UNIVERSITÄT
LEIPZIG

Institut für Informatik
Fakultät für Mathematik und Informatik
Abteilung Datenbanken

Evaluation von Graph Stream Analyse Pipelines mit Apache Flink

Bachelorarbeit

vorgelegt von:
Maximilian Martin

Matrikelnummer:
3701160

Betreuer:
Prof. Dr. Erhard Rahm
Chrstopher Rost

Inhaltsverzeichnis

Abbildungsverzeichnis

1. Einleitung

1.1. Motivation

Tag für Tag werden Unternehmen von gigantischen Mengen an Daten überschwemmt. Diese Datenmengen, für die sich der Begriff BigData etabliert hat, können, falls sie richtig analysiert werden, einen großen Wert in sich tragen [1]. Doch ihre Auswertung bringt Herausforderungen mit sich.

Stream Processing wurde für Anwendungsfälle entwickelt, in denen die Datenmengen zu groß sind, um zwischengespeichert zu werden, oder eine Echtzeitverarbeitung benötigt wird [2]. Graphen als für den Stream zugrundenliegende Datