UNIVERSITÄT
LEIPZIG

Institut für Informatik
Fakultät für Mathematik und Informatik
Abteilung Datenbanken

Optimierung des Gradoop I/O mittles Columnar Store Apache Parquet

Bachelorarbeit

vorgelegt von:
Maurice Eisenblätter

Matrikelnummer:
3711464

Betreuer:
Prof. Dr. Erhard Rahm
Kevin Gomez

Abstract

Diese Arbeit integriert das spaltenorientierte Speicherformat Apache Parquet [1] in das von der Abteilung für Datenbanken an der Universität Leipzig entwickelte parallele Graphverarbeitungssystem Gradoop [2]. Gradoop ist die Referenzimplementierung des EPGMs und TPGMs und basiert auf Apache Flinks [3, 4] DataSet API. Ziel ist es die bisherige I/O Leistung von Gradoops Hauptpeicherformat CSV zu übertreffen und eine Schnittstelle für die zukünftige Verwendung von Projection- und Filter-Pushdown mit Apache Flinks [3, 4] Table API zu schaffen. Um dieses Ziel zu erfüllen, werden zwei verschiedene auf Parquet basierende Speicherformate für Gradoop entworfen und getestet. Der Erste verwendet Parquets Protobuf Integration und der Zweite verwendet eine speziell auf Gradoop zugeschnittene Implementation von Parquet. Darüber hinaus werden alle Speicherformate intensiv getestet und miteinander verglichen, um die Frage beantworten zu können, welches Format sich für welchen Verwendungszweck und welche Datenmenge am besten eignet. Zu diesem Zweck wird eine Testsuite mit verschiedenen Operatoren und Datensätzen verschiedenster Größen verwendet. Insgesamt, konnte das Datenvolumen im Durchschnitt auf 25% der Originalgröße reduziert werden. Des Weiteren konnten fast alle Benchmarks beschleunigt werden und erreichten teilweise sogar 10% der Originallaufzeit von CSV.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Listings

Abkürzungsverzeichnis

CBA CitiBike Analytical 28–30, 32–34

EPGM Extended Property Graph Model II, 1, 4, 5, 7, 14, 16, 18, 20, 24

GDL Graph Definition Language 7

HDFS Hadoop Distributed File System 3, 5–7

IDL Interface description language VII, 7, 8, 23

PGM Property Graph Model V, 1, 3, 4

POJO Plain Old Java Object 7, 22, 23, 26

SF Skalierungsfaktor 27, 33, 34

TPGM Temporal Property Graph Model II, V, 1, 4, 5, 7, 14–16, 20, 24, 27, 28

1. Einleitung

1.1. Motivation

In den letzten Jahren hat die Verarbeitung und Analyse von Graphen immer mehr an Bedeutung gewonnen, da sich Graphen eignen um reale Zusammenhänge abstrakt zu repräsentieren und sie eine Vielzahl von Techniken anbieten, um die Relationen zwischen komplexen Daten zu analysieren und zu verstehen. Besonders viel Relevanz hat die Analyse von Graphen in Bereichen wie sozialen Netzwerken, Biowissenschaften und Business Intelligence gewonnen. Des Weiteren kann eine große Menge von alltäglichen Dinge als Graph repräsentiert werden, wie zum Beispiel das Internet (Verlinkung der Websites untereinander), Stammbäume und öffentliche Verkehrsnetze (allgemein Transportnetze). Mit Hilfe der Analyse eines Graphen können die unterschiedlichsten Ziele verfolgt werden, wie komplexe Suchanfragen, Mustererkennungen oder allgemeines Datamining. So kann zum Beispiel für ein Verkehrsnetz analysiert werden, wie sich das Fahrverhalten von Passagieren unter bestimmten Umständen, wie der Länge der Fahrt oder Uhrzeit, verändert. Diese Daten können anschließend verwendet werden, um gezielte Verbesserungen am Netz vorzunehmen.

Zur Analyse und Auswertung solcher Graphen werden verschiedene, auf Graphen spezialisierte, Frameworks und Datenbanken verwendet. Die meisten dieser Frameworks und Datenbank können die Daten auf mehreren Systemen verteilt verarbeiten. Grund dafür ist, dass Graphen, wie zum Beispiel jene aus einem sozialem Netzwerk, zu groß für herkömmliche Systeme werden können. Solche Frameworks und Datenbanken verwenden für gewöhnlich einfache Graphmodelle, welche komplexe Analysen erschweren. Ein Beispiel dafür wäre das Property Graph Model (PGM), welches beispielsweise die Analyse von Gemeinschaften innerhalb eines sozialen Netzwerkes und wie diese miteinander zusammen hänge erschwert, da es kein Möglichkeit gibt mehrere Untermengen eines Graphen (Gemeinschaften) darzustellen. Daher wurde an der Universität Leipzig an einem neuen Graphmodell (EPGM [5], TPGM [6]) geforscht, welches die Analyse von Relationen zwischen Untergraphen und sich im Laufe der Zeit verändernden Graphen ermöglicht. Um eine ausführliche Analyse zu ermöglichen, definiert das Modell diverse Operatoren. Gradoop [2] ist die Referenzimplementierung des Modells und der Operatoren und basiert auf dem verteilen Datenstrom-Verarbeitungs-Framework Apache Flink [3, 4].

Das verteile Verarbeiten von Daten, wie mit Gradoop, bringt jedoch viele Schwierigkeiten mit sich. So entstehen zum Beispiel häufig hohe Kommunikationskosten zwischen den einzelnen Systemen, welche das Verarbeiten der Daten verlangsamen können oder allgemein die Effizienz des Programms senken. Ein weiterer Grund für das langsamere Verarbeiten der Daten in Gradoop ist das primär verwendete CSV Speicherformat zum Ein- und Auslesen der Daten, welches für gewöhnlich im Vergleich zu neueren Alternativen langsamer ist und mehr Speicherplatz benötigt. Neben den herkömmlichen Problemen von verteilten Systemen hat Gradoop noch ein weiteres Problem, es verwendet hauptsächlich das CSV Speicherformat für alle Analysen. Dieser ist jedoch relative langsam und groß im Vergleich zu neueren Alternativen.

Eine dieser Alternative ist Apache Parquet, da es sich einfach in das Flink Ökosystem intrigieren lässt und Platz für zukünftige Verbesserungen, wie zum Beispiel Projection- und Filter-Pushdown, bietet. Außerdem kodiert und dekodiert Parquet Daten effizienter als herkömmliches CSV.

1.2. Ziel der Arbeit

Ziel der Arbeit ist es das spaltenorientierte Speicherformat Parquet in Gradoop einzubinden, um die allgemeine Laufzeit von lese- und schreibintensiven Aufgaben zu verringern, sowie die Reduzierung des Datenvolumens der Datensätze. Außerdem soll ein Grundlage für Projection-Pushdown als auch Filter-Pushdown gelegt werden, welche von spätere Arbeiten aufgegriffen werden kann.

Des Weiteren soll eine intensive Evaluation durchgeführt werden, welche das bereits vorhandene CSV Speicherformat von Gradoop mit den in dieser Arbeit entstanden Speicherformaten vergleicht. Dafür werden ein Reihe von Benchmarks mit den verschiedenen Speicherformaten durchgeführt und bezüglich ihrer Laufzeit und Skalierbarkeit verglichen. Drüber hinaus wird das Datenvolumen der Datensätze mit verschiedenen Speicherformaten miteinander verglichen.

1.3. Aufbau der Arbeit

Die Arbeit ist wie folgt aufgebaut: In Kapitel 2 werden grundlegende Konzepte der Graphentheorie sowie diverse, für die Arbeit benötige, Softwarekomponenten vorgestellt. Anschließend werden in Kapitel 3 verschiedene spaltenorientierte Speicherformate und deren grobe Funktionsweise sowie deren Aufbau erklärt. Daraufhin werden in Kapitel 4 allgemeine Anforderungen an die Implementation sowie das Konzept der Implementation vorgestellt. Darauf aufbauend wird in Kapitel 5 die Implementation konkretisiert. Danach wird in Kapitel 6 die Implementation evaluiert und mit CSV verglichen. Abschließend folgt noch eine kurze Zusammenfassung der Ergebnisse dieser Arbeite in Kapitel 7.

2. Grundlagen

In diesem Abschnitt werden diverse Grundlagen und Techniken, welche für diese Arbeit essenziell sind, vorgestellt. Zuerst werden verschiedene Konzepte der Graphentheorie definiert. Anschließend werden die Softwarekomponenten Hadoop Distributed File System (HDFS) und Flink vorgestellt. Zum Schluss folgt ein Absatz über Gradoop und dessen Funktionsweise sowie Protobuf.

2.1. Graphen

Ein Graph ist ein Tupel $G = (V, E)$ und besteht aus einer endlichen Menge von Knoten V sowie einer endlichen Menge von Kanten $E := \{\{i, j\} \mid i, j \in V\}$. Die Kante $e = \{i, j\}$ beschreibt die Verbindung zwischen den beiden Knoten i und j in beide Richtungen. Für einen Graph G beschreibt $V(G)$ die Menge der Knoten von G und $E(G)$ die Menge der Kanten von G. Die Knoten können auch Vertex und die Kanten Edge genannt werden.

Ein Graph gilt als gerichtet genau dann, wenn die Menge der Kanten definiert ist als eine Menge von geordneten Paaren von Knoten $E := \{(i, j) \mid i, j \in V\}$. Eine gerichtete Kante $e = (i, j)$ beschreibt die Verbindung des Quellknotens i in Richtung des Zielknotens j. Falls nicht explizit angegeben, wird davon ausgegangen, dass Graphen gerichtet sind.

Ein Graph $G' = (V', E')$ wird als Subgraph oder Untergraph des Graphen $G = (V, E)$ bezeichnet genau dann, wenn $V' \subseteq V$ und $E' \subseteq ((V' \times V') \cap E)$. Ist G' ein Subgraph von G so heißt G Supergraph oder Obergraph von G'.

2.1.1. Property Graph Model

Ein weit verbreitetes Graphmodell ist das Property Graph Model (PGM), welches unter anderem häufig in Datenbanken Verwendung findet, wie beispielsweise in Neo4j [7]. In dem PGM werden den Knoten und Kanten Beschriftungen und Schlüssel-Wert-Paare zugeordnet. Die Beschriftungen können auch als Label und die Schlüssel-Wert-Paare als Eigenschaften oder Properties bezeichnet werden. Properties sind heterogen und fakultativ. Die Abbildung 2.1 zeigt einen beispielhaften PGM Graphen.

Ein Property Graph ist ein Tupel $G = (V, E, T, \tau, K, A, \kappa)$ bestehend aus einem Graphen (V, E), einer endlichen Menge von Beschriftungen, Schlüssel und Werten T, K, A, sowie den partiellen Funktionen $\tau : (V \cup E) \mapsto T$ und $\kappa : (V \cup E) \times K \mapsto A$. Die Funktion τ bildet die Knoten und Kanten auf eine Beschriftung ab und κ bildet die Knoten und Kanten in Kombination mit einem Schüssel auf einen Wert ab.

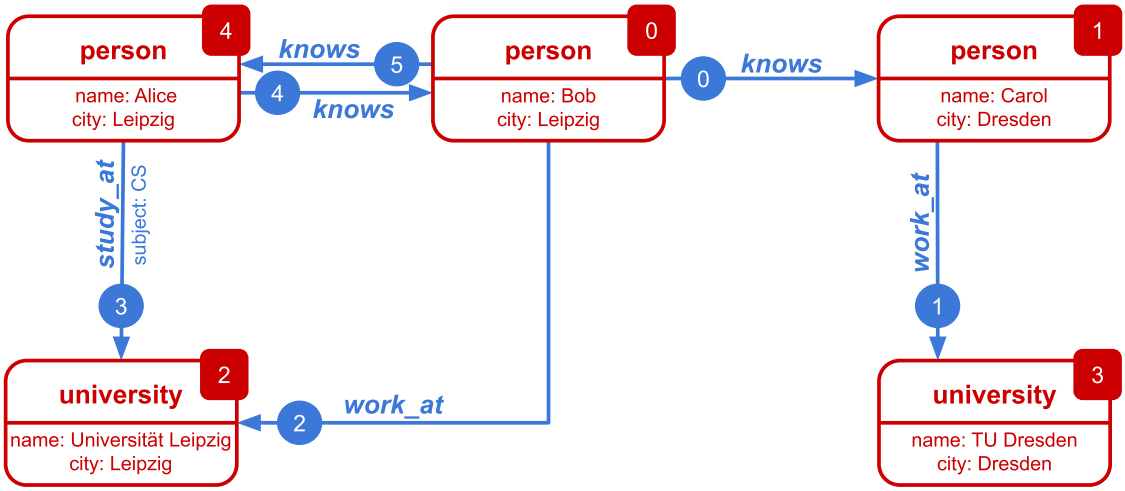

Abbildung 2.1.: Beispiel für einen PGM Graphen

2.1.2. Extended Property Graph Model

Das Extended Property Graph Model (EPGM) wurde erstmals von Junghanns et al. [5] vorgestellt. Dabei handelt es sich um eine Erweiterung des PGM, durch das Hinzufügen von logischen Graphen und einer Vielzahl von Operatoren, welche darauf anwendbar sind.

Ein Extended Property Graph ist ein Tupel $G = (V, E, L, T, \tau, K, A, \kappa)$ bestehend aus einem Property Graph $(V, E, T, \tau, K, A, \kappa)$ und einer endlichen Menge von logischen Graphen L. Die partiellen Funktionen $\tau : (L \cup V \cup E) \mapsto T$ und $\kappa : (L \cup V \cup E) \times K \mapsto A$ wurden um logische Graphen erweitert. Der Graph $G_m = (V_m, E_m)$ ist ein logischer Graph von G genau dann, wenn G_m ein Subgraph von G ist. Ein logischer Graph besitzt genauso wie die Knoten und die Kanten eine Beschriftung und fakultative Properties. Mehrere logische Graphen dürfen sich überschneiden, das heißt es gilt $\forall G_i, G_j \in G : |V(G_i) \cap V(G_j)| \geq 0 \wedge |E(G_i) \cap E(G_j)| \geq 0$. Eine Menge von logischen Graphen wird als Graphmenge oder Graphcollection bezeichnet.

Die vom EPGM definierten Operatoren können abhängig vom Operator mehrere oder einen logischen Graphen oder Graphcollection als Ein- und Ausgabe verwenden.

2.1.3. Temporal Property Graph Model

Das Temporal Property Graph Model (TPGM) bildet eine Erweiterung des EPGM, welche von Rost et al. [6] vorgeschlagen wurde. Die Erweiterung umfasst vier obligatorische zeitliche Attribute, welche den Knoten, Kanten und logischen Graphen zugeordnet werden, als auch Erweiterungen für bestehende sowie komplett neue Operatoren. Dies erlaubt das Verarbeiten und Analysieren von sich im Laufe der Zeit verändernden Graphen. Die Abbildung 2.2 zeigt einen beispielhaften TPGM Graphen, welcher eine Erweiterung des Graphen der Abbildung 2.1 darstellt. Eine derartige Erweiterung wird häufig als bitemporal bezeichnet und findet vor allem in klassischen relationalen Datenbanken Verwendung, wie beispielsweise in MariaDB (ab v10.3.4) [8].

Die vier Attribute (*tx-from*, *tx-to*, *val-from*, *val-to*) können als zwei verschiedene Zeitintervalle verstanden werden. Der Erste (*tx-from*, *tx-to*) beschreibt die Transaktion des Elements und der Zweite

(*val-from*, *val-to*) den Zeitraum in dem das Element als gültig gilt. Jeder der beiden Zeitintervalle kann je ein leerer, offener oder geschlossener Intervall sein. Das TPGM ist bis auf wenige Ausnahmen abwärtskompatibel mit dem EPGM.

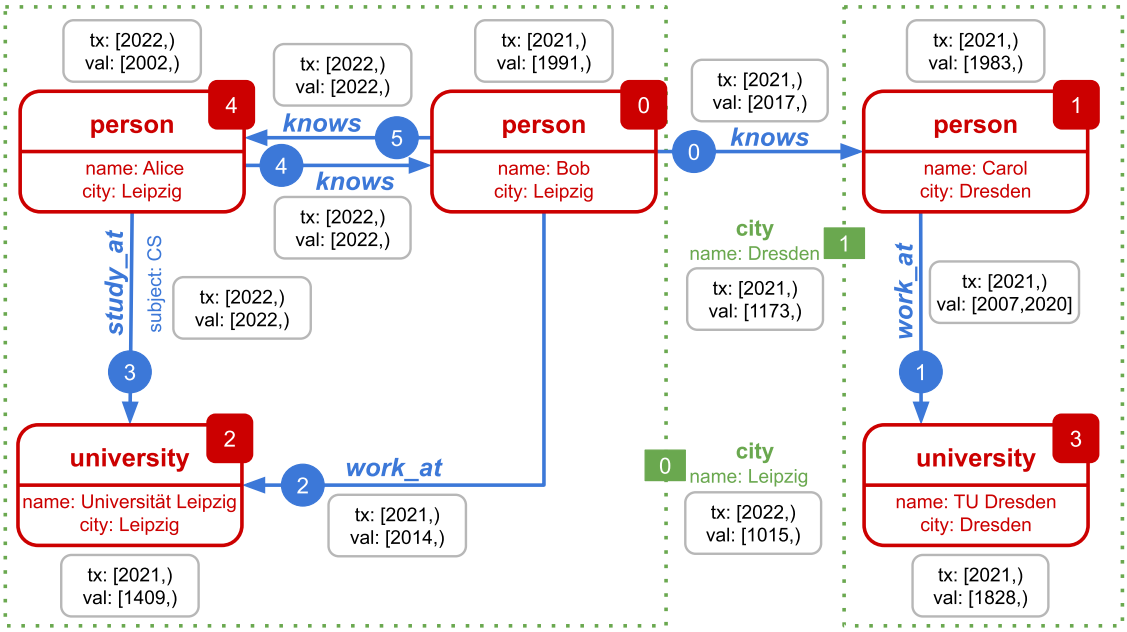

Abbildung 2.2.: Beispiel für einen TPGM Graphen

2.2. HDFS

Das Hadoop Distributed File System (HDFS) [9, 10] ist ein verteiltes Dateisystem, geschrieben in Java und ein Bestandteil von Hadoop. Ziel des HDFS ist es auf handelsüblicher Hardware zu laufen, hochgradig fehlertolerant zu sein und große Mengen von Daten zu speichern.

Es besteht aus mindestens einer NameNode und mehreren DataNodes. Die NameNodes sind für Metadaten und den geregelten Zugriff auf Dateien zuständig und die DataNodes verwalten den Speicher des Rechners, auf dem sie ausgeführt werden. Eine Datei wird in mehrere Blöcke aufgeteilt und diese Blöcke werden wiederum auf verschiedene DatenNodes abgelegt, das heißt die NameNodes kümmern sich um Operationen, wie das Öffnen oder Schließen von Dateien oder Verzeichnissen sowie das Zuordnen der Blöcke auf die jeweiligen DataNodes, während die DataNodes die Lese- und Schreiboperationen für die jeweiligen Blöcke verarbeiten.

2.3. Flink

Apache Flink [3, 4] ist ein Datenstrom-Verarbeitungs-Framework, welches in Java und Scala geschrieben wurde. Außerdem beinhaltet Flink eine Datenfluss-Engine, welche für die verteilte Verarbeitung von Datenströmen zuständig ist. Diese Engine ist in der Lage sowohl endlose (unbounded streams) als auch endliche (bounded streams) Datenströme zu verarbeiten. Des Weiteren ist sie

für Parallelisierung und Ausfallsicherheit zuständig und dafür entworfen worden, auf einem shared-nothing Cluster[1] zu arbeiten.

Ein Flink Programm besteht aus Datenströmen und Transformationen. Diese Transformationen nehmen einen oder mehrere Datenströme als Eingabe entgegen und erzeugen einen oder mehrere Datenströme als Ausgabe. Darüber hinaus verfügt jedes Flink Programm über eine oder mehrere Datenquellen und -senken. Die Ausführung eines solchen Programms beginnt immer mit der Datenquelle, welche die initialen Datenströme erzeugt und führt nach und nach die Transformationen auf den Datenströmen aus bis schließlich alle Transformationen angewandt wurden und die letzten Ausgabedatenströme an die Datensenke weiter gegeben werden.

Weitere Bestandteil von Flink ist die TableApi, welche das Verarbeiten von Daten auf einer höheren Abstraktionsebene erlaubt. Die TableApi verwendet Apache Calcite [11] um das Flink Programm zu optimieren. Zu den Optimierungen zählen unter anderem: Filter- und Projection-Pushdown, welche dafür verantwortlich sind bereits in der Datenquelle unbrauchbare Daten zu überspringen oder gar nicht erst zu lesen. Projection-Pushdown schränkt die zu lesenden Felder ein und der Filter schränkt die zu lesenden Werte der Felder ein.

Flink stellt kein eigenes verteiltes Datenspeichersystem bereit, aber bietet diverse Datenquellen und -senken für bereits existierende Systeme, wie zum Beispiel das HDFS, an. Zusätzlich bietet Flink die Möglichkeit eigenen Datenquellen und -senken einzubinden, dazu zählen auch bereits existierenden Datenquellen oder -senken für Hadoop MapReduce (Hadoop Input-/OutputFormat).

In Flink wird eine Datenquelle in mehrere *Splits* aufgeteilt. Ein *Split* repräsentiert einen Teil der Datenquelle, wie zum Beispiel einen Ausschnitt einer Datei und wird jeweils von einem Worker (Task Manager) verarbeitet. Die *Splits* wiederum werden von dem Job Manager einmalig generiert und an die Worker verteilt. Die Abbildung 2.3 zeigte wie eine Datenquelle in Flink verarbeitet wird. Sobald ein Worker mit seinem *Split* fertig ist, wird der nächste *Split* beim Job Manager angefragt.

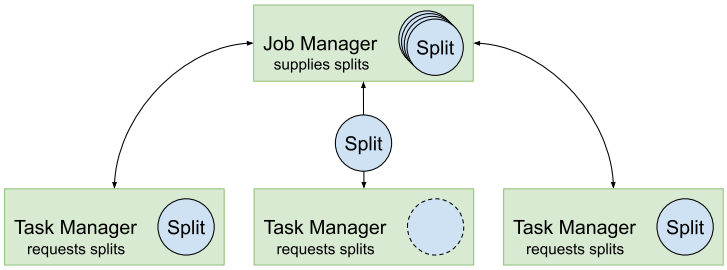

Abbildung 2.3.: Flink Datenquellen

Die genaue Funktionsweise einer Datensenke hängt von der Art der Datensenke ab. Für eine Datensenke, welche die Daten in eine Datei schreibt, wird in der Regel für jeden Worker eine Datei geöffnet in welche nur der aktuelle Worker Daten schreibt und zum Schluss diese schließt.

[1] Ein Verbund aus mehreren über ein Netzwerk verbunden Rechnern, welche keine Systemressourcen teilen.

2.4. Gradoop

Gradoop [2] ist eine paralleles Graphverarbeitungssystem, welche sowohl das EPGM als auch das TPGM unterstützt. Es basiert auf Apache Flink und verwendet das HDFS als verteiltes Datenspeichersystem. Bei Gradoop handelt es sich um ein Proof-of-Concept für das EPGM und TPGM. Das Projekt wurde erstmals von Junghanns et al. [12] vorgestellt und wird von der Abteilung für Datenbanken an der Universität Leipzig seit mehreren Jahren gepflegt und weiterentwickelt.

Die aktuellste Version von Gradoop (v0.6.0) verwendet die endlichen Datenströme (DataSet API) aus Flink. Darin werden die EPGM-Elemente (Knoten, Kanten, Logische Graphen) als Plain Old Java Objects (POJOs) dargestellt. Die Operatoren des EPGM und TPGM basieren häufig auf einer Kombination von bereits existierenden Transformationen aus Flink. Neben den Operatoren unterstützt Gradoop auch Graph Definition Language (GDL), einen eigenen Dialekt von Cypher für das EPGM und Cypher [13], einer Abfragesprache für Graphen. Außerdem bietet Gradoop diverse Datenquellen und -senken für das EPGM und TPGM an, wie zum Beispiel CSV oder HBase [14].

2.5. Protobuf

Protobuf [15] oder auch Protocol Buffers genannt, ist ein von Google entwickeltes system- und programmiersprachenunabhängiges Datenformat für das Serialisieren von verschachtelten strukturierten Daten. Die Daten-Objekte werden auch *Message* genannt und werden in der *proto* Interface description language (IDL) definiert. Die aktuellen Versionen der IDL sind *proto2* und das neuere *proto3*. Des Weiteren stellt Google mehrere Transpiler[2] zur Verfügung, welche den Quellcode für das Lesen, Schreiben und Bearbeiten der Daten in einer gewünschten Programmiersprache für ein Schema generieren. Unterstützte Programmiersprache sind unter anderem Java, C++ und weitere.

Listing 2.1 zeigt eine beispielhafte Definition einer *Message* mit dem Namen *Person*. Jedem Feld wird eine Zahl zu geordnet, welche der Identifikation des Felds, in serialisierten Daten, dient. Die Zahl muss eindeutig sein, also zum Beispiel hat das Feld *Name* die Eins und *Address* die Zwei zu geordnet. Weiterhin gilt zu beachten, dass das Feld *Street* in *Address* ebenfalls die Eins zugeordnet hat, da es Bestandteil des *Address* Felds ist und nicht ein direktes Feld in *Person*. Außerdem kann für jedes Feld die Anzahl der definierten Werte eingeschränkt werden. So kann ein Feld obligatorisch (*required*), optional (*optional*) und beliebig oft wiederholend (*repeated*) sein. Das Feld *Name* ist *required* und muss daher immer genau einmal definiert sein, währenddessen das *Address* Feld *optional* ist und daher undefiniert oder genau einmal definiert sein kann. Des Weiteren ist das Feld *Phone repeated* und kann daher beliebig oft definiert oder gar undefiniert sein.

[2] Ein Compiler, welcher den Quellcode einer Programmiersprache in den Quellcode einer anderen Programmiersprache übersetzt.

```
message Person {
  required string Name = 1;
  optional group Address = 2 {
    optional string Street = 1;
    required string City = 2;
    required int64 ZipCode = 3;
  }
  repeated group Phone = 3 {
    required string Number = 1;
    optional string Type = 2;
  }
}
```

Listing 2.1: Beispiel eins Protobuf Schemas in der *proto2* IDL

3. Related Work

In diesem Abschnitt werden diverse spaltenorientierte Speicherformat für komplexe verschachtelte Datensätze und deren Funktionsweise sowie Aufbau genauer betrachtetet.

3.1. Google Dremel

Dremel [16] ist ein von Google entworfenes verteiltes Querysystem für große Datenmengen, welches unter anderem einen komplett neuen Ansatz für das Speichern von verschachtelten Datensätzen in spaltenorientierter Repräsentation liefert. Die verschachtelten Datensätze werden wie Bäume behandelt, in denen der Datensatz die Wurzel repräsentiert und jedes Feld mit einem atomaren Wert (*integers*, *floating-point numbers*, *strings*, etc.) ein Blatt repräsentiert. Ein Knoten kann obligatorisch, optional oder wiederholend sein. Ein optionaler Knoten und dessen Kinder können entfallen (undefiniert sein). Ebenso kann ein wiederholender Knoten und dessen Kinder entfallen oder mehrfach für einen Datensatz definiert sein.

p_1
Name: 'Alice'
Address:
  Street: 'Augustusplatz 10'
  City: 'Leipzig'
  ZipCode: 04109
Phone:
  Number: '01749464308'
  Type: 'work'
Phone:
  Number: '015228817386'

p_2
Name: 'Bob'
Phone:
  Number: '01729980752'
  Type: 'mobile'

p_3
Name: 'Carol'
Address:
  City: 'Dresden'
  ZipCode: 01069

Listing 3.1: Beispiel Datensätze für Google Dremel für das Schema aus Listing 2.1

Das *repetition level* eines Blatts beschreibt die Tiefe des Knotens, von welchem die Wiederholung des Blatts ausgeht. Der *level* null notiert den Start eines neuen Datensatzes. Das *repetition level* wird verwendet, um ein Blatt genau zu einem Elternknoten zuordnen zu können. So können unter anderem sich wiederholende Felder in wiederholenden Feldern geschrieben und korrekt ausgelesen werden.

Das *definition level* eines Blatts beschreibt die Anzahl der Knoten ausgehend von der Wurzel bis zum Blatt, welche nicht definiert sein können, jedoch aber definiert sind. Ist das *definition level* kleiner als das kleinste maximale mögliche *definition level* für ein Blatt so beschreibt dieses den Wert *NULL* (undefiniert).

Jedes Blatt besitzt eine eigene Spalte, welche das *definition level*, *repetition level* und den Wert des Blatts speichert. Dies ist in der Tabelle 3.1 für die Beispieldaten aus Listing 3.1 zu sehen. Ist der Wert undefiniert, so wird dieser nicht gespeichert, da dies aus dem *definition level* herleitbar ist. Des

Name		
Wert	r	d
Alice	0	0
Bob	0	0
Carol	0	0

Address.Street		
Wert	r	d
Augustusplatz 10	0	2
NULL	0	0
NULL	0	1

Address.City		
Wert	r	d
Leipzig	0	1
NULL	0	0
Dresden	0	1

Address.ZipCode		
Wert	r	d
04109	0	1
NULL	0	0
01069	0	1

Phone.Number		
Wert	r	d
01749464308	0	1
015228817386	1	1
01729980752	0	1
NULL	0	0

Phone.Type		
Wert	r	d
work	0	2
NULL	1	1
mobile	0	2
NULL	0	0

Tabelle 3.1.: Spaltenweise Darstellung der Beispieldaten aus Listing 3.1 mit repetition levels (r) und definition levels (d)

Weiteren können *definition levels* entfallen, wenn das Feld obligatorisch ist, genauso wie *repetition level* nur gespeichert werden, wenn diese zwingend benötigt werden. Darüber hinaus können die Datensätze mit Hilfe der *level* partiell rekonstruiert werden, dass heißt es werden nur für den Anwendungsfall benötigte Felder konstruiert. Diese Optimierung wird auch häufig als Projection-Pushdown bezeichnet.

3.2. Apache Parquet

Apache Parquet [1] ist ein spaltenorientiertes Speicherformat, welcher spezielle für Hadoop und das HDFS entwickelt wurde. Des Weiteren unterstützt Parquet komplexe verschachtelte Datensätze und verwendet für dessen Serialisierung den zuvor vorgestellten Ansatz aus Google Dremel, jedoch werden die eigentlichen atomaren Werte mit effizienten Kodierungen und Komprimierungen serialisiert. Da Parquet genau wie Google Dremel Datensätze in Spalten abspeichert ist es auch hier möglich Projection-Pushdown zu verwenden.

Eine Parquet Datei besteht aus einem Footer sowie einer oder mehreren *row groups*. Der Footer enthält für das Einlesen der Datei wichtig Metadaten, wie zum Beispiel das Schema der Datensätze sowie die Position der *row groups* und *column chunks*. Eine *row group* ist eine Untermenge von Datensätzen, welche wiederum in *column chunks* unterteilt wird, sodass jeder Spalte genau ein *column chunk* zugeordnet wird. Ein *column chunk* hingegen besteht aus einer oder mehreren *pages*, welche die eigentlichen Datensätze enthalten. Die Abbildung 3.1 zeigt den vereinfachten Aufbau einer Parquet-Datei.

Eine *page* besteht aus dem *page*-Header, den *repetition levels*, den *definition levels* und den kodierten Werten. Ähnlich wie bei Google Dremel werden die *repetition levels* und *definition levels* nur gespeichert, wenn dies zwingend notwendig ist. Außerdem werden diese mit einem Hybriden aus Lauflängenkodierung und Bit-Packing kodiert, um diese möglichst platzsparend und effizient abspeichern zu können. Dies erlaubt unter anderem die Kodierung von mehreren undefinierten Werten in einer einzigen Lauflängenkodierung, da undefinierte Werte von den *definition levels* abgeleitet

werden. Die eigentlichen Werte werden ebenfalls mit verschiedenen Variationen von Lauflängenkodierung und Bit-Packing kodiert sowie zusätzlich komprimiert, um Speicherplatz zu sparen. Der Header einer *page* kann abhängig von der Art der *page* verschiedene Daten enthalten. Unter anderem setzt sich der Header einer Wörterbuch-*page* anderes zusammen als der einer Daten-*page*. Zusätzlich enthält eine *page* Statistiken über die eigentlichen gespeicherten Werte wie zum Beispiel, den kleinsten und größten gespeicherten Wert. Zusätzlich können diese Kennwerte direkt vor dem Footer gespeichert werden, um *pages* effizient überspringen zu können, ohne jeden *page*-Header einzeln einlesen zu müssen. Ziel dieser Statistiken ist es Filter-Pushdown zu unterstützten. Zudem werden auch Bloom Filter unterstützt, welche ebenfalls für Filter-Pushdown verwendet werden können. Ein Bloom Filter kann für einen beliebigen Wert Aussage darüber treffen, ob er eventuell oder gar nicht in einer Menge (*column chunk*) enthalten ist.

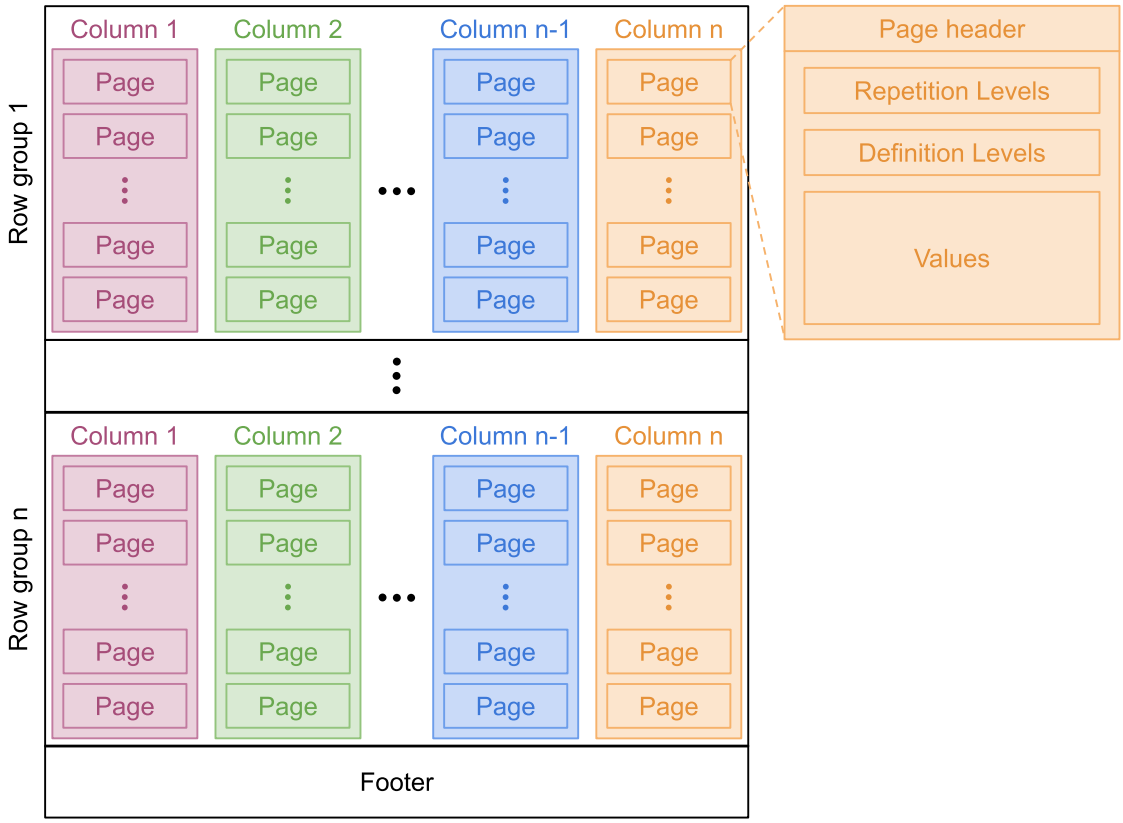

Abbildung 3.1.: Vereinfachter Aufbau einer Parquet-Datei

3.3. Apache ORC

Apache ORC [17] ist ein spaltenorientiertes Speicherformat, welches für Hive entwickelt wurde und komplexe verschachtelte Datensätze unterstützt, jedoch verwendet ORC nicht Google Dremel. Stattdessen wird ein verschachtelter Datensatz als Baum betrachtet, welcher durch eine Vorordnungs-Traversierung (pre-order traversel) in eine eindimensionale Liste geebnet wird. Jedem Knoten innerhalb des Baums wird somit eine Spalte und eindeutige Spalten-Id zugeordnet. Abbildung 3.2 zeigt den Datensatz aus Listing 3.1 als Typbaum für ORC inklusive Spalten-Id, Typ und Spaltenname.

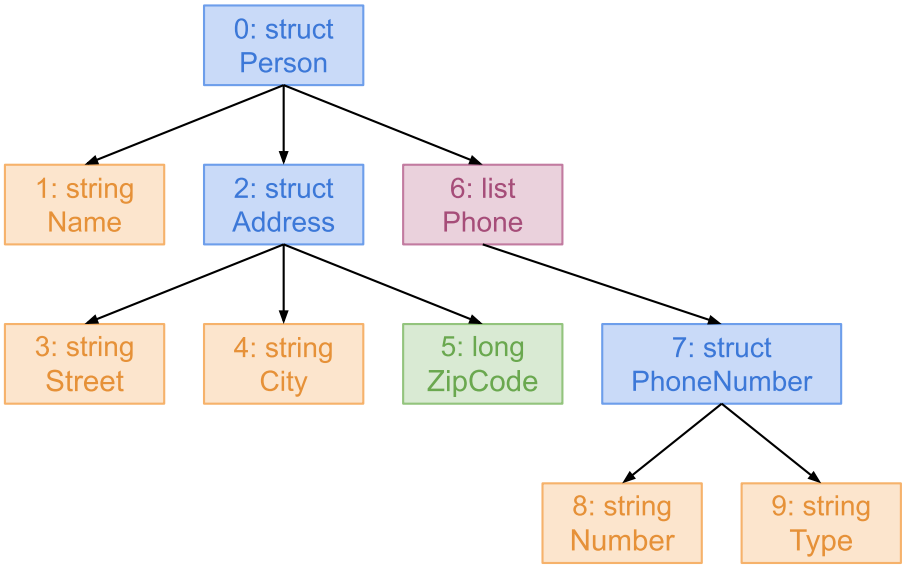

Abbildung 3.2.: ORC Typbaum des Beispieldatensatzes aus Listing 3.1

Eine ORC Datei besteht aus einem Postscript, Footer und einem oder mehreren Stripes. Das Postscript enthält alle notwendige Informationen, wie zum Beispiel Kompression, um den Rest der Datei einlesen zu können. Ebenso enthält der Footer notwendige Metadaten, wie zum Beispiel Stripe Positionen und den Aufbau der Datensätze (Typ, Spalten-Id). Ein Stripe besteht aus mehreren Indizes, einer Untermenge der Datensätzen und einem Stripe Footer. Die Indizes und Datensätze werden wiederum in Spalten aufgeteilt. Diese Aufteilung erfolgt hingegen nur im übertragenen Sinne, da sich ein Stripe eigentlich nur aus mehreren Streams zusammensetzt. Ein Stream kann als Block von kodierten Daten verstanden werden und wird genau einer Spalte zugeordnet, jedoch kann sich eine Spalte aus mehrere Streams zusammensetzten. Daher ändert sich abhängig vom Typen der Spalten die Anzahl und Art der zugeordneten Streams. So benötigt eine obligatorisch Ganzzahlspalte lediglich einen $DATA$ Stream, währenddessen eine optionale Binärspalte gleich drei Streams ($PRESENT$, $DATA$, $LENGTH$) benötigt. Dieser modulare Ansatz erlaubt das Verwenden von Projection-Pushdown, da nur benötigte Streams eingelesen werden können. Die Abbildung 3.3 zeigt den vereinfachten Aufbau einer ORC-Datei.

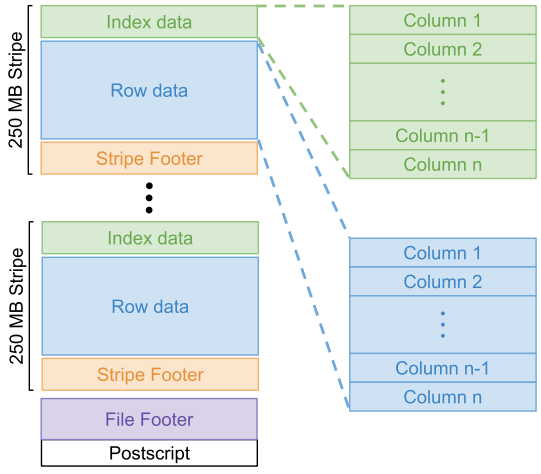

Abbildung 3.3.: Vereinfachter Aufbau einer ORC-Datei

Die Indizes bestehen aus statistischen Kennwerten, welche in der Regel alle 10.000 Datensätze angelegt werden. Demzufolge hängt die Art und Anzahl der Kennwerte vom Typen der Spalte ab. So werden zum Beispiel für Zahlen der minimale und maximale Wert sowie die Summe aller Werte abgespeichert. Da die Kennwerte sehr allgemein gefasst sind, wurden in späteren Version Bloom Filter unterstützt, diese erlauben ein noch genaueres Filtern der Daten. Ziel der Indizes ist es Filter-Pushdown anzubieten, welcher das gezielte Überspringen von für die Anwendung irrelevanter Daten erlaubt.

ORC verwendet eine Vielzahl von Methodiken und Algorithmen, um den benötigten Speicherplatz je Datensatz so gering wie möglich zu halten. So werden unter anderem bis auf das Postscript alle Daten in Blöcke aufgeteilt und komprimiert, wenn die Komprimierung einen Verbesserung in der Größe des Blocks hervorrufen würde. Darüber hinaus werden die eigentlichen Daten der Streams mit diversen Kodierungen möglichst dicht gepackt. Weiterhin können Wörterbücher verwendet werden, um die eigentlichen Daten innerhalb einer Spalte ohne Duplikate zu speichern. Außerdem verfügt ORC über die Möglichkeit die Spalten zu verschlüsseln und anschließend wieder zu entschlüsseln.

4. Konzept

Dieser Abschnitt befasst sich mit allgemeinen konzeptionellen Entscheidungen, welche im Laufe der Arbeit getroffen wurden. Des Weiteren werden allgemeine Anforderungen an die zu implementierenden Speicherformate gestellt. Zum Schluss folgt noch ein kurzer Vergleich zwischen CSV und den neuen Formaten.

4.1. Warum Parquet statt ORC?

Im folgenden werden Parquet und ORC verglichen und erklärt warum Parquet statt ORC für die Arbeit verwendet wurde. Beide unterstützen in etwa die selben Features und sollten ähnliche Laufzeitverbesserungen erzielen. Darunter zählen unter anderem Filter- und Projection-Pushdown, welche für die Arbeit erforderlich sind. Der wesentliche Unterschied zwischen den beiden Formaten ist, wie die Spalten kodiert werden, daher könnte das Datenvolumen für den selben Datensatz zwischen den Formaten variieren. Der Hauptgrund warum Parquet verwendet wird ist, dass Parquet speziell für das Hadoop Ökosystem entwickelt wurde, währenddessen ORC für Apache Hive [18] entwickelt und optimiert wurde. Daher war es naheliegend Parquet für Flink zu verwenden, da Flink das MapReduce Modell von Hadoop direkt unterstützt.

4.2. CSV

Um die Anforderungen an die neue Implementation zu bestimmen und um einen besseren Vergleich zwischen dem aktuellen CSV Speicherformat und den neuen Formaten zu erlauben, wird als erstes das CSV Speicherformat und dessen Aufbau genauer erklärt. CSV oder Comma-separated values ist ein einfaches, in RFC 4180 [19] grundlegend definiertes, Textformat, welches Daten in Form einer Tabelle speichert. Dafür werden die einzelnen Datensätze für gewöhnlich mit Zeilenumbrüchen und die Spalten innerhalb eines Datensatzes mit Begrenzungszeichen, wie zum Beispiel dem Komma, getrennt. Darüber hinaus muss jede Zeile beziehungsweise jeder Datensatz gleich viele Spalten enthalten. Da alle Daten in reiner Textform gespeichert werden, ist das Format für einen Menschen lesbar und kann mit einem einfachen Texteditor bearbeitet werden. Außerdem ist zu beachten, dass verschachtelte Daten nicht mit reinem CSV gespeichert werden können.

Damit genauer verstanden werden kann, wie EPGM und TPGM Elemente in CSV abgespeichert werden, müssen jedoch erst die Elemente und ihre Felder definiert werden. Jedem Feld wird ein Typ zugeordnet, welcher bestimmt wie das Feld kodiert wird. Darüber hinaus handelt es sich bei manchen Feldern um verschachtelte Felder, welche beim Kodieren in ihre Einzelteile zerlegt werden müssen und anschließend beim Dekodieren wieder zusammengesetzt werden müssen. Um einen bessern Überblick über alle Typen von Feldern zu verschaffen, werden im Folgenden alle Typen inklusive ihres groben Aufbaus und Eigenschaften erläutert.

GradoopId ist ein zwölf Byte langer eindeutiger Identifikator, welcher jedem Element in Gradoop zugeordnet wird. Er wird in CSV als seine Hexadezimaldarstellung kodiert, also eine 24 Zeichen lange Zeichenkette.

String ist eine beliebige Zeichenkette und wird primär für die Beschriftungen und Eigenschaftsschlüssel der Elemente verwendet. Die Zeichenketten bekommen alle Zeichen escaped, welche normalerweise als Begrenzungszeichen in Gradoop CSV verwendet werden, wie zum Beispiel , ; | und werden dann ohne weitere Modifikationen gespeichert.

Properties sind Mengen von Schlüssel-Wert-Paaren, welche für den Schlüssel immer einen *String* verwenden und für den Wert eine Vielzahl von verschiedenen Typen annehmen können, wie beispielsweise *Boolean, Int, List, Map* und so weiter. Die Schlüssel sowie der Typ des Werts werden in einer separaten Metadaten Datei abgespeichert um duplizierte Daten zu verringern und die Lesegeschwindigkeit zu erhöhen. Die Werte hingegen werden abhängig vom Typen des Werts in Zeichenketten umgewandelt. Zum Beispiel werden Zahlen (*Int*) als ihre Dezimalrepräsentation dargestellt, während dessen komplexere Typen wie eine Liste (*List*) erst die Werte der Liste in Zeichenketten umwandeln und diese anschließend in die folgende Form bringen: $[wert_1, wert_2, ..., wert_n]$. Im Anschluss werden alle Zeichenketten der Werte in der selben Reihenfolge wie in den Metadaten aneinandergereiht und durch ein Begrenzungszeichen (|) voneinander getrennt.

GradoopIdSet ist eine Menge von *GradoopIds* und beschreibt die Zugehörigkeit von einem Element zu beliebig vielen anderen Elementen. Die *GradoopIds* können wie oben beschrieben als Hexadezimaldarstellung kodiert werden und werden daraufhin in die folgende Form gebracht: $[id_1, id_2, ..., id_n]$.

TimeInterval ist ein Zeitintervall, welcher sowohl offen als auch geschlossen sein kann. In Gradoop werden die Grenzen der Zeitintervalle durch Unix Zeitstempel repräsentiert. Eine offene untere Grenze ist als -2^{63} und eine offene obere Grenze als $2^{63} - 1$ definiert. Das bedeutet, eine Grenze des Intervalls kann immer durch eine 64 Bit Ganzzahl repräsentiert werden. Daher setzt sich ein *TimeInterval* aus zwei 64 Bit Ganzzahlen zusammen. Die beiden Ganzzahlen werden als ihre Dezimalrepräsentation kodiert und in die folgende Form gebracht: $(from, to)$.

Im Anschluss gilt es zu klären, aus welchen Feldern sich die Elemente zusammensetzten. Logische Graphen werden in Gradoop lediglich durch einen sogenannten *GraphHead* repräsentiert. Dabei handelt es sich um eine nur auf das Nötigste reduzierte Version des logischen Graphen. So wird unter anderem die Zugehörigkeit der Knoten und Kanten zu einem logischen Graphen in den jeweiligen Knoten und Kanten selbst hinterlegt, jedoch nicht im *GraphHead*. Demzufolge setzt sich ein *GraphHead* nur aus einer Id (*GradoopId*), einer Beschriftung (*String*) und den Eigenschaften (*Properties*) zusammen. *Vertices* (Knoten) besitzen die selben Felder wie ein *GraphHead* und ein zusätzliches Feld vom Typen *GradoopIdSet* (GraphIds), welches einen Knoten mehreren *GraphHeads* (logischen Graphen) zuordnet. Im Übrigen besitzen *Edges* (Kanten) ebenfalls die gleichen Felder wie ein *Vertex* und zwei zusätzliche *GradoopId* (SourceId/TargetId) Felder. Die beiden Felder Verweisen jeweils auf den Quell- und Zielknoten auf den sich die *Edge* bezieht. Sollten die Elemente temporal (TPGM) sein, so besitzen alle von ihnen zwei zusätzliche Felder vom Typen *TimeInterval*. Die beiden Intervalle repräsentieren den Gültigkeitszeitraum und die Transaktionszeit des jeweiligen Elements.

Wie eben bereits erwähnt werden für die *Properties* Metadaten in einer separaten Datei (*metadata.csv*) abgespeichert. Diese Metadaten bestimmen in Abhängigkeit vom Label und dem Typen des Elements (*GraphHead, Vertex, Edge*), in welcher Reihenfolge die Werte der *Properties* abgespeichert sind, welchen Schlüssel sie haben und welcher Typ ihnen zugeordnet wird. Des Weiteren werden für verschachtelte Werte, wie *Map* oder *List* weitere Metadaten über deren Kinder-Werte abgespeichert. Daher müssen die Metadaten vor dem Speichern der Elemente ermittelt werden und vor dem Lesen eingelesen werden. Listing 4.1 zeigt wie Metadaten für *Properties* kodiert werden. Zum Beispiel haben *GraphHeads* mit dem Label *city* eine *name* Property mit dem Typen *String*.

```
1 g;city;name:string
2 v;person;name:string,yearOfBirth:int,city:string,phone:list:string
3 e;knows;since:localdate
```

Listing 4.1: Metadaten im CSV Speicherformat

Für jedes Feld eines EPGM und TPGM Elements wird eine Spalte in der CSV angelegt. Die Spalten sind durch Semikola getrennt. Die Werte der Felder werden wie oben beschrieben kodiert mit Ausnahme von den beiden *TimeIntervals* eines TPGM Elements. Diese werden in eine einzige Spalte gepackt und haben das Format: $(tx - from, tx - to), (val - from, val - to)$. Die Listings 4.2, 4.3 und 4.4 zeigen die Kodierung aller TPGM Elemente für die Metadaten aus Listing 4.1. Die kodierten Elemente entsprechen ungefähr den Elementen aus Abbildung 2.2 inklusive weiterer Properties, um die Kodierung besser zu veranschaulichen.

```
1 000000000000000000000001;city;Leipzig;(1640995200000,9223372036854775807)
      ,(−30136924800000,9223372036854775807)
2 000000000000000000000002;city;Dresden;(1609459200000,9223372036854775807)
      ,(−25150867200000,9223372036854775807)
```

Listing 4.2: Temporale *GraphHeads* im CSV Speicherformat für die Metadaten aus Listing 4.1

```
1 000000000000000000000100;[000000000000000000000001];person;Alice|2002|Leipzig
      |[01749464308,015228817386];(1640995200000,9223372036854775807)
      ,(1009843200000,9223372036854775807)
2 000000000000000000000101;[000000000000000000000001];person;Bob|1991|Leipzig
      |[01729980752];(1609459200000,9223372036854775807),(662688000000,9223372036854775807)
3 000000000000000000000102;[000000000000000000000002];person;Carol|1983|Dresden
      |[];(1609459200000,9223372036854775807),(410227200000,9223372036854775807)
```

Listing 4.3: Temporale *Vertices* im CSV Speicherformat für die Metadaten aus Listing 4.1

```
1 000000000000000000010001;[000000000000000000000001];000000000000000000000100;000000000000000000000101;
      knows;2022;(1640995200000,9223372036854775807),(1640995200000,9223372036854775807)
2 000000000000000000010002;[000000000000000000000001];000000000000000000000101;000000000000000000000100;
      knows;2022;(1640995200000,9223372036854775807),(1640995200000,9223372036854775807)
3 000000000000000000010003;[];000000000000000000000101;000000000000000000000102;knows
      ;2017;(1609459200000,9223372036854775807),(1483228800000,9223372036854775807)
```

Listing 4.4: Temporale *Edges* im CSV Speicherformat für die Metadaten aus Listing 4.1

4.3. Parquet

Parquet bietet verschiedene Möglichkeiten für das Kodieren von Daten an, dazu zählen unter anderem eine Implementation für Protocol Buffers (Protobuf) [15] sowie die Option einen komplett individuellen Serialisierer zu implementieren. Ein Vorteil von Parquet mit Protobuf gegenüber einer individuellen Implementation ist, dass der Programmcode einfacher zu warten und zu verstehen ist, da Konzepte wie die Schemavalidierung/-generierung und Projection-Pushdown vor dem Nutzer versteckt werden. Ein Nachteil wiederum ist die schlechtere Effizienz mit der Daten kodiert und dekodiert werden können. Das liegt vor allem daran, dass viele temporäre Objekte erzeugt werden, was vor allem einen Garbage-Collector-Overhead erzeugt. Daher wird sowohl eine mit Protobuf als auch eine komplett individuelle auf Gradoop zugeschnittene Implementation bereit gestellt und getestet.

4.3.1. Parquet mit Protobuf

Für die Implementation mit Protobuf muss lediglich für jedes Element ein Protobuf Daten-Objekt definiert werden, welches die selben Felder hat. Die Felder können jedoch nur Protobuf kompatible Typen annehmen. Deshalb müssen die Felder der Elemente in Protobuf kompatible Typen umgewandelt werden. Dafür wird eine Map-Funktion benötigt, welche die Elemente in die Protobuf Daten-Objekte umwandelt und umgekehrt. Der Rest, wie Schemaerzeugung für Parquet und das Kodieren/Dekodieren der Daten wird komplett vom Parquet Protobuf Modul übernommen. Im Folgenden werden alle Typen und ihre Kodierung für Protobuf beschrieben.

GradoopId wird in Protobuf als ein Byte-Array (*Binary*) kodiert.

String können direkt ohne weitere Anpassungen verwendet werden.

Properties benötigen keine extra Metadaten, wie mit CSV, da Werte inklusive ihres Typen direkt binär kodiert werden können. Dafür werden Methoden von Gradoop verwendet, welche in der Lage sind den Wert unabhängig vom Typen binär zu kodieren und dekodieren. Diese Methoden können ebenfalls verschachtelte Typen, wie eine *Map* oder *List* in ein einziges Byte-Array umwandeln. Die *Properties* können daher direkt als *Map* mit *String* Schlüsseln und Byte-Array (*Binary*) Werten von Protobuf kodiert werden.

GradoopIdSet wird in Protobuf als ein wiederholendes Byte-Array Feld kodiert und stellt damit eine List von *GradoopIds* dar, welche in einen Byte-Array (*Binary*) kodiert wurden.

TimeInterval kann als zwei 64 Bit Ganzzahlen kodiert werden. Um weiteren Objektinstanzen mit Protobuf zu vermeiden wurden die beiden *TimeIntervals* direkt unverschachtelt auf vier Felder aufgeteilt (*txFrom*, *txTo*, *valFrom*, *valTo*).

4.3.2. Parquet ohne Protobuf

Für die Implementation ohne Protobuf müssen ebenfalls alle Typen von Feldern in Parquet kompatible Typen beziehungsweise Werte umgewandelt werden. Daher wird im Folgenden grob beschrieben, wie die einzelnen Typen im Vergleich zu CSV kodiert werden. Zu beachten ist, das Parquet das Speichern von verschachtelten Typen unterstützt und es daher möglich ist Typen wie *List* und *Map* direkt mit Parquet abzuspeichern, solange alle Unter-Typen richtig kodiert und definiert werden, also zum Beispiel die Schlüssel und Werte einer *Map*.

GradoopId kann als ein Byte-Array mit der festen Länge zwölf kodiert werden, da er ein zwölf Byte langer unveränderbarer Identifikator ist.

String wird als Byte-Array beliebiger Länge kodiert. Traditionell lassen sich Zeichenketten über einen Zeichensatz wie zum Beispiel *ASCII* oder *UTF-8* binär (zu einem Byte-Array) kodieren. Da Parquet direkt UTF-8 unterstützt, werden Zeichenketten binär mit UTF-8 kodiert.

Properties benötigen ebenfalls keine extra Metadaten, weil die Werte wie bei der Implementation mit Protobuf kodiert werden. Die Schlüssel werden als *String* wie oben beschrieben kodiert. Die Paare selbst werden in Parquet als *Map* abgespeichert.

GradoopIdSet kann in Parquet als *List* von *GradoopIds* kodiert werden, also eine Liste von Byte-Arrays mit fester Länge. Die *GradoopIds* können wie oben beschrieben binär kodiert werden.

TimeInterval kann als zwei 64 Bit Ganzzahlen kodiert werden. Jeder der beiden *TimeIntervals* bekommt ein eigenes Feld im Parquet Schema, damit das Schema möglichst deckungsgleich mit dem Aufbau der Elemente aus Gradoop ist.

Wie mit CSV wird in Parquet für jedes Feld eines Elements eine Spalte beziehungsweise ein Feld im Schema angelegt. Da sich Parquet um das eigentliche Speichern der Spalten kümmert, muss sich die Implementation nur um das Kodieren der Werte in einen Parquet kompatiblen Typen kümmern.

4.4. Vergleich CSV mit Parquet

Ein wesentlich Unterschied zwischen Parquet und CSV ist, dass CSV Metadaten für die *Properties* braucht. Das heißt, dass CSV extra Schritte braucht, um Daten zu speichern und lesen, da immer erst die Metadaten ermittelt werden müssen. Außerdem ist es durch die Metadaten nur möglich strikt typisierte verschachtelte *Properties* zu speichern, obwohl das EPGM keine solche Restriktion definiert. Das bedeutetet zum Beispiel, dass CSV keinen Listen mit mehreren Typen abspeichern kann. Darüber hinaus kann CSV von Menschen gelesen und bearbeitet werden, währenddessen Parquet nur maschinenlesbar und nicht mehr modifizierbar ist. Die Unmodifizierbarkeit ergibt sich aus der Tatsache, dass Parquet während dem Schreibvorgang alle nötigen Metadaten für den Parquet Format sammelt und diese am Ende der Datei speichert. Sollte sich die Position der Daten durch das Bearbeiten verschieben, sind die Metadaten nicht mehr gültig und müssten neu ermittelt werden. Sonst ist der grundlegende Aufbau recht ähnlich, beide definieren beispielsweise fast genau die selben Spalten bis auf die verketteten *TimerIntervals* in CSV. Die beiden Parquet Implementation

sollten in etwa die gleiche Schnittstelle, wie CSV definieren, das heißt beide benötigen lediglich den Ordner in dem sich ein Datensatz befinden soll, um diesen zu beschreiben oder zu lesen.

5. Implementierung

In diesem Kapitel wird die Implementation von Parquet in Gradoop behandelt. Als Erstes werden die von Gradoop und Parquet verwendeten Schnittstellen genau erklärt. Anschließend wird die Implementation mit Protobuf und abschließend die Implementation ohne Protobuf behandelt.

5.1. Verwendete Schnittstellen

Gradoop ist in mehrere Module eingeteilt, so kümmert sich unter anderem das *gradoop-flink* Modul primär um das EPGM Modell, währenddessen das *gradoop-temporal* Modul das *gradoop-flink* Module um das TPGM erweitert. Für das Lesen von *GraphCollections* (Graphmengen) und *LogicalGraphs* (logische Graphen) wird in *gradoop-flink* das *DataSource* Interface verwendet und für das Schreiben das *DataSink* Interface. Ebenfalls definiert *gradoop-temporal* das *TemporalDataSource* Interface für das Lesen von *TemporalGraphCollections* (temporalen Graphmengen) und *TemporalGraphs* (temporale logische Graphen) sowie das *TemporalDataSink* Interface für das Schreiben. Sowohl der *DataSink* als auch der *TemporalDataSink* unterstützten das optionale Überschreiben von bereits existierenden Dateien. Gradoop stellt verschiedene Layouts zur Verfügung, welche definieren, wie ein *LogicalGraph* oder eine *GraphCollection* aufgebaut ist. Das Einfachste von ihnen ist das GVE-Layout, welches aus drei *DataSets* (Flink's Datenströmen) für *GraphHead*, *Vertex* (Knoten), *Edge* (Kanten) besteht. Für einen *LogicalGraph* besteht das *GraphHead DataSet* nur aus einem einzigen *GraphHead* und für *GraphCollections* aus mehreren. Unabhängig von der Art des Layouts definiert jedes Layout Funktionen, um genau drei *DataSets* für *GraphHeads, Vertices* und *Edges* zu bekommen. Die temporalen Versionen haben genau die gleichen Layouts, verwenden jedoch statt dem *GraphHead*, *Vertex* und *Edge* den *TemporalGraphHead*, *TemporalVertex* und *TemporalEdge*. Die temporalen Elemente bilden eine direkte Erweiterung der EPGM-Elemente durch das Hinzufügen der zwei *TimeIntervals* für den Gültigkeitszeitraum und die Transaktionszeit des jeweiligen Elements. Da in Gradoop alle für diese Arbeit relevanten Elemente des EPGM eine Untermenge der Element des TPGM bilden, wird im Folgenden primär auf die Implementation für das TPGM eingegangen.

Alle Einstellungen und Metadaten bezüglich eines Hadoop Speicherformats werden auf einem gewöhnlichem Hadoop *Job* konfiguriert, welcher im weiteren Verlauf der Anwendung nicht ausgeführt wird, sondern lediglich für das Halten des *Configuration* Objekts und eventueller *Credentials* verantwortlich ist. Da es sich bei Parquet um einen Hadoop Speicherformat handelt, welcher speziell für Hadoop MapReduce entwickelt wurde, kann dieser nicht einfach in Flink eingebunden werden. Daher bietet Flink Wrapper (*HadoopInputFormat*, *HadoopOutputFormat*) für das Verwenden von Hadoop MapReduce Speicherformaten an, welche einen beliebigen Hadoop *InputFormat/OutputFormat* in einen Flink *InputFormat/OutputFormat* umwandeln. Ein Flink *InputFormat* kann einfach über die *ExecutionEnvironment::createInput* Methode als *DataSet* eingelesen werden. Währendessen ein Flink *OutputFormat* mit Hilfe der *DataSet::output* Methode ein *DataSet* speichern kann. Die Wrapper selbst sind sehr minimalistisch, da Hadoop's Ein- und Ausgabeformate eine nahezu gleiche Schnittstelle wie jene aus Flink verwenden. Der einzige wesentliche Unterschied ist,

dass Hadoop standardmäßig mit Schlüssel-Wert-Paaren arbeitet, welche für diesen Anwendungsfall unnötig sind, da mit Parquet nur der Wert definiert ist. Daher wurden abgewandelte Versionen der Wrapper (*HadoopValueInputFormat/HadoopValueOutputFormat*) erstellt, welche den Wert direkt zurückgeben oder entgegennehmen. Dies spart im späteren Verlauf eine Map-Funktion, welche normalerweise den Wert in ein Schlüssel-Wert-Paar (*Tuple*) umwandelt und umgekehrt.

Um mit Parquet Daten einzulesen wird zunächst ein *ReadSupport* benötigt. Dieser hat die Aufgabe einen *RecordMaterializer* für die gegeben Metadaten und das Schema zu generieren, sowie den *ReadContext* zu initialisieren. Der *ReadContext* legt das angeforderte Schema (Projection-Pushdown), sowie für den *RecordMaterializer* benötigte Metadaten fest. Das angeforderte Schema kann als Baum verstanden werden, in dem jedem Knoten (Feld) ein *Converter* zugewiesen wird. Ein Blatt bekommt einen *PrimitiveConverter*, welche einfache Datentypen dekodiert, währenddessen alle anderen (verschachtelten) Knoten einen *GroupConverter* bekommen, welcher für alle Unterknoten die *Converter* definiert. Der *RecordMaterializer* definiert den Wurzel *GroupConverter* für das Schema und gibt die gelesenen und dekodierten Daten als Objekt zurück. Filter, wie jene aus Filter-Pushdown, werden intern von Parquet verarbeitet und ausgewertet, können jedoch aber selbstständig definiert und in der *Configuration* gesetzt werden.

Für das Schreiben von Daten mit Parquet wird ein *WriteSupport* benötigt, welcher optionale Metadaten und das zu schreibende Schema definiert. Außerdem hat er die Aufgabe Datenobjekte zu kodieren. Dies geschieht mit Hilfe des *RecordConsumer*. Der *RecordConsumer* enthält eine Vielzahl von Methoden, um den Feldern Werte zu zuweisen. Listing 5.1 zeigt beispielhafte Methodenaufrufe an den *RecordConsumer*. Jedes Feld wird über einen eindeutigen Namen und seinen Index im Schema identifiziert und adressiert. Die *startMessage/stopMessage*, *startGroup/endGroup* und *startField/endField* Methoden definieren den allgemeinen Rahmen des zu schreibenden Objekts, also welche Felder definiert sind. Die Platzhalter-Methode *addValue*, welche sinnbildlich für alle Wertzuweisungs-Methoden steht, kann verwendeten werden, um dem aktuelle adressierten Feld primitive Datentypen (*boolean, int, long, float, double, byte[]*) zu zuweisen. Eine weitere Besonderheit ist, dass der Standard-*ParquetOutputFormat* nicht bereits existierende Dateien überschreibt. Diese Funktionalität wird jedoch von Gradoop angeboten, daher wurde eine Erweiterung der Klasse (*ParquetOutputFormatWithMode*) angelegt, welche diese Funktionalität unterstützt. Zusätzlich wurde eine neue Option in der *Configuration* angelegt, welche für bereits existierende Dateien bestimmt, ob diese überschrieben werden sollen oder ob ein Fehler geworfen wird, wenn die Datei bereits existiert.

```
startMessage()
  startField("X", 0)
    addValue(1)
    addValue(2)
  endField("X", 0)
  startField("Y", 1)
    startGroup()
      startField("Z", 0)
        addValue(3)
      endField("Z", 0)
    endGroup()
  endField("Y", 1)
endMessage()
```

Listing 5.1: Beispielhafte Methodenaufrufen an den *RecordConsumer*

```
{
  X: [1, 2]
  Y: {
    Z: 3
  }
}
```

Listing 5.2: Resultat zu den Aufrufen aus Listing 5.1

5.2. Implementation mit Protobuf

Für die Implementation mit Protobuf wurde *proto2* verwendet, da *proto3* den *required* Modifikator entfernt hat, dieser jedoch für ein genauer definiertes Schema benötigt wird. Für jedes Datenobjekt in der *proto* Definitionsdatei wird eine *Message* und *Builder* Klasse generiert. Eine *Message* Instanz ist unveränderbar (immutable) und hat alle obligatorischen Felder mit Werten initialisiert. Der *Builder* ist ein Fluent-Interface[3], welches für die Instanziierung von *Message* Objekten verwendet wird. Die Felder des *Builders* und der *Builder* selbst sind zurücksetzbar, was ihn wiederverwendbar macht.

Das Parquet Protobuf Modul definiert seinen eigenen *WriteSupport* (*ProtoWriteSupport*) und *ReadSupport* (*ProtoReadSupport*). Der *ProtoWriteSupport* unterstützt sowohl das Schreiben von *Message* als auch *Builder* Objekten. Außerdem ist er in der Lage das Protobuf Schema in ein Parquet Schema zu übersetzen. Daher muss der *ProtoWriteSupport* und die *Message* Klasse lediglich in der *Configuration* konfiguriert werden, um Parquet mit Protobuf Datenobjekt verwenden zu können. Da Gradoop seine eigenen POJO Klassen verwendet müssen die POJOs erst in *Message* oder *Builder* Objekte umgewandelt werden. Um möglichst wenig Objekte erzeugen zu müssen, sollte der *Builder* statt der *Message* verwendet werden, da dieser wiederverwendbar ist. Daher wurde für jedes POJO ein eigener *Message* Typ im *proto* Schema angelegt. Darüber hinaus wurde für jedes POJO *Message* Paar eine Map-Funktion angelegt, welche das POJO in den jeweiligen *Builder* der *Message* übersetzt. Die Map-Funktionen erzeugen sofern möglich nur eine *Builder* Instanz je Map-Funktions Instanz. Das heißt alle POJO Instanzen werden über die Map-Funktion in *Builder* Instanzen übersetzt, welche mit dem *ProtoWriteSupport* gespeichert werden. Eine weitere Besonderheit ist, dass weder die *Message* noch die *Builder* Klasse die POJO-Regeln von Flink erfüllt, was dazu führt, dass diese nicht von Flink serialisiert werden können. Außerdem genügt nicht der Standard-Kryo-Serializer um ein *Message* oder *Builder* Objekt zu serialisieren, da diese teilweise aus unveränderbaren Objekten bestehen, welche während der Dekodierung Fehler werfen. Infolgedessen wurde ein benutzerdefinierter Kryo-Serializer für *Message* als auch *Builder* Objekte angelegt. Dieser verwendet die Lese- und Schreibfunktionen von Protobuf, um diese als Byte Array zu kodieren und dekodieren. Des Weiteren wurde die *writeSpecsCompliant* Option auf *true* gesetzt, damit die

[3] Ist ein Schnittstelle (Interface), welche über Verkettung/Aneinanderreihung von Methoden verwendbar ist.

Protobuf Parquet Implementation *Map* und *repeated* Felder mit den jeweiligen korrektem Parquet Typen (*Map, List*) kodiert. Zudem gab es während der Evaluation Schwierigkeiten mit dem Protobuf Package. Dies könnte unter anderem daran liegen, dass eine JAR-Datei im *lib* Verzeichnis von Flink ebenfalls eine andere Version von Protobuf enthält. Daher wurde für die Evaluation das Protobuf Package mit Hilfe des Maven Shade Plugins in das *shaded.gradoop* Package verschoben. Listing 5.3 zeigt einen Ausschnitt der *proto* Definitionsdatei (*TemporalEdge*). Dieser ist in etwa repräsentativ für alle anderen Elemente, da die Felder der *TemporalEdge* eine Obermenge der Felder von jedem anderen Element bilden.

```
1  message TemporalEdge {
2    required bytes id = 1;
3    required string label = 2;
4    map<string, bytes> properties = 3;
5    repeated bytes graphIds = 4;
6    required bytes sourceId = 5;
7    required bytes targetId = 6;
8    required int64 txFrom = 7;
9    required int64 txTo = 8;
10   required int64 valFrom = 9;
11   required int64 valTo = 10;
12 }
```

Listing 5.3: Protobuf Schema einer *TemporalEdge*

Um Protobuf Datenobjekte aus Parquet auslesen zu können, muss der *ProtoReadSupport* und die *Message* Klasse in der *Configuration* gesetzt werden. Der *ProtoReadSupport* unterstützt Projection-Pushdown über das Definieren eines partiellen Schemas in der proto IDL in der *Configuration*. Sollte keine Projektion definiert sein, wird das gesamte Schema der konfigurierten *Message* Klasse verwendet. Standardmäßig gibt der *ProtoReadSupport* die gelesen Daten als *Builder* Objekte zurück, um Objekte zu sparen. Anschließend werden diese in einer Map-Funktion in Gradoop POJOs übersetzt.

Für den Schreib- als auch Leseprozess wird ein Pfad festgelegt. Von diesem ausgehend wird für jeden Elementypen eine eigene Parquet Datei (Ordner) angelegt oder ausgelesen. So liegen Elemente vom Typen *GraphHead* in *graphs.parquet*, *Vertex* in *vertices.parquet* und *Edge* in *edges.parquet*. Das selbe gilt auch für die folgende Implementation ohne Protobuf.

5.3. Implementation ohne Protobuf

Für den *WriteSupport* ohne Protobuf (*GradoopWriteSupport*) wurde zunächst das *ParquetWriter* Interface angelegt. Es kümmert sich um die Generierung des zu schreibenden Schemas und das Schreiben der POJOs in den *RecordConsumer*. Zusätzlich wurde eine neue Option in die *Configration* aufgenommen, um den *ParquetWriter* für den *GradoopWriteSupport* zu setzten. Der *GradoopWriteSupport* holt sich zunächst den *ParquetWriter* aus der *Configuration* und setzt das Schema für den Schreibvorgang mit Hilfe des *ParquetWriters*. Anschließend wird jeder Schreibaufruf an den *WriteSupport* und der *RecordConsumer* an den *ParquetWriter* übergeben und von diesem verarbeitet. Damit duplizierter Programmkode vermieden wird und um den Schreibvorgang zu vereinheitlichen, wurden verschiedene abstrakte Basis-*ParquetWriter* erstellt. Die Abbildung 5.1

zeigt alle *ParquetWriter* und ihre Abhängigkeiten (Vererbung) untereinander. Der *ElementParquetWriter* bildet die Basisklasse für alle anderen *ParquetWriter* und kümmert sich um gemeinsame Felder aller Elemente, wie die Id, Label und Properties. Außerdem generiert er für diese Felder das Schema, welches von anderen Unterklassen erweitert werden kann. Der *GraphElementParquetWriter* erweitert alle Funktionen des *ElementParquetWriter* um das GraphIds Feld. Die eigentlichen *ParquetWriter* für die EPGM Elemente basieren auf diesen beiden Klassen. So verwendet der *GraphHeadParquetWriter* den *ElementParquetWriter* als Überklasse und der *VertexParquetWriter* und *EdgeParquetWriter* verwenden den *GraphElementParquetWriter*. Diese fügen bis auf Ausnahme von dem *EdgeParquetWriter* keine weiteren Felder hinzu. Der *EdgeParquetWriter* fügt das SourceId und TargetId Felder hinzu. Da Java keine Mehrfachvererbung unterstützt, wurde die temporale Erweiterung des TPGMs in das *TemporalParquetWriter* Interface in *default* Funktionen ausgelagert. Das Interface kann das Schema um das TransactionTime und ValidTime Feld erweitern, sowie die beide Felder in den *RecordConsumer* schreiben. Die temporale Erweiterung wird am Ende des EPGM Schemas angefügt, damit es theoretisch möglich ist einen TPGM Datensatz als EPGM Datensatz einzulesen, indem die zusätzlichen Feldern einfach ignoriert werden können. Daher basieren die *TemporalGraphHeadParquetWriter*, *TemporalVertexParquetWriter* und *TemporalEdgeParquetWriter* auf ihren EPGM Gegenstücke und dem *TemporalParquetWriter* und dienen primär als Glue Code[4].

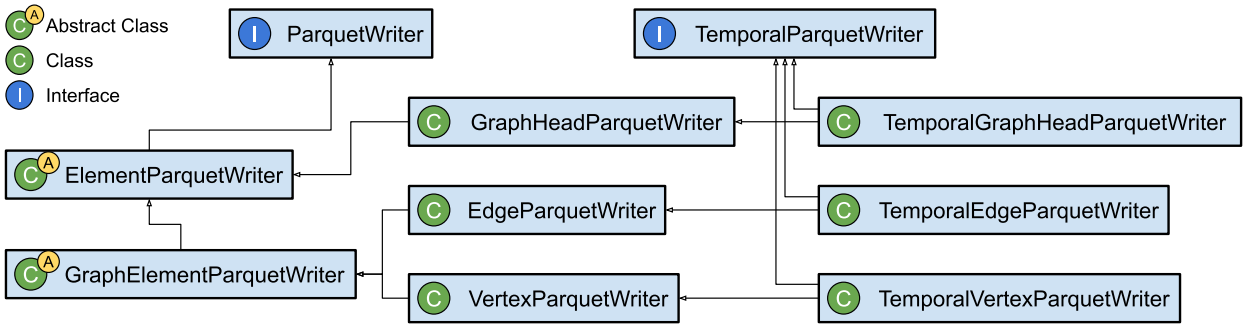

Abbildung 5.1.: Vereinfachtes Klassendiagramm der *ParquetWriter*

In Parquet muss jedem Feld oder Gruppe genau einer der drei Modifikator: *required*, *optional* oder *repeated* zugeordnet werden. Diese haben den selben Effekt wie die gleichnamigen Modifikatoren in Protobuf. Außerdem wird jedem Feld ein *PrimitiveTypeName* zugeordnet. Zur Verfügung stehen: *BOOLEAN, INT32, INT64, INT96, FLOAT, DOUBLE, FIXED_LEN_BYTE_ARRAY, BINARY*. Darüber hinaus kann für jedes Feld mit dem *PrimitiveTypeName FIXED_LEN_BYTE_ARRAY* eine Länge für das Byte Array festgelegt werden. Neben dem *PrimitiveTypeName* kann jedem Feld noch eine optionale *LogicalTypeAnnotation* zugeordnet werden. Diese ordnet dem Feld weitere Metadaten zu, damit es genauer von Lesern interpretiert werden kann. So können Zeichenketten zum Beispiel mit dem *PrimitiveTypeName BINARY* gespeichert werden und zusätzlich noch mit der *LogicalTypeAnnotation String* versehen werden, um zu zeigen, dass es sich um eine mit UTF-8 kodierte Zeichenkette handelt. Außerdem bietet Parquet Schemavorlagen für *List* und *Map* Felder an. Ein Parquet *List* Feld besteht immer aus einer wiederholenden Gruppe *list*, welche das eigentliche Element-Feld (*element*) enthält. Ein Parquet *Map* Feld hingegen setzt sich immer aus

[4]Ist Programmcode, dessen einzige Aufgabe es ist verschiedene Teile eines Programms miteinander zu verknüpfen, welche sonst untereinander inkompatibel wären

einer wiederholenden Gruppe *key_value* zusammen, welche wiederum aus dem *key* (Schlüssel) und *value* (Wert) Feld besteht. Sowohl das *element* als auch die *key* und *value* Felder können beliebig konfiguriert werden.

```
message temporal_edge {
    // Element
    required FIXED_LEN_BYTE_ARRAY id = 0 [len=12];
    required BINARY label = 1 [typ=string];
    optional GROUP properties = 2 [typ=map] {
        repeated GROUP key_value = 0 {
            required BINARY key = 0 [typ=string];
            required BINARY value = 1;
        }
    }
    // GraphElement
    required GROUP graph_ids = 3 [typ=list] {
        repeated GROUP list = 0 {
            required FIXED_LEN_BYTE_ARRAY element = 0 [len=12];
        }
    }
    // Edge
    required FIXED_LEN_BYTE_ARRAY source_id = 4 [len=12];
    required FIXED_LEN_BYTE_ARRAY target_id = 5 [len=12];
    // TemporalEdge
    required GROUP transaction_time = 6 {
        required INT64 from = 0 [typ=timestamp, utc=true, unit=millis];
        required INT64 to = 1 [typ=timestamp, utc=true, unit=millis];
    }
    required GROUP valid_time = 7 {
        required INT64 from = 0 [typ=timestamp, utc=true, unit=millis];
        required INT64 to = 1 [typ=timestamp, utc=true, unit=millis];
    }
}
```

Listing 5.4: Schema einer *TemporalEdge*

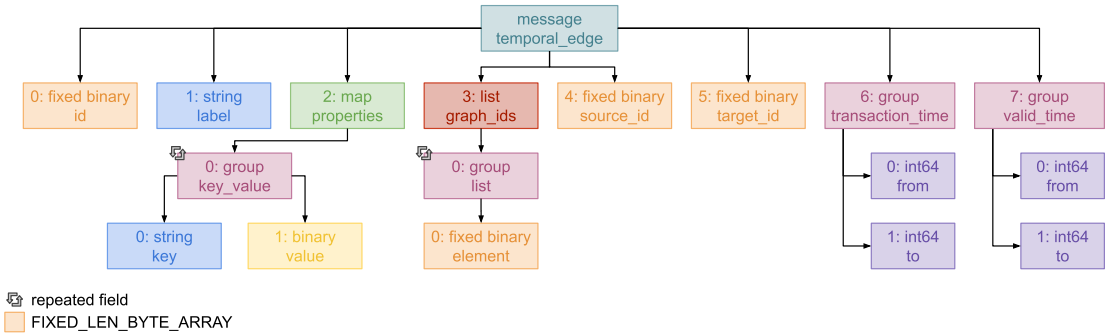

Abbildung 5.2.: Vereinfachtes Diagramm zum Schema aus Listing 5.4

Das Listing 5.4 und die Abbildung 5.2 zeigen das für die individuelle Implementation verwendete Schema der *TemporalEdge*. Alle *GradoopIds* werden als *FIXED_LEN_BYTE_ARRAY* der Länge zwölf kodiert, Zeichenketten als *BINARY* Feld mit der *string Annotation* und Properties als Parquet *Map*, wobei das *value* Feld als *BINARY* Feld kodiert wird. Dies geschieht mit Hilfe der internen Funktionen von Gradoop zum Kodieren und Dekodieren von *PropertyValues*. Des Weiteren werden *GradoopIdSets* als Parquet *List* von *GradoopIds* kodiert und ein *TimeInterval* wird als obligatorische Gruppe von zwei Ganzzahlen mit der *timestamp Annotation* kodiert. Alle Felder

sind obligatorisch mit Ausnahme des Properties Felds, welches nur gesetzt wird, wenn das POJO Properties zugewiesen hat.

Für den *ReadSupport* ohne Protobuf (*GradoopReadSupport*) wurde die *ParquetReader* Klasse angelegt und eine neue Option in der *Configuration* aufgenommen, welche den *ParquetReader* für den *GradoopReadSupport* definiert. Der *GradoopReadSupport* verwendet aktuell immer das Schema der Parquet Datei als zu lesendes Schema und übergibt dies dem *ParquetReader*, welche wiederum an den *GradoopRecordMaterializer* weiter gegeben wird. Die abstrakte *ParquetReader* Klasse ist ein *GroupConverter* und fungiert als Root *GroupConverter* für den *RecordMaterializer*. Daher definiert der *GradoopRecordMaterializer* lediglich den *ParquetReader* als Root *GroupConverter* und gibt das gelesene Datenobjekt des *ParquetReaders* weiter. Die *ParquetReader* Klasse enthält *Converter* Klassen für jeden Feldtypen (*GradoopIdConverter, StringConverter, PropertyValueConverter, LongValueConverter, KeyValueConverter, PropertiesConverter, ListElementConverter, GradoopIdSetConverter und TimeIntervalConverter*). Jeder Converter nimmt einen Java *Consumer* entgegen, welcher den gelesenen Wert des Felds zurück gibt. Einige der *Converter* sind *GroupConverter* und basieren daher auf anderen *Convertern*, wie zum Beispiel der *PropertiesConverter*, welcher den *KeyValueConverter* verwendet, um die Schlüssel-Wert-Paare einzulesen. Jede Unterklasse des *ParquetReaders* definiert in der *initializeConverters* Methode welches Feld welchen *Converter* verwenden sollte und welches Feld im POJO vom *Consumer* gesetzt werden soll. Der *ParquetReader* selbst nimmt ein Parquet Schema im Konstruktor entgegen und legt für die im Schema und in der *initializeConverters* Methode definierten Felder die *Converter* an. Anschließend wird für jeden Leseaufruf ein neues POJO angelegt, durch die *Converter* befüllt und an den *GradoopRecordMaterializer* weitergegeben. Dieser Ansatz erlaubt später das Verwenden von Projection-Pushdown, indem ein reduziertes Schema in der *Configuration* angelegt und an den Konstruktor weiter gegeben wird, da der Konstruktor nur für definierte Felder *Converter* anlegt.

Die neuen *DataSource*, *DataSink*, *TemporalDataSource* und *TemporalDataSink* Implementationen mit Parquet benötigen lediglich den Dateipfad des Datensatzes, um ihn zu beschreiben oder einzulesen. Das heißt, dass die Parquet Implementation genau sowie die CSV Implementation in Gradoop eingebunden werden können. Das bedeutetet auch, dass ein CSV Datensatz ohne große Probleme in einen Parquet Datensatz umgewandelt werden kann, indem er mit der *CSVDataSource* eingelesen wird und dann mit dem *ParquetDataSink* oder *ProtobufParquetDataSink* abgespeichert wird.

6. Evaluation

6.1. Ziel der Evaluation

Ziel der Evaluation ist es verschiedene Operatoren und Datensätze mit den neuen Parquet Implementationen zu testen und mit der bereits bestehenden CSV Implementation zu vergleichen. Die erhaltenen Daten können dann anschließend ausgewertet werden, um Entscheidungen darüber treffen zu können, welche Implementation wann genutzt werden sollte. Zu diesem Zweck wurde eine leicht abgewandelte Testsuite von verschieden Benchmarks verwendet, welche bereits in einer vorheriger Arbeit zum TPGM von Rost et al. [20] verwendet wurde.

6.2. Grundlagen

Die gesamte Evaluation wurde auf einem Cluster mit 16 Workern ausgeführt. Jeder Worker hat einen Intel Xeon E5-2430 mit 12 Threads und 6 Kernen auf 2.5GHz, 48 GB RAM sowie zwei 4 TB SATA Festplatten und läuft auf openSUSE 13.2 mit Hadoop v2.6.0 und Flink v1.9.0. Auf jedem Worker läuft ein Flink Task Manager mit 6 Taskslots und 40 GB Heap-Speicher. Alle Worker sind untereinander mit 1 GBit Ethernet verbunden.

Datensatz	SF	$\|V\|$	$\|E\|$	Größe (CSV)
LDBC	1	3,3 M	17,9 M	4,2 GB
LDBC	10	30,4 M	180,4 M	42,3 GB
LDBC	100	282,6 M	1,77 B	421,9 GB
CitiBike	100	1174	97,5 M	22,6 GB

Tabelle 6.1.: Eckdaten der verwendeten Datensätze

Für die Evaluation wurden zwei verschiedenen Datensätze verwendet: LDBC und CitiBike. Bei LDBC handelt es sich um einen synthetischen Datensatz mit fester Struktur, welcher den allgemeinen Strukturen eines sozialen Netzwerks ähnelt. CitiBike hingegen ist ein realer Datensatz welcher den Fahrradverleih seit 2013 bis 2021 für Citi Bike[5] beschreibt. Darüber hinaus wird die Größe der beiden Datensätze mit dem Skalierungsfaktor (SF) beschrieben, welcher linear mit dem vervielfachen der Daten wächst. Für LDBC wurden drei verschiedene Größe verwendet, wobei der größte Datensatz (SF=100) aus circa 282,6 Million Knoten und 1,77 Milliarden Kanten besteht und eine Größe von rund 421,9 GB hat. CitiBike hingegen wurde nur mit einer einzigen Größe getestet und umfasst circa 1174 Knoten und 97,5 Million Kanten und eine Größe von rund 22,6 GB. Weitere Eckdaten zu den verwendeten Datensätzen können der Tabelle 6.1 entnommen werden.

[5] `https://citibikenyc.com/`

6.3. Technische Evaluation

Alle Benchmarks wurden mindestens dreimal mit jeweils dem CSV, Protobuf Parquet und Parquet Speicherformat sowie den verschiedenen LDBC Datensätzen ausgeführt. Jeder Benchmark folgt im Wesentlichen dem selben Ablauf: Als erstes wird der ausgewählte Datensatz mit dem auszuwertenden Format eingelesen. Anschließend wird der Operator ausgeführt und das Ergebnis im auszuwertenden Format abgespeichert. Die einzige Ausnahme bildet der CitiBike Analytical (CBA) Benchmark, welcher nur auf dem CitiBike Datensatz ausgeführt wird und anstelle des einzelnen Operators eine Reihe von Operatoren zur Analyse des Datensatzes verwendet. Alle temporalen Operatoren verwenden immer den Gültigkeitszeitraum (*val-from*, *val-to*) der TPGM-Elemente, sofern nicht ausdrücklich anders angegeben. Um einen besseren Überblick über die Benchmarks zu bekommen, werden im Folgenden alle Benchmarks etwas genauer erklärt.

Der **Snapshot** Benchmark verwendet den *Snapshot* Operator in Kombination mit dem *AsOf* Prädikat und dem Zeitstempel 1308000000000, welcher circa 30% der Knoten und Kanten selektieren sollte. Der *verify* Operator wurde bewusst nicht verwendet um einen starken I/O Engpass zu erzeugen, damit die Lese- und Schreibgeschwindigkeit der Implementationen besser evaluiert werden kann.

Der **Difference** Benchmark verwendet den *Difference* Operator in Kombination mit zwei *AsOf* Prädikaten, welche mit einer Reihe von Zeitintervallen ausgeführte werden. Für jeden Intervall wird die untere Grenze des Intervalls als Zeitstempel für das erste Prädikat und die obere Grenze für das zweite Prädikat verwendet. Die verwendeten Zeitintervalle sind: $[1287000000000, 1298000000000]$, $[1287000000000, 1308000000000]$, $[1298000000000, 1298000000000]$ und $[1298000000000, 1308000000000]$. Ebenfalls wurde hier der *verify* Operator nicht verwendet.

Der **Grouping** Benchmark gruppiert sowohl Knoten als auch Kanten nach ihrer Beschriftung und der Woche der unteren Grenze des Gültigkeitsintervalls. Für die Aggregierung wird die *Count* Funktion verwendet, welche jeweils für Knoten und Kanten die Anzahl dieser, in ihrer Gruppe, ermittelt. Daraus ergeben sich anschließend Knoten und Kanten, welche die Anzahl von Elementen in Abhängigkeit von der Beschriftung und dem Erstellungszeitpunkt abbilden.

Der **Pattern Matching** Benchmark wertet die temporale Abfrage aus Listing 6.2 aus. Die Abfrage gibt alle Nutzer aus, welche einen Kommentar zu einem Beitrag in einem festen Zeitraum geliket haben.

Der **CitiBike Analytical (CBA)** Benchmark setzt sich aus einer Vielzahl von Operatoren zusammen. Das CBA Programm kann verwendet werden um herauszufinden, in welchen Teilen von New York City die meisten und wenigsten Räder für mindesten 40 beziehungsweise 90 Minuten gemietet werden, wie lange die Durchschnittsmietdauer ist und wie sich das Mietverhalten in Abhängigkeit von der Jahreszeit verändert. Listing 6.1 zeigt den beispielhaften Arbeitsablauf des Benchmarks. Zu Beginn wird der CitiBike Datensatz auf die Zeit zwischen 2017 und 2019 mit dem *Snapshot* Operator beschränkt (Zeile 2-3). Anschließend wird jedem Knoten eine *cellId* zugewiesen, welche aus der geographischen Position des Knotens berechnet wird (Zeile 4-5). Daraufhin wird eine temporale Abfrage von Zeile 6 - 14 ausgeführt. Diese selektiert alle Paare von aufeinanderfolgenden

Trips, welche mindestens 40 beziehungsweise 90 Minuten lang sind, in Abhängigkeit vom **X** und wo der Erste der beiden Trips in der Zelle 2883 startet und der zweite Trip an der selben Station beginnt wo der Erste beendet wurde. Im Anschluss werden die aus der Abfrage resultierenden Graphen in einen einzige Graphen zusammengeführt (Zeile 15). Danach werden die Knoten über die Eigenschaftsschlüssel *name* und *cellId* (Zeile 17), sowie die Kanten über den Monat der unteren Grenze des Gültigkeitsintervalls (Zeile 19) gruppiert. Für jede daraus resultierende Kante wird die Anzahl der zugehörigen Trips sowie die durchschnittliche Dauer dieser ermittelt (Zeile 20-22). Abschließend werden alle Kanten, welche keinen zugehörigen Trip haben entfernt (Zeile 23).

```
1  outGraph = citiBikeGraph
2    .snapshot(
3      Overlaps('2017-01-01', '2019-01-01'))
4    .transform(
5      v -> v['cellId'] = getGridCellId(v))
6    .query(
7      "MATCH (v1:Station)-[t1:Trip]->(v2:Station)
8            (v2)-[t2:Trip]->(v3:Station)
9       WHERE v1.cellId = 2883 AND
10           v2.id != v1.id AND
11           v2.id != v3.id AND
12           t1.val.precedes(t2.val) AND
13           t1.val.lengthAtLeast(Minutes(X)) AND
14           t2.val.lengthAtLeast(Minutes(X))")
15   .reduce(g, h => g.combine(h))
16   .groupBy(
17     [label(),prop('name'),prop('cellId')],
18     (),
19     [label(),timestamp(val-from, MONTH)],
20     (superEdge, edges =>
21       superEdge['count'] = edges.count(),
22       superEdge['avgDur'] = edges.avgDur()))
23   .subgraph(e => e['count'] > 1)
```

Listing 6.1: CitiBike Analytical Programm (X ist die Mindestdauer des Trips)

```
1  MATCH (p:person)-[l:likes]->(c:comment),
2        (c)-[r:replyOf]->(po:post)
3  WHERE l.val_from.after(Timestamp(2012-06-01)) AND
4        l.val_from.before(Timestamp(2012-06-02)) AND
5        c.val_from.after(Timestamp(2012-05-30)) AND
6        c.val_from.before(Timestamp(2012-06-02)) AND
7        po.val_from.after(Timestamp(2012-05-30)) AND
8        po.val_from.before(Timestamp(2012-06-02))
```

Listing 6.2: Pattern Matching Benchmark GDL

6.3.1. Scalability

Genauere Daten zur Laufzeit in Abhängigkeit zur Datenmenge können den Abbildungen 6.1 (CSV), 6.2 (Parquet) und 6.3 (Protobuf) entnommen werden. Weiterhin können den Abbildungen 6.4 (CSV), 6.5 (Parquet) und 6.6 (Protobuf) genauere Daten zur Laufzeit in Abhängigkeit zur Anzahl der Worker für LDBC.100 und CBA entnommen werden.

Die Laufzeit fast aller Benchmarks konnte mit den neuen Implementationen im Vergleich zu CSV mit Ausnahme von CBA unabhängig von der Größe des Datensatzes verbessert werden. Der genaue Grund warum CBA langsamer geworden ist, ist nicht ganz klar. Es könnte eventuell daran liegen, dass der Datensatz allgemein kleiner ist als LDBC oder an der hohen Datenschiefe zwischen Knoten und Kanten. Diese Theorie wird weiterhin von den Laufzeiten der Protobuf Implementation auf kleinen Datensätzen (LDBC.1), wie für die Difference und Grouping Benchmarks, untermauert, siehe Tabelle 6.2. Des Weiteren ist der Tabelle zu entnehmen, dass allgemein die Laufzeit auf größeren Datensätzen stärker reduziert werden kann, als auf kleineren. So benötigt der Snapshot Benchmark mit 16 Workern für LDBC.1 circa 72,5% der Zeit die CSV dafür benötigte, für LDBC.100 jedoch wurden nur 23,4% der Zeit von CSV benötigt. Diese Beobachtung kann für alle

Benchmarks auf den LDBC Datensätzen gemacht werden. Das bedeutet, je größer der Datensatz um so größer wird der prozentuale Abstand der Laufzeit zwischen Parquet und CSV. Außerdem scheint die reine Parquet Implementation fast immer besser als die Protobuf Implementation auf 16 Workern abzuschneiden.

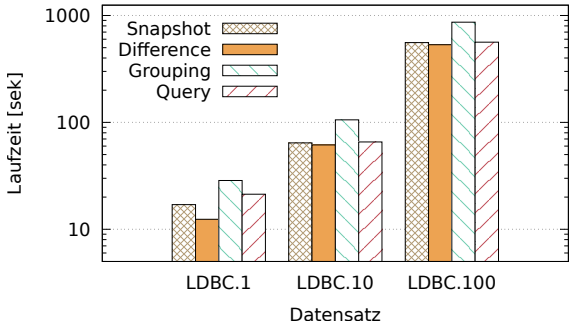

Abbildung 6.1.: Laufzeit CSV mit 16 Workern (niedriger ist besser)

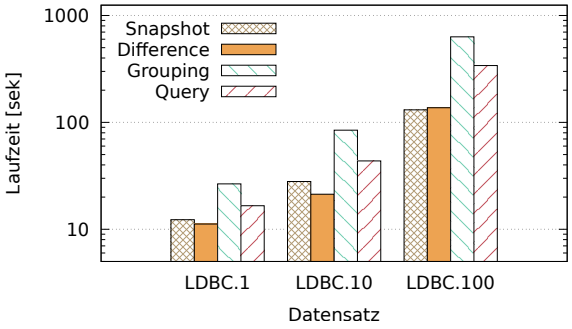

Abbildung 6.2.: Laufzeit Parquet mit 16 Workern (niedriger ist besser)

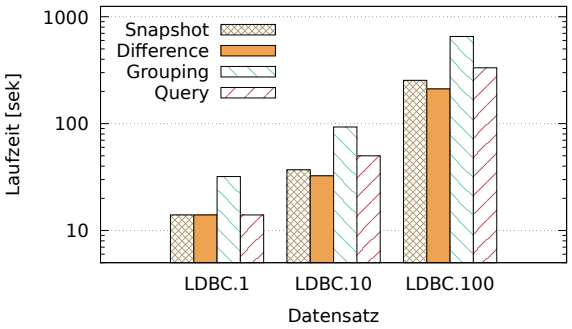

Abbildung 6.3.: Laufzeit Protobuf mit 16 Workern (niedriger ist besser)

Implementation	Datensatz	Snapshot	Difference	Grouping	Query
Parquet	LDBC.1	72,5%	82,2%	93,0%	78.1%
	LDBC.10	43,5%	34,5%	80,1%	66.5%
	LDBC.100	23,4%	25,7%	72,9%	60.4%
Protobuf	LDBC.1	82,4%	112,7%	111,8%	65,7%
	LDBC.10	57,5%	52,8%	88,0%	76,2%
	LDBC.100	45,5%	39,6%	75,6%	59,1%

Tabelle 6.2.: Laufzeit relative zu CSV in % mit 16 Workern (niedriger ist besser)

Eine weitere interessante Beobachtung ist, dass Protobuf in I/O-lasting Benchmarks, wie Snapshot und Difference, schlechter abschneidet als die reine Parquet Implementation. Andererseits ist Protobuf in den CBA und Grouping Benchmark fast immer schneller als die reine Parquet Implementation, siehe Tabelle 6.3. So benötigt die reine Parquet Implementation für CBA40 auf einem Worker bis zu 142,2% der Zeit die CSV dafür benötigt, währenddessen Protobuf dafür nur 129,4% der Zeit von CSV dafür benötigt. Das Protobuf manchmal schneller und manchmal langsamer als

die reine Parquet Implementation ist, könnte daran liegen, dass die Daten ungleich verteilt sind und in bestimmten Benchmarks diese Verteilung zu einer Beschleunigung führt. Es wäre denkbar, dass im Grouping Benchmark die Daten, die eine Gruppe bilden häufiger auf dem selben Worker sind wie mit Parquet ohne Protobuf. Das würde auch erklären, warum der Snapshot und Difference Benchmark davon nicht betroffen sind, da die Position der Daten auf dem Cluster für diese Benchmarks keine Rolle spielt. In zukünftigen Arbeiten könnte dieses Verhalten genauer untersucht werden.

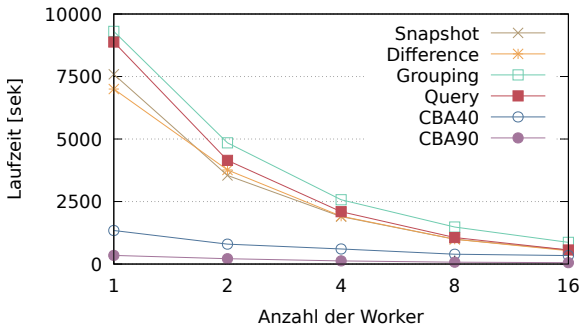

Abbildung 6.4.: Laufzeit CSV auf LDBC.100 (niedriger ist besser)

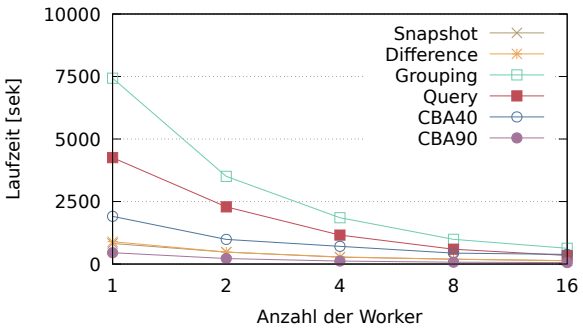

Abbildung 6.5.: Laufzeit Parquet auf LDBC.100 (niedriger ist besser)

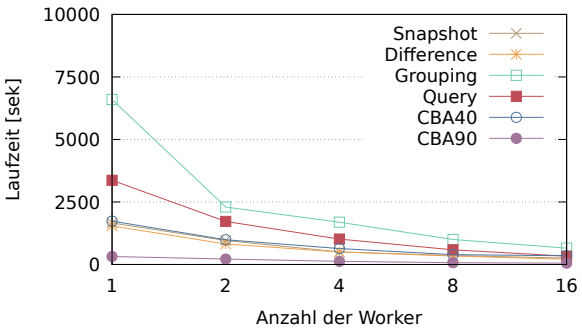

Abbildung 6.6.: Laufzeit Protobuf auf LDBC.100 (niedriger ist besser)

Implementation	Worker	Snapshot	Difference	Grouping	Query	CBA40	CBA90
Parquet	1	10,9%	12,8%	79,9%	47,8%	142,2%	131,1%
	2	13,5%	12,7%	72,3%	55,1%	123,9%	104,4%
	4	14,5%	14,9%	75,0%	55,4%	116,9%	93,7%
	8	20,0%	19,2%	66,7%	55,7%	110,9%	99,1%
	16	23,5%	25,7%	73,0%	60,5%	114,9%	104,3%
Protobuf	1	21,9%	22%	70,9%	37,9%	129,4%	91,7%
	2	27,0%	21,8%	47,4%	41,5%	124,3%	102,4%
	4	26,3%	26,2%	65,8%	48,5%	105,4%	100,0%
	8	35,7%	34,3%	67,5%	55,3%	99,9%	96,5%
	16	45,5%	39,6%	75,6%	59,1%	103,5%	97,1%

Tabelle 6.3.: Laufzeit relative zu CSV in % auf LDBC.100 (niedriger ist besser)

6.3.2. Speedup

Sowohl die Protobuf als auch die reine Parquet Implementation skalieren unabhängig vom verwendeten Benchmark nahezu gleich, was den Speedup der beiden Implementation in etwa gleich macht. Dies liegt vor allem daran, dass beide Implementationen eine ähnliche Verbesserung der Laufzeit in Bezug auf I/O erreichen. Im Vergleich zur CSV Implementation ist der Speedup jedoch gesunken. Das kann damit erklärt werden, dass der I/O Bottleneck mit den neuen Implementationen bereits auf einem einzigen Worker stark reduziert werden konnte. Dadurch verändert sich der Ausgangspunkt (Laufzeit) für den Speedup. Der CPU Bottleneck und die hohen Kommunikationskosten, welche bei einer hohen Anzahl von Workern entstehen, werden dadurch jedoch nicht gesenkt, was dazu führt, dass der Speedup geringer ausfällt. Besonders gut lässt sich das bei I/O-lastigen Benchmarks, wie dem Snapshot oder Difference Benchmark, beobachten. Hier gelingt es der CSV Implementation einen nahe zu linearen Speedup zu erreichen. Die höhere Anzahl der Worker beschleunigt jedoch nicht den eigentlichen Operator des Benchmarks, sondern verringert den I/O Bottleneck. Daher skaliert der Snapshot und Difference Benchmark mit den neuen Implementation schon bei 4 Workern nicht mehr linear. Währenddessen der Query und Grouping Benchmark bis zu 8 Workern noch linear skaliert und dann wegen der hohen Kommunikationskosten ausgebremst wird. Die einzige Ausnahme bilden die beiden CBA Benchmarks, welche insgesamt etwas langsamer geworden sind und daher einen höheren Speedup als mit CSV erzielen. Genauere Daten zum Speedup für LDBC.100 und CBA können den Abbildungen 6.7 (CSV), 6.8 (Parquet) und 6.9 (Protobuf) entnommen werden.

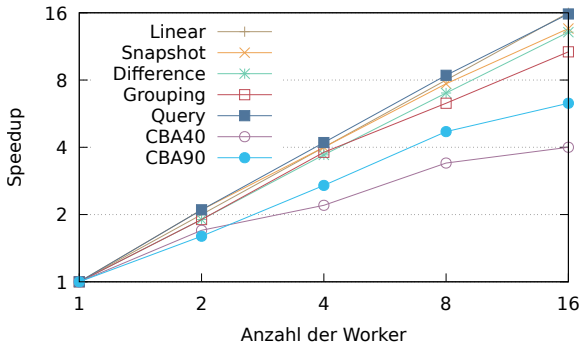

Abbildung 6.7.: Speedup CSV für LDBC.100 (höher ist besser)

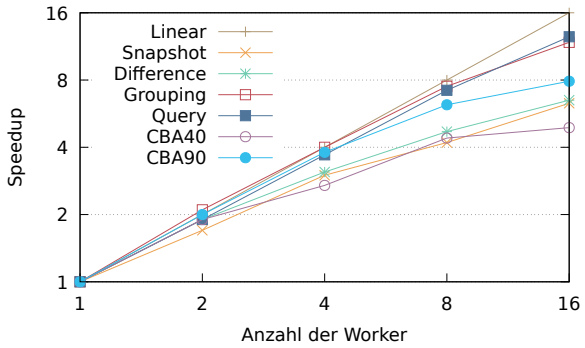

Abbildung 6.8.: Speedup Parquet für LDBC.100 (höher ist besser)

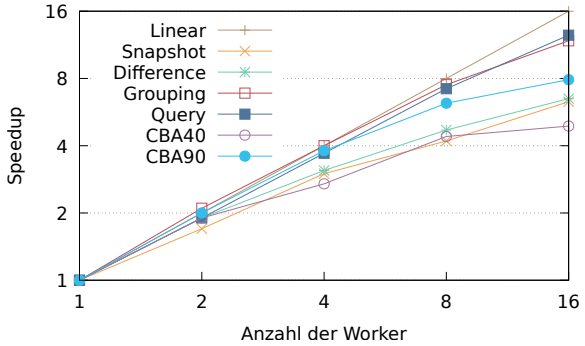

Abbildung 6.9.: Speedup Protobuf für LDBC.100 (höher ist besser)

6.3.3. Datenvolumen

Die beiden neuen Implementationen konnten im Durchschnitt das Datenvolumen der Datensätze im Vergleich zu CSV vierteln. So konnte mit der reinen Parquet Implementation die Größe der LDBC Datensätze unabhängig vom SF auf circa 19-20% der originalen Größe (CSV) verkleinert werden. Ebenfalls wurde der Citibike Datensatz unabhängig von vom SF auf circa 30-32% der originalen Größe reduziert. Die Protobuf Implementation sind im Durchschnitt weitere 2-4% kleiner als die reine Parquet Implementation im Vergleich zu CSV. Da der SF fast keinen Einfluss auf das Größenverhältnis zwischen den Parquet Implementation und CSV hat, ist davon auszugehen, dass die Datenersparnisse in etwa linear zum Datenvolumen skalieren. Genauere Daten zum Datenvolumen können der Abbildung 6.10 und der Tabelle 6.4 entnommen werden.

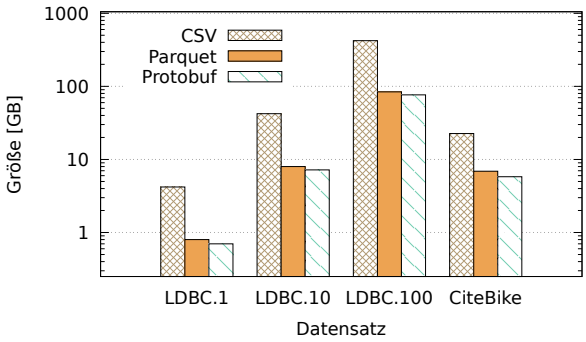

Abbildung 6.10.: Datenvolumen
(niedriger ist besser)

Datensatz	SF	Größe	Rel. zu CSV
LDBC	1	0,8 GB	18,4%
	10	8,0 GB	18,9%
	100	84,2 GB	20,0%
CitiBike	1	0,07 GB	32,0%
	10	0,7 GB	31,7%
	100	6,9 GB	30,5%

Tabelle 6.4.: Datenvolumen Parquet ohne Protobuf
(niedriger ist besser)

6.4. Zusammenfassung

Zusammenfassend kann man sagen, das die neuen Implementationen in Bezug auf die allgemeine Laufzeit im Normalfall besser abschneiden als die CSV Implementation. Die wenigen Ausreißer, wie das CBA Benchmark, müssten genauer untersucht werden um festzustellen, warum diese langsamer geworden sind. Der Speedup wurde vor allem für höhere Anzahlen von Workern reduziert, da der I/O Bottleneck stark reduziert werden konnte. Dies führt dazu, dass diverse Benchmarks in der selben Zeit wie mit CSV auf weniger Workern ausgeführt werden können. Des weiteren konnte das Datenvolumen für jeden Datensatz in etwa geviertelt werden. Dabei spielt es keine Rolle, welche der beiden Implementationen gewählt wurde und wie groß der ursprüngliche Datensatz war.

7. Zusammenfassung und Ausblick

Ziel der Arbeit war die Implementation und Evaluation von Parquet in das Gradoop Ökosystem. Außerdem sollte eine Basis für Zukünftige Arbeiten an dem Projekt für Projection- und Filter-Pushdown geschaffen werden. In diesem Abschnitt wird zusammengefasst, ob diese Ziele erfüllt wurden. Des Weiteren wird ein Ausblick auf zukünftige Arbeiten und weitere Fragestellungen gegeben.

Der Parquet Speicherformat wurde gleich in zwei verschiedenen Implementationen in Gradoop erfolgreich eingebunden, die reine Parquet und die Protobuf Implementation. Beide funktionieren ohne große Probleme und unterstützten alle von Gradoop benötigten Features. Außerdem unterstützten beide sowohl Filter-Pushdown, über die Standard-API von Parquet, als auch Projection-Pushdown, welches in beiden Fällen über ein partielles Schema in der Konfiguration des Jobs gesetzt werden könnte. Jedoch kann eine Vielzahl von Optimierungen an der aktuellen Implementation in zukünftigen Arbeiten vorgenommen werden. Dazu zählt unter anderem eine Schemavalidierung, um eine spätere Evolution des Schemas zu erlauben, beziehungsweise um das Projection-Schema mit dem Schema der Datei zu vergleichen und eventuelle Fehler zu werfen. Sowie weitere Optimierungen am aktuellen Programmcode, wie zum Beispiel, dass nur noch Felder mit Werten beschrieben werden, wenn diese nicht den Standardwert des Felds angenommen haben, um das Datenvolumen zu senken und das I/O zu beschleunigen. Weiterhin können Verbesserungen an der Konfiguration des Jobs vorgenommen werden, wie zum Beispiel *BlockSize, PageSize, Compression* und *WriterVersion*. Alle diese Optionen verändern wie Daten gespeichert werden und können abhängig von den konfigurierten Werten verschiedene Ergebnisse liefern. Daher müssten diese in Kombination mit intensiven Tests konfiguriert werden, bis die gewünschte Verbesserung erreicht ist. Ebenfalls Thema für weitere Arbeiten wäre es Apache ORC in Gradoop einzubinden, da es viele Ähnlichkeiten zu Parquet hat und in etwa den selben Funktionsumfang anbietet. Darüber hinaus bietet ORC einen Hadoop MapReduce Adapter an, welcher das Einbinden in Flink vereinfachen sollten.

Allgemein konnte das Datenvolumen aller Datensätze mit beiden Implementationen in etwa auf das Viertel des Datenvolumen von CSV reduziert werden. Außerdem scheint der SF keinen Einfluss auf das Verhältnis zwischen CSV und Parquet für einen bestimmten Datensatz zu haben. So wurde für alle Datensätze in etwa das gleiche Verhältnis des Datenvolumens zwischen CSV und Parquet für jeden SF erreicht. Des Weiteren zeigt die Evaluation, dass beide Implementationen die CSV Implementation in allen Benchmarks bis auf CBA in Bezug auf Laufzeit geschlagen haben. Warum genau CBA als einziges Benchmark langsamer wurde, müsste genauer in zukünftigen Arbeiten analysiert werden. Dafür wäre es denkbar den *Java Flight Recorders* zu verwenden, um das Benchmark genauer zu profilen und eventuelle Optimierungen für den CitiBike Datensatz und das CBA Benchmark sowie für ähnliche Datensätze und Workflows vorzunehmen. Eine weitere Besonderheit, die ebenfalls in zukünftigen Arbeiten erforscht werden könnte, ist, dass in manchen Benchmarks (vor allem I/O-lastige) die reine Parquet Implementation bessere Laufzeiten erzielt, als die Protobuf Implementation, obwohl beide ein ähnliches Schema verwenden. Des Weiteren war zu beobachten, dass vor allem bei großen Datensätzen der prozentuale Abstand der Laufzeit zwischen Parquet und CSV immer weiter wächst, vor allem bei geringer Parallelität. Daher eignen sich

die neuen Implementationen hervorragend, um große Datensätzen auf kleineren Maschinen oder Clustern auszuwerten. Das ist vor allem interessant, für Nutzer von Gradoop, welche bisher größere und teurere Cluster zur Analysen verwendeten, um die Analyse in einer realistischen Zeit abzuschließen. Das gilt jedoch primär für simple Analysen, wie die Evaluation gezeigt hat. Komplexere Benchmarks, wie Grouping, konnten beschleunigt werden, aber im Vergleich zu Benchmarks, wie dem Snapshot, nur in etwa die Leistung verdreifachen, währenddessen der Snapshot Benchmark bis zu einem Zehntel der Zeit von CSV benötigt. Aufgrund dieser Tatsache, wäre es denkbar für alle größeren Datensätze eine der Parquet Implementationen zum Standardformat zu machen.

Literaturverzeichnis

[1] The Apache Software Foundation, "Apache Parquet." `https://parquet.apache.org/`, accessed on 2022-03-01.

[2] ScaDS.AI, "Distributed Graph Analytics with Apache Flink." `https://github.com/dbs-leipzig/gradoop`, 2014.

[3] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache flink™: Stream and batch processing in a single engine," *IEEE Data Eng. Bull.*, vol. 38, pp. 28–38, 2015.

[4] The Apache Software Foundation, "Apache Flink." `https://flink.apache.org/`, accessed on 2022-02-02.

[5] M. Junghanns, A. Petermann, N. Teichmann, K. Gómez, and E. Rahm, "Analyzing extended property graphs with apache flink," in *Proceedings of the 1st ACM SIGMOD Workshop on Network Data Analytics*, NDA '16, (New York, NY, USA), Association for Computing Machinery, 2016.

[6] C. Rost, A. Thor, and E. Rahm, "Temporal graph analysis using gradoop," in *BTW 2019 – Workshopband* (H. Meyer, N. Ritter, A. Thor, D. Nicklas, A. Heuer, and M. Klettke, eds.), pp. 109–118, Gesellschaft für Informatik, Bonn, 2019.

[7] Neo4j, Inc., "Neo4j." `https://neo4j.com/`, accessed on 2022-01-26.

[8] MariaDB Corporation Ab, "MariaDB." `https://mariadb.com/kb/en/system-versioned-tables/`, accessed on 2022-08-03.

[9] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pp. 1–10, 2010.

[10] The Apache Software Foundation, "Apache Hadoop." `https://hadoop.apache.org/`, accessed on 2022-02-02.

[11] E. Begoli, J. Camacho-Rodríguez, J. Hyde, M. J. Mior, and D. Lemire, "Apache calcite: A foundational framework for optimized query processing over heterogeneous data sources," *CoRR*, vol. abs/1802.10233, 2018.

[12] M. Junghanns, A. Petermann, K. Gómez, and E. Rahm, "GRADOOP: scalable graph data management and analytics with hadoop," *CoRR*, vol. abs/1506.00548, 2015.

[13] Neo4j, Inc., "openCypher." `https://opencypher.org/`, accessed on 2022-02-20.

[14] The Apache Software Foundation, "Apache HBase." `https://hbase.apache.org/`, accessed on 2022-02-20.

[15] Google Inc, "Protocol Buffers." `https://developers.google.com/protocol-buffers/`, accessed on 2022-07-07.

[16] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis, "Dremel: Interactive analysis of web-scale datasets," in *Proc. of the 36th Int'l Conf on Very Large Data Bases*, pp. 330–339, 2010.

[17] The Apache Software Foundation, "Apache ORC." `https://orc.apache.org/`, accessed on 2022-03-01.

[18] The Apache Software Foundation, "Apache Hive." `https://hive.apache.org/`, accessed on 2022-08-06.

[19] Y. Shafranovich, "Common format and mime type for comma-separated values (csv) files," RFC 4180, RFC Editor, October 2005. `http://www.rfc-editor.org/rfc/rfc4180.txt`.

[20] C. Rost, K. Gomez, M. Täschner, P. Fritzsche, L. Schons, L. Christ, T. Adameit, M. Junghanns, and E. Rahm, "Distributed temporal graph analytics with GRADOOP," *The VLDB Journal*, may 2021.

Erklärung

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

„Optimierung des Gradoop I/O mittles Columnar Store Apache Parquet“

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

Leipzig, den 08.08.2022

Maurice Eisenblätter