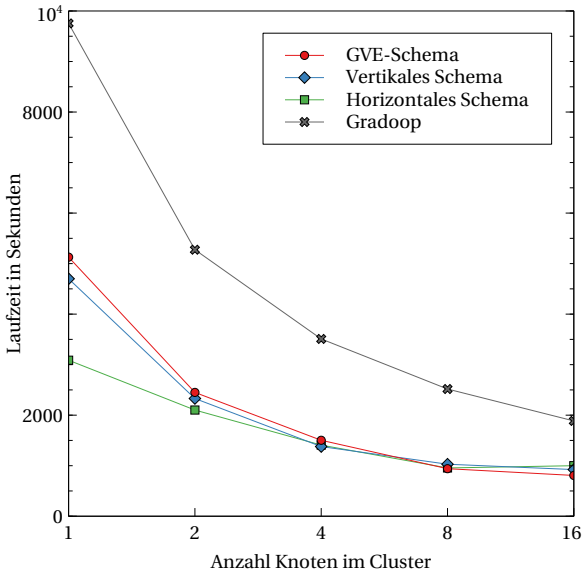
Die Operatoren Combination, Exclusion und Difference sind jeweils auf zwei logischen Graphen definiert. Daher mussten aus den Datensätzen zunächst zwei Teilgraphen G_1 und G_2 extrahiert werden. Zu diesem Zwecke wurde erneut der Subgraph-Operator herangezogen, um die durch die Knoten mit den Bezeichnern `person`, `forum`, `post`, `university` und `city` bzw. `person`, `forum`, `comment`, `tag` und `tagclass` induzierten Teilgraphen (G_1 bzw. G_2) zu erzeugen. Durch diese Wahl an Bezeichnern ist sichergestellt, dass sich die beiden logischen Graphen sowohl in Knoten als auch in Kanten überlappen. Da die Überlappung nicht vollständig ist, sind alle Ergebnisse der Operatoren nicht-leer. Tabelle 6.3 führt die Metriken der beiden Graphen und des Overlap-Graphen auf.

Quelldatensatz	G_1		G_2		$G_1 \cap G_2$	
	Knoten	Kanten	Knoten	Kanten	Knoten	Kanten
LDBC1	1,1 M	4,7 M	2,2 M	9,7 M	0,1 M	1,9 M
LDBC10	8,1 M	43,5 M	22,5 M	105,0 M	0,7 M	19,7 M
LDBC100	62,5 M	420,2 M	224,6 M	1.094,4 M	4,5 M	203,9 M

Tabelle 6.3: Datensatzmetriken der Evaluation der Mengenoperatoren auf logischen Graphen

Combination Abbildung 6.4 zeigt die Evaluationsergebnisse der Combination-Ausführungen auf den beiden aus LDBC100 gewonnenen logischen Graphen. Offensichtlich finden sich die aus den vorangegangen Abschnitten bekannten Beobachtungen wieder. In diesem Fall skaliert die Implementierung im GVE-Schema sogar besser als die Referenzimplementierung. Scheinbar kann die Bildung einer duplikatfreien Vereinigung relational betrachtet besser optimiert werden. Es sei angemerkt, dass in der Gradoop-Implementierung eine benutzerdefinierte Funktion verwendet werden muss, um das Identifikations-Merkmal zu bestimmen. In der relationalen Betrachtung ist die Vereinigung (`UNION`) bereits duplikatfrei definiert.

(a) LDBC100 Laufzeit



(b) LDBC100 Speedup

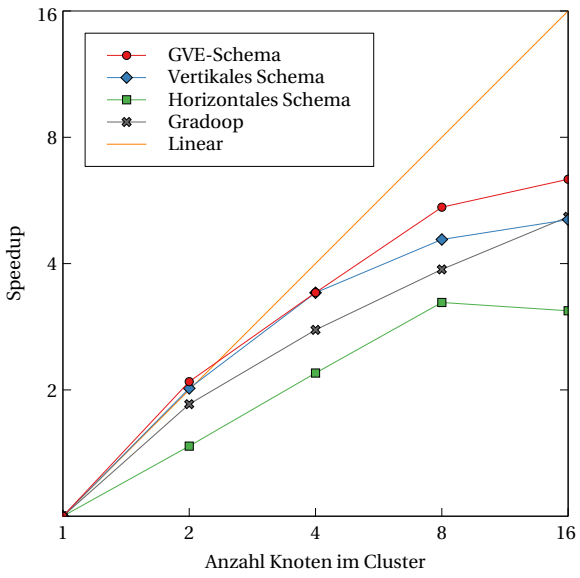


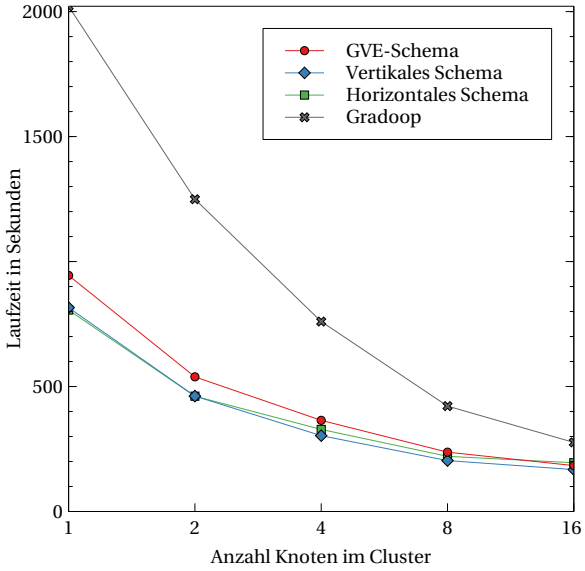
Abbildung 6.4: Laufzeit und Speedup des Combination-Operators

Der Vorteil hinsichtlich des E/A-Verhaltens bei niedrigerer Parallelität im horizontalen Schema erscheint erneut sehr deutlich - dies ist insbesondere auch darauf zurückzuführen, dass die Eigenschaften-Dupel des zweiten Graphen erst nach Bestimmung der Vereinigung eingelesen werden müssen - im Vergleich zum GVE-Schema, in dem alle Eigenschaften mit eingelesen werden müssen, lässt sich Netzwerkkommunikation sparen. Bei höherer Parallelität fällt hingegen der schon diskutierte Mehraufwand im horizontalen Schema mehr ins Gewicht.

Exclusion und Overlap Am Beispiel des Exclusion-Operators (der sich hinsichtlich der qualitativen Ergebnisse vernachlässigbar gering vom Overlap-Operator unterscheidet) wird ersichtlich, dass das vertikale Schema möglicherweise eine geeignete Abwägung zwischen besserem E/A-Verhalten und schlechterem Netzwerkverhalten aufgrund von Verbundoperationen darstellt. In Abbildung 6.5(a) wird ersichtlich, dass die Laufzeit der Implementierung im vertikalen Schema bei maximaler Parallelität sogar schneller als diejenige im GVE-Schema ist. Es wird erneut viel Aufwand für das Einlesen von Eigenschaften gespart, da im Falle des Exclusion-Operators ohnehin keines der Elemente bzw. Eigenschaften aus dem zweiten Eingabegraphen im Ergebnis enthalten sein kann. Zu beachten ist, dass die Rohdatenmenge im vertikalen und horizontalen Schema theoretisch deutlich größer als im GVE-Schema ist - die Graphzugehörigkeit und ein Großteil der Eigenschaften müssen allerdings gar nicht eingelesen werden.

Die Mengenoperationen (Differenz- und Schnittmenge) wurden in der Referenzimplementierung durch einen Verbund gelöst - die abstrakten relationalen Operatoren hingegen werden von Apache Flink jeweils mit einem `CoGroup`-Operator umgesetzt. Unter anderem damit lässt sich der Laufzeitunterschied zwischen Gradoop und den übrigen Implementierungen erklären. Abbildung 6.5 bestätigt, dass die Laufzeiten gut mit der Größe der Datensätze skalieren. Gleichzeitig wird der hohe Zusatzaufwand bei kleinen Datensätzen im horizontalen Schema ersichtlich: Die Laufzeiten auf dem LDBC1 und LDBC10 unterscheiden sich nur unwesentlich.

(a) LDBC100 Laufzeit



(b) LDBC100 Scaleup auf 16 Knoten

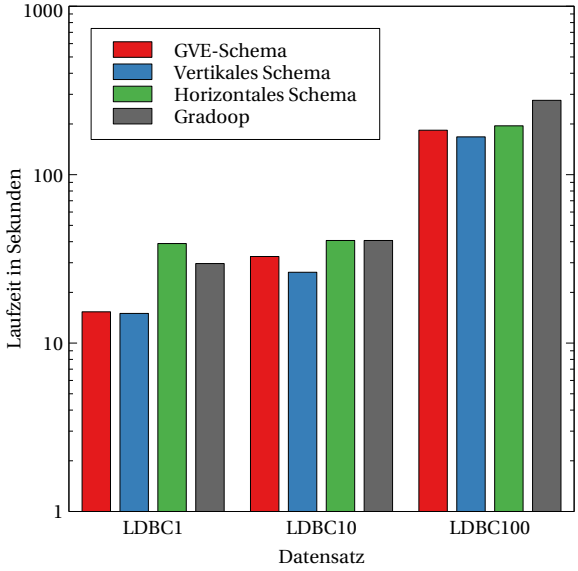


Abbildung 6.5: Laufzeit und Speedup des Exclusion-Operators

6.3.4 Mengenoperatoren auf Graph Collections

Zur Evaluation der Mengenoperatoren auf Graph Collections mussten zunächst zwei EPGM-Instanzen $\mathcal{G}_1$ und $\mathcal{G}_2$ aus dem LDBC-Datensatz erzeugt werden, die als Eingaben der jeweiligen Operatoren dienen können. Idealerweise enthält jede der beiden EPGM-Instanzen eine hinreichend große Menge an logischen Graphen, welche sich wiederum in einem Teil überlappen. Eine derartige Präparation eines LDBC-Datensatzes wurde bislang nicht durchgeführt. Im Rahmen dieser Arbeit wurde wie folgt vorgegangen:

1. Zunächst wurde der logische Graph G, der in einem LDBC-Datensatz enthalten ist, mittels des Subgraph-Operators auf die Nachrichten- und Personen-Knoten (`person`, `post` und `comment`) und die `hasCreator`-Kanten eingeschränkt.

2. Der Teilgraph von G wurde anschließend unter Verwendung des in Gradoop implementierten Operators zur Berechnung der Zusammenhangskomponenten in eine Graph Collection $\mathcal{G}$ überführt. Es sei erwähnt, dass durch die Verwendung von Zusammenhangskomponenten ein Knoten bzw. eine Kante niemals in mehr als einem logischen Graphen enthalten sein kann. Die konzeptionelle `n:m`-Beziehung findet sich so in diesem Evaluationsdatensatz nicht wieder.

3. Abschließend wurden zwei Teilmengen selektiert: Zunächst alle logischen Graphen, deren ältester vorhandener Nachricht-Knoten vor 2012 erzeugt wurde ($\mathcal{G}_1$) und analog alle logischen Graphen mit Knoten nach 2010 ($\mathcal{G}_2$). So wurde die Überlappung der beiden Graph-Mengen sichergestellt.

Datensatz	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 6.4: Datensatzmetriken der Evaluation der Mengenoperatoren auf Graph Collections

Mit Blick auf den Umfang der Arbeit und die zeitlich begrenzte Verfügbarkeit des Universitäts-Clusters wurden die Operatoren nur auf den aus dem LDBC10-Datensatz gewonnenen Graph Collections evaluiert. Tabelle 6.4 zeigt die grundlegenden Metriken der beiden gewonnenen Graph Collections und der Intersection. Dabei fällt auf, dass in einem logischen Graphen durchschnittlich mehrere Hundert Knoten und Kanten enthalten sind. Jeder der logischen Graphen enthält eine Person und ihre verfassten Nachrichten. Zwischen n solcher Knoten müssen folglich $n - 1$ `hasCreator`-Kanten existieren. Da die Anzahl an logischen Graphen deutlich kleiner als die Anzahl der Knoten und Kanten ist, erklärt sich, dass nur minimal mehr Knoten als Kanten existieren. Offensichtlich fällt die Anzahl an logischen Graphen noch nicht in Big Data Größenordnungen.

Union Die Implementierung des Union-Operators unterscheidet sich konzeptionell nicht von der des Combination-Operators - erwartungsgemäß ergaben sich keine von Combination verschiedenen Evaluationsergebnisse. Einzig die zu verarbeitende Datenmenge war deutlich geringer - dies hat sich im Vergleich negativ auf die Skalierbarkeit ausgewirkt.

Difference und Intersection Da sich der Difference-Operator wiederum vom Intersection-Operator nur hinsichtlich einer Mengenoperation unterscheidet, seien nachfolgend stellvertretend die Evaluationsergebnisse des Difference-Operators besprochen. Abbildung 6.6 zeigt Laufzeit und Speedup der Ausführung auf dem Cluster. Augenscheinlich sind die absoluten Laufzeiten schon bei geringster Parallelität sehr niedrig und eine Skalierung teilweise nicht messbar. Dies ist wie bereits in der vorangegangen Evaluation beobachtet auf die kleine Datenmenge zurückzuführen. Diese Evaluationsergebnisse genügen folglich i. A. nicht den Anforderungen im Big Data Umfeld, decken sich allerdings weitestgehend mit den vorangehend angestellten Beobachtungen.

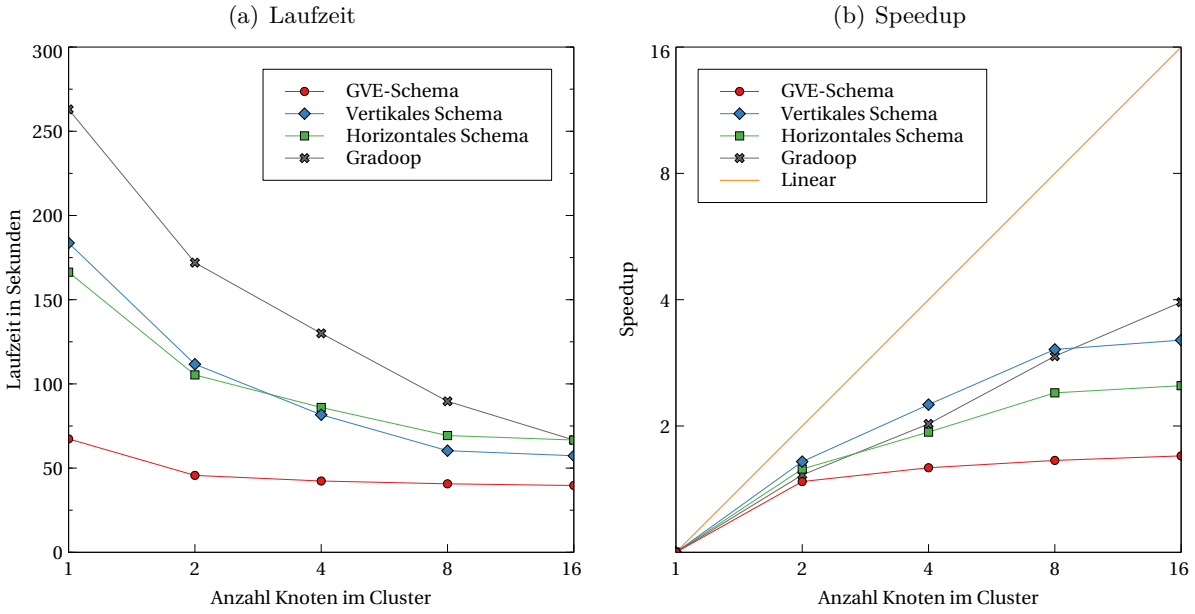


Abbildung 6.6: Laufzeit und Speedup des Difference-Operators

Nichtsdestoweniger sei der Vollständigkeit halber auf einige Aspekte detaillierter eingegangen. Wie im Kapitel über die Implementierung der Operatoren veranschaulicht wurde, liegen diesen analog zu den Mengenoperatoren auf logischen Graphen die Differenz- und Schnittmenge zugrunde. Eine genaue Analyse hat ergeben, dass die eigentliche Mengenoperation bei der Ausführung des Difference- und Intersection-Opterators im Vergleich zum E/A-Aufwand und der Berechnung der induzierten Knoten- und Kantenmengen nicht ins Gewicht fällt. Die Unterschiede zwischen der Referenzimplementierung und der GVE-Implementierung sind in diesem Fall nicht auf die Verwendung unterschiedlicher Flink-Operatoren (vgl. Evaluation von Exclusion und Overlap) zurückführen. Ein Teil lässt sich hingegen dadurch erklären, dass in der Referenzimplementierung die induzierte Knotenmenge durch einen Verbund mit der Knoten- anstatt der Graph-Menge gebildet wird.

Einige testweise Ausführungen der GVE-Implementierung auf deutlich größeren Knoten- und Kantenmengen haben hingegen ergeben, dass die Verwendung der Tabellenfunktion unter Umständen schlecht skaliert bzw. höhere Laufzeiten im Vergleich zur Referenzimplementierung erzeugt. An dieser Stelle hat der Verlust an Ausdrucksstärke evtl. auch messbare negative Implikationen (vgl. Kapitel 2.4). Im Bereich der Graph Collections besteht also ein großer Bedarf an weiterer Forschung bzw. Evaluation, der im Rahmen dieser Arbeit nicht abgedeckt werden konnte.

6.3.5 Grouping

Um eine optimale Vergleichbarkeit zur Grouping-Implementierung in Gradoop zu ermöglichen, wurden dieselben Datensätze und Konfigurationen herangezogen, die zur Evaluation der Referenzimplementierung verwendet wurden. Die Erläuterung der bestehenden Implementierung und eine ausführliche Evaluation liefern Junghanns et. al in [31]. Die darin verwendeten Datensätze wurden nicht mittels des für den SNB-Benchmark entwickelten Datengenerators erzeugt, sondern mit einem Generator für den *Graphalytics*-Benchmark (GA) [11]. Diese Datensätze lagen bereits im HDFS vor und wurden in die entsprechenden relationalen Schemata konvertiert. Tabelle 6.5 listet alle verwendeten Datensätze inklusive relevanter Metriken auf - die mit S bezeichneten Versionen sind bereits auf Knoten mit Bezeichner `Person` und `knows`-Kanten reduziert.

Datensatz	Anzahl		Datengröße			
	Knoten	Kanten	Gradoop	GVE-Schema	Vertikales Schema	Horizontales Schema
GA10	0,3 M	16,6 M	1,5 GB	2,0 GB	2,7 GB	2,6 GB
GA100	1,7 M	147,4 M	12,9 GB	18,2 GB	24,1 GB	23,5 GB
GA1000	12,8 M	1.363,7 M	118,6 GB	169,0 GB	224,8 GB	218,4 GB
GA10-S	0,2 M	10,2 M	0,9 GB	1,3 GB	1,8 GB	1,7 GB
GA100-S	1,7 M	101,7 M	8,9 GB	13,2 GB	17,9 GB	17,2 GB
GA1000-S	12,8 M	1.014,7 M	88,1 GB	131,2 GB	177,1 GB	170,89 GB

Tabelle 6.5: Zur Evaluation herangezogene Datensätze

In [31] werden insgesamt 13 Grouping-Konfigurationen, i. e. Attribute und Aggregatfunktionen, definiert - drei davon wurden im Rahmen dieser Evaluation aufgegriffen. Tabelle 6.6 listet diese Konfigurationen auf. In K-1 werden nur die Knoten nach Bezeichner gruppiert; K-4 entspricht dem Beispiel aus Kapitel 2.2.4. K-13 beinhaltet als einzige Konfigurationen Eigenschaften, nach denen gruppiert werden soll.

Konfiguration	Datensatz	K_v	K_ϵ	Λ_v	Λ_ϵ
K-1	GA	$\{\tau\}$	$\emptyset$	$\emptyset$	$\emptyset$
K-4	GA	$\{\tau\}$	`COUNT()`	$\{\tau\}$	`COUNT()`
K-13	GA-S	{`city`}	`COUNT()`	$\{\tau\}$	{`COUNT()`,`MIN(since)`,`MAX(since)`}

Tabelle 6.6: Evaluierte Grouping-Konfigurationen

Im Rahmen dieser Arbeit wurden keine Laufzeiten der Grouping-Referenzimplementierung gemessen, sondern stets die in [31] veröffentlichten Werte herangezogen. Abbildungen 6.7 (a) und (b) zeigen die Laufzeiten inklusive Speedup für Konfiguration K-4 auf dem GA1000-Datensatz. Es lässt sich erneut beobachten, dass die Referenzimplementierung höhere Laufzeiten aufweist - darüber hinaus skaliert sie allerdings nicht besser als die relational abstrahierten Implementierungen (vergleiche auch die Diskussion zu $p = 6$ zu Beginn des Kapitels). Die gemessenen Werte des vertikalen, horizontalen und GVA-Schemas unterscheiden sich nur marginal, da weder für Knoten noch für Kanten Eigenschaftswerte herangezogen werden müssen. Die vernachlässigbaren Unterschiede lassen sich nur durch verschiedenes E/A-Verhalten beim Speichern des berechneten zusammenfassenden Graphen(-Eigenschaften) erklären. Der Laufzeitvorteil gegenüber der Gradoop-Implementierung lässt sich unter anderem auf ein Implementierungsdetail zurückführen: Ein zentraler Bestandteil des Grouping-Datenflusses ist die Zuordnung von ur-

sprünglichen Knoten zu den Superknoten. Diese wird in der relationalen Abstraktion durch einen Verbund mit einer Selektionsbedingung auf allen Gruppierungsattributen gelöst. In der Referenzimplementierung ist dieser Verbund nicht nötig, da nach einem `groupBy`-Operator stets eine nutzerdefinierte Funktion ausgeführt wird, die einer Aggregationsfunktion ähnelt. Diese kann mehr als ein Ergebnis-Tupel liefern und somit den Superknoten sowie alle ursprünglichen Knoten mit einer Referenz auf jenen Superknoten zurückliefern. Knoten und Superknoten liegen in der Referenzimplementierung also in einem gemeinsamen verteilten Datensatz. Betrachtet man die Evaluationsergebnisse, ist dies jedoch offensichtlich kein Vorteil. Die Diskussion verdeutlicht, dass der Verlust an Ausdrucksstärke und der Verzicht auf benutzerdefinierte Funktionen am Beispiel des Grouping von Vorteil ist. Abbildung 6.7(c) veranschaulicht die Skalierung der Laufzeit bezogen auf die Datensatzgröße. Offensichtlich skaliert Gradoop annähernd linear, während die relationalen Varianten einen gewissen Zusatzaufwand auf kleinen Datensätzen verursachen.

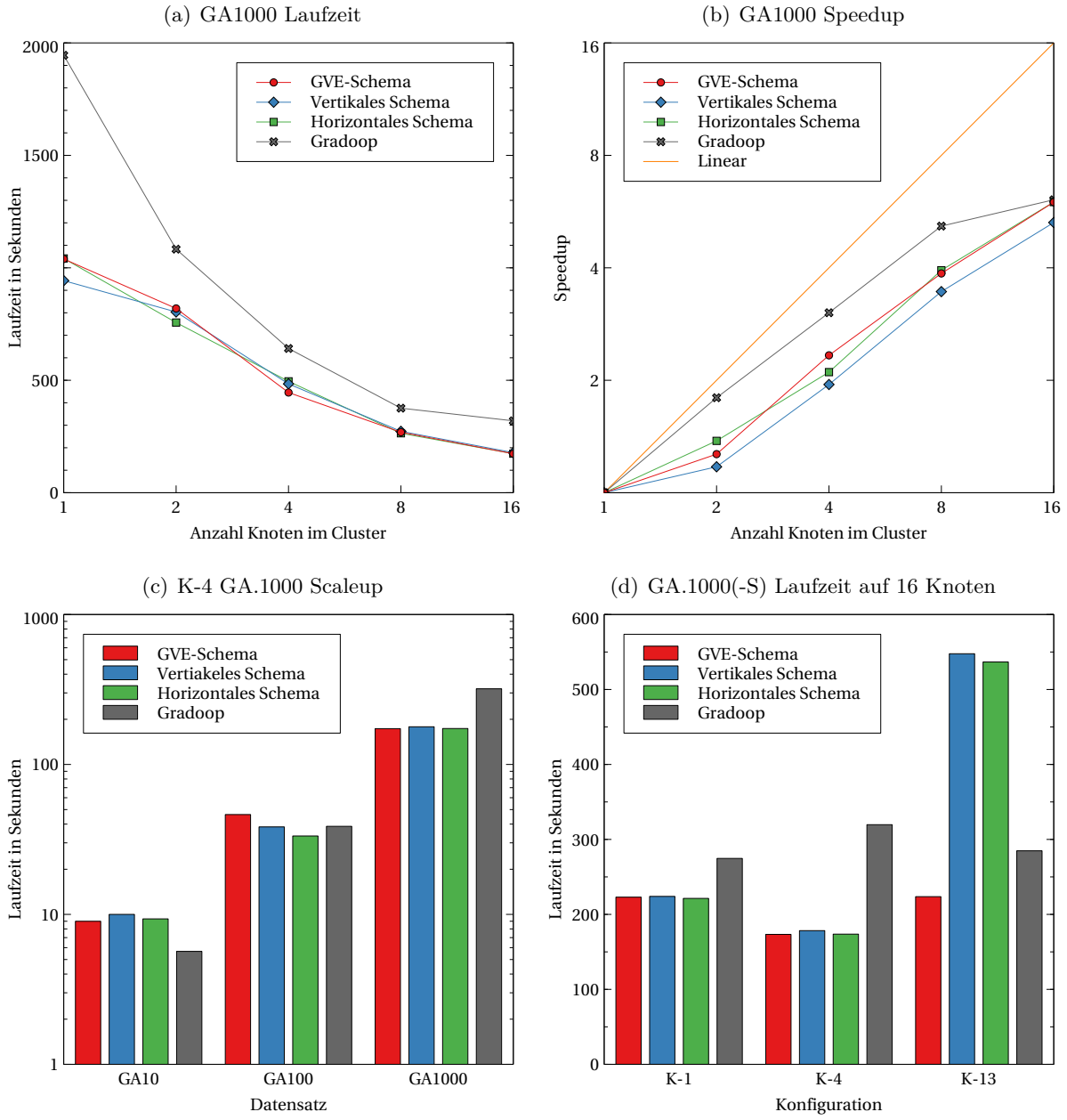


Abbildung 6.7: Laufzeit, Speedup und Scaleup des Grouping-Operators

In der bislang betrachteten Konfiguration K-4 wurden keine Eigenschaften zur Gruppierung herangezogen - dadurch ergeben sich kaum Unterschiede unter den relationalen Implementierungen. Betrachte zur weiteren Analyse Abbildung 6.7(d) - darin werden die Laufzeiten der verschiedenen Konfigurationen bei maximaler Parallelität gegenübergestellt. In K-13 weisen die Implementierung im vertikalen und horizontalen Schema offensichtlich eine deutlich höhere Laufzeit auf, als die übrigen Varianten. Dies ist insbesondere auf die Zusammensetzung der Kanten zurückzuführen: Es gibt keine Kanteneigenschaftswerte, die nicht zur Gruppierung herangezogen werden. Der nötige Verbund erzeugt also im Vergleich zum GVE-Schema viel Zusatz- bzw. Netzwerkaufwand. Im Rahmen dieser Arbeit stand der Vergleich zu [31] im Vordergrund - im weiteren Verlauf der Forschung gilt es jedoch zu untersuchen, wie sich die Implementierungen bei geringer Selektivität auf Eigenschaftswerten verhalten.

Der Laufzeitunterschied der relationalen Implementierungen zwischen K-1 und K-4 ist darauf zurückzuführen, dass der relationale Optimierer in K-1 den Gruppierungsoperator durch ein alleinstehendes `Distinct` ersetzt. In dieser Version hat der Workflow allerdings eine höhere Laufzeit als unter Verwendung einer zuvor ausgeführten Gruppierung. Diese Verhalten findet sich in der Referenzimplementierung nicht wieder, da die Flink-Operatoren auf einem geringeren Abstraktionslevel definiert werden. An diesem Beispiel wird deutlich, dass die Formulierung von Workflows auf einem höheren (relationalen) Abstraktionslevel nicht in allen Fällen besser optimiert werden kann.

7 Ausblick

Zum Abschluss dieser Arbeit werden Vorschläge und Ideen für zukünftige weitere Forschung in dem in dieser Arbeit behandelten Themengebiet unterbreitet.

Um weitere Aussagen über die Laufzeit und Skalierbarkeit der im Rahmen dieser Arbeit entstandene EPGM-Implementierung machen zu können, ist eine erweiterte Evaluation denkbar. Wünschenswert wäre zum Beispiel eine Evaluation auf Datensätzen aus der realen Welt. Darüber hinaus ist ein Vergleich zur Referenzimplementierung anhand der Ausführung eines ganzen Workflows, der mehrere verkettete Operatoren enthält, sinnvoll, um ein repräsentatives Gesamtbild zu erhalten. Weiterhin sollten die Operatoren jeweils mit mehr Konfigurationen evaluiert werden, um Spezialfälle abzufangen bzw. auszuschließen. Der Subgraph-Operator sollte der Vollständigkeit halber in der vollständigen Version inklusive Verifizierung der Kanten evaluiert werden. Generell besteht Optimierungspotential in der Konzeption der Evaluation: Bei einer Ausführung auf geringer Parallelität p sollte der Cluster inklusive verteiltem Dateisystem spezifisch für diese Parallelität konfiguriert sein, i. e. die Daten sollten wirklich nur auf p Knoten und das Programm tatsächlich nur auf p Knoten ausgeführt werden. Es sei darauf hingewiesen, dass hinsichtlich der Mengenoperatoren auf Graph Collections weiterer Forschungs- und Evaluierungsbedarf besteht - nicht nur im Kontext dieser Arbeit, sondern auch hinsichtlich des EPGM und Gradoop im übergreifenden Kontext. Es empfiehlt sich in Zukunft größere Datensätzen mit echten `n:m`-Beziehungen zu bearbeiten.

Teil des Ergebnisses dieser Arbeit ist, dass die meisten Operatoren gar nicht oder nur in ineffizienter Weise unter ausschließlicher (technischer) Verwendung der Table-API implementiert werden können. Wie bereits mehrfach erläutert, wird an vielen Stellen auf die Dataset-API zurückgegriffen. Daher liegt der Schluss nahe, den Wechsel zwischen Table- und DataSet-API mit dem Ziel der besseren Performance bzw. Optimierung weiter auszubauen. Unter der Voraussetzung, dass keine Schema-Transformationen durchgeführt werden, können dadurch möglicherweise die Vorteile beider Abstraktionsebenen miteinander kombiniert werden. Die Verwendung der DataSet-API würde z. B. ermöglichen, bestimmte Verbundoperationen innerhalb eines Table-Workflows zu optimieren. Weiterhin könnte anstelle von komplexen Konstrukten mit Tabellenfunktionen auf den FlatMap-Operator zurückgegriffen werden, wenn sich dieser als einfacher und effizienter erweist. Die Kodierung der Eigenschaften-Objekte mit Base64 sollte möglicherweise überdacht werden, da dadurch ein zusätzlicher Speicheraufwand entsteht (vergleiche Evaluation). Darüber hinaus sollten die übrigen von Junghanns et. al. definierten Operatoren unter Verwendung der Table-API implementiert werden, um weitere Erkenntnisse zu gewinnen. Im Fokus sollten dabei z. B. die Mustersuche in Graphen stehen - an einer Unterstützung der Abfragesprache OpenCypher wird an der Abteilung für Datenbanken bereits gearbeitet. Darüber hinaus sind die iterativen Graphalgorithmen von Interesse (vgl. auch die verwandten Arbeiten).

Die im Rahmen dieser Arbeit entwickelten und implementierten Schemata sind wenig komplex. Die Ergebnisse der Evaluation sollen Anlass geben, weitere Schemata zu implementieren und evaluieren. Eine mögliche Adaption wäre beispielsweise die Kombination aus GVE- und relationalen Schemata, in denen die Graphzugehörigkeit in einer eigenen Tabelle gehalten wird. Die

gilt es insbesondere im Kontext von Graph Collections zu evaluieren. Eine einfache aber möglicherweise sehr effiziente Anpassung besteht darin, die Eigenschaften im GVE-Schema jeweils in eigenen Spalten zu speichern. Dies würde den durch den Datentypen `Properties` entstehenden Zusatzaufwand verringern und bei optimierten Datenquellen ermöglichen, dass nur diejenigen Eigenschaftswerte eingelesen werden, die auch tatsächlich benötigt werden. Ein solches Schema wäre analog zum horizontalen Schema allerdings hinsichtlich der Menge an Eigenschaftsschlüsseln unflexibel. Um dem zu begegnen, wäre es möglich, auf die von Bornea et. al. in [10] vorgestellte Idee des *Schredderns* der Daten in eine feste Anzahl an Spalten zurückzugreifen. Darüber hinaus besteht die Möglichkeit, die Adjazenz analog zu Sun et. al. [42] in der *geschredderten* Form abzulegen. Als neuartige Speicherung der Graphzugehörigkeit könnte ebenfalls die Idee des *Schredderns* aufgegriffen werden und evaluiert werden. Unabhängig davon besteht weiterer Bedarf an Evaluation verschiedener relationaler Speicherungsformen der Graphadjazenz bzw. der Kanten - zu beachten ist dabei, dass das EPGM als Multigraph definiert ist. Viele Ideen wurden dazu im Kontext von RDF veröffentlicht - im Bereich von Property Graphs oder gar des EPGM besteht offensichtlich noch viel Forschungsbedarf. Die im Kontext von relationalen Datenbankmanagementsystemen häufig untersuchten Indexstrukturen (auf verteilten Daten) gilt es in Zukunft auch innerhalb von Gradoop anzuwenden bzw. zu erforschen. Von Interesse sind unter anderem auch Strukturen, die ein schnelles Navigieren innerhalb des Graphen ermöglichen.

Um eine möglichst optimale Vergleichbarkeit zur Referenzimplementierung Gradoop zu ermöglichen, wurden viele Voraussetzungen und Bedingungen daraus übernommen. Einige davon sollten mit Hinblick auf die Operator-Laufzeiten und mögliches Optimierungspotential möglicherweise überdacht werden. So würden beispielsweise eine Vielzahl an Operationen eingespart werden, wenn in bestimmten Fällen auf das Halten der trivialen Graphzugehörigkeit von logischen Graphen verzichtet würde. Oftmals erfüllt diese bei einfachen Operator-Ausführungen keinen tatsächlichen Zweck, erzeugt allerdings zusätzlichen Aufwand bzw. zusätzliche Daten in eigentlich nicht benötigten Tabellen. Ähnlich verhält es sich mit der im Rahmen dieser Arbeit geforderten Surjektivität: Diese verursacht im vertikalen und horizontalen Schema insbesondere nach der eigentlichen Operatorausführung eine Menge an Verbundoperationen.

Denkbar sind zukünftig auch EPGM- bzw. Operator-Implementierungen in Form von SQL-Abfragen auf relationalen Datenbanksystemen (vergleichbar zu den Studien von Jindal et. al. in [28] oder Fan et. al. in [22]). Von Interesse wäre insbesondere ein Laufzeitvergleich zu der Ausführung mit Gradoop auf horizontal skalierten Clustern. Wie in dieser Arbeit gezeigt wurde, lassen sich die behandelten EPGM-Operatoren nur unter Zuhilfenahme von nutzerdefinierten Funktionen implementieren - im Kontext von RDBMS wird von *gespeicherten Prozeduren* gesprochen, die in diesem Fall bereitgestellt werden müssten. Darüber hinaus ließe sich auch das Konzept einer abstrakten Programmierschnittstelle von Gradoop erweitern: Denkbar wäre eine abstrakte relationale Schnittstelle für den Endnutzer, mit der sich z. B. unter Verwendung einer SQL-Syntax eine Selektion der in einer Graph Collection enthaltenen logischen Graphen formulieren lässt - in Anlehnung an die Idee von *GraphFrames* (Dave et. al. [19]).

Die Arbeit ist ein erster Beitrag zu der Entwicklung von Gradoop hin zu einem Verarbeitungssystem für Graph-Streams: Wie bereits in den Grundlagen erläutert, können mit der Table-API von Apache Flink auch kontinuierliche Daten verarbeitet werden. An Anwendungsszenarien und einer Implementierung im Kontext von Graphen wird in der entsprechenden Arbeitsgruppe bereits gearbeitet. Von Interesse ist dabei beispielsweise eine Grouping-Implementierung, die auf kontinuierlichen Daten arbeitet.

Literaturverzeichnis

[1] Bson specification version 1.1. `http://bsonspec.org/spec.html`. Abgerufen: November 2019.

[2] Dataflow programming model. `https://ci.apache.org/projects/flink/flink-docs-release-1.6/concepts/programming-model.html`. Abgerufen: November 2019.

[3] The gremlin graph traversal machine and language. `https://tinkerpop.apache.org/gremlin.html`. Abgerufen: November 2019.

[4] Introducing gelly: Graph processing with apache flink. `https://flink.apache.org/news/2015/08/24/introducing-flink-gelly.html`. Abgerufen: November 2019.

[5] Introduction to apache giraph. `https://giraph.apache.org/intro.html`. Abgerufen: November 2019.

[6] Daniel J Abadi, Adam Marcus, Samuel R Madden, and Kate Hollenbach. Scalable semantic web data management using vertical partitioning. In *Proceedings of the 33rd international conference on Very large data bases*, pages 411–422. VLDB Endowment, 2007.

[7] Amr Azzam, Sabrina Kirrane, and Axel Polleres. Towards making distributed rdf processing flinker. In *2018 4th International Conference on Big Data Innovations and Applications (Innovate-Data)*, pages 9–16. IEEE, 2018.

[8] Edmon Begoli, Jesús Camacho-Rodríguez, Julian Hyde, Michael J Mior, and Daniel Lemire. Apache calcite: A foundational framework for optimized query processing over heterogeneous data sources. In *Proceedings of the 2018 International Conference on Management of Data*, pages 221–230. ACM, 2018.

[9] Tim Berners-Lee, James Hendler, Ora Lassila, et al. The semantic web. *Scientific american*, 284(5):28–37, 2001.

[10] Mihaela A Bornea, Julian Dolby, Anastasios Kementsietsidis, Kavitha Srinivas, Patrick Dantressangle, Octavian Udrea, and Bishwaranjan Bhattacharjee. Building an efficient rdf store over a relational database. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 121–132. ACM, 2013.

[11] Mihai Capotă, Tim Hegeman, Alexandru Iosup, Arnau Prat-Pérez, Orri Erling, and Peter Boncz. Graphalytics: A big data benchmark for graph-processing platforms. In *Proceedings of the GRADES'15*, page 7. ACM, 2015.

[12] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4), 2015.

[13] Stefano Ceri, Georg Gottlob, and Letizia Tanca. What you always wanted to know about datalog (and never dared to ask). *IEEE transactions on knowledge and data engineering*, 1(1):146–166, 1989.

[14] Donald D Chamberlin and Raymond F Boyce. Sequel: A structured english query language. In *Proceedings of the 1974 ACM SIGFIDET (now SIGMOD) workshop on Data description, access and control*, pages 249–264. ACM, 1974.

[15] Surajit Chaudhuri. An overview of query optimization in relational systems. In *Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 34–43. ACM, 1998.

[16] Edgar F Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[17] World Wide Web Consortium. Sparql query language for rdf. `https://www.w3.org/TR/rdf-sparql-query`. Abgerufen: November 2019.

[18] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2009.

[19] Ankur Dave, Alekh Jindal, Li Erran Li, Reynold Xin, Joseph Gonzalez, and Matei Zaharia. Graphframes: an integrated api for mixing graph and relational queries. In *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems*, page 2. ACM, 2016.

[20] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[21] Orri Erling, Alex Averbuch, Josep Larriba-Pey, Hassan Chafi, Andrey Gubichev, Arnau Prat, Minh-Duc Pham, and Peter Boncz. The ldbc social network benchmark: Interactive workload. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 619–630. ACM, 2015.

[22] Jing Fan, Adalbert Gerald Soosai Raj, and Jignesh M Patel. The case against specialized graph analytics engines. In *CIDR*, 2015.

[23] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. Cypher: An evolving query language for property graphs. In *Proceedings of the 2018 International Conference on Management of Data*, pages 1433–1445. ACM, 2018.

[24] Jun Gao, Jiashuai Zhou, Chang Zhou, and Jeffrey Xu Yu. Glog: A high level graph analysis system using mapreduce. In *2014 IEEE 30th International Conference on Data Engineering*, pages 544–555. IEEE, 2014.

[25] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *ACM SIGOPS operating systems review*, volume 37, pages 29–43. ACM, 2003.

[26] Joseph E Gonzalez, Reynold S Xin, Ankur Dave, Daniel Crankshaw, Michael J Franklin, and Ion Stoica. Graphx: Graph processing in a distributed dataflow framework. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, pages 599–613, 2014.

[27] Alastair Green, Martin Junghanns, Max Kießling, Tobias Lindaaker, Stefan Plantikow, and Petra Selmer. opencypher: New directions in property graph querying. In *EDBT*, pages 520–523, 2018.

[28] Alekh Jindal, Samuel Madden, Malú Castellanos, and Meichun Hsu. Graph analytics using vertica relational database. In *2015 IEEE International Conference on Big Data (Big Data)*, pages 1191–1200. IEEE, 2015.

[29] Martin Junghanns, Max Kießling, Alex Averbuch, André Petermann, and Erhard Rahm. Cypher-based graph pattern matching in gradoop. In *Proceedings of the Fifth International Workshop on Graph Data-management Experiences & Systems*, page 3. ACM, 2017.

[30] Martin Junghanns, André Petermann, Kevin Gómez, and Erhard Rahm. Gradoop: Scalable graph data management and analytics with hadoop. *arXiv preprint arXiv:1506.00548*, 2015.

[31] Martin Junghanns, André Petermann, and Erhard Rahm. Distributed grouping of property graphs with gradoop. *Datenbanksysteme für Business, Technologie und Web (BTW 2017)*, 2017.

[32] Martin Junghanns, André Petermann, Niklas Teichmann, Kevin Gómez, and Erhard Rahm. Analyzing extended property graphs with apache flink. In *Proceedings of the 1st ACM SIGMOD Workshop on Network Data Analytics*, page 3. ACM, 2016.

[33] Andrew Lamb, Matt Fuller, Ramakrishna Varadarajan, Nga Tran, Ben Vandiver, Lyric Doshi, and Chuck Bear. The vertica analytic database: C-store 7 years later. *Proceedings of the VLDB Endowment*, 5(12):1790–1801, 2012.

[34] Ora Lassila and Ralph R Swick. Resource description framework (rdf) model and syntax specification. 1999.

[35] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146. ACM, 2010.

[36] Maged Michael, Jose E Moreira, Doron Shiloach, and Robert W Wisniewski. Scale-up x scale-out: A case study using nutch/lucene. In *2007 IEEE International Parallel and Distributed Processing Symposium*, pages 1–8. IEEE, 2007.

[37] Justin J Miller. Graph database applications and concepts with neo4j. In *Proceedings of the Southern Association for Information Systems Conference, Atlanta, GA, USA*, volume 2324, 2013.

[38] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: a not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1099–1110. ACM, 2008.

[39] Ian Robinson, Jim Webber, and Emil Eifrem. *Graph Databases: New Opportunities for Connected Data*. O'Reilly Media, Inc., 2nd edition, 2015.

[40] Sherif Sakr and Ghazi Al-Naymat. Relational processing of rdf queries: a survey. *ACM SIGMOD Record*, 38(4):23–28, 2010.

[41] Alexander Schätzle, Martin Przyjaciel-Zablocki, Simon Skilevic, and Georg Lausen. S2rdf: Rdf querying with sparql on spark. *Proceedings of the VLDB Endowment*, 9(10):804–815, 2016.

[42] Wen Sun, Achille Fokoue, Kavitha Srinivas, Anastasios Kementsietsidis, Gang Hu, and Guotong Xie. Sqlgraph: An efficient relational-based property graph store. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1887–1901. ACM, 2015.

[43] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Ning Zhang, Suresh Antony, Hao Liu, and Raghotham Murthy. Hive-a petabyte scale data warehouse using hadoop. In *2010 IEEE 26th international conference on data engineering (ICDE 2010)*, pages 996–1005. IEEE, 2010.

[44] Apache TinkerPop. Table api sql: User-defined functions. `https://ci.apache.org/projects/flink/flink-docs-release-1.6/dev/table/udfs.html`. Abgerufen: November 2019.

[45] Cathrin Weiss, Panagiotis Karras, and Abraham Bernstein. Hexastore: sextuple indexing for semantic web data management. *Proceedings of the VLDB Endowment*, 1(1):1008–1019, 2008.

[46] Kevin Wilkinson, Craig Sayers, Harumi Kuno, and Dave Reynolds. Efficient rdf storage and retrieval in jena2. In *Proceedings of the First International Conference on Semantic Web and Databases*, pages 120–139. Citeseer, 2003.

[47] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

Abbildungsverzeichnis

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbstständig 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.
Ich versichere, dass das elektronische Exemplar mit den gedruckten Exemplaren übereinstimmt.

Leipzig, den 31.12.2019

_____

Elias Saalmann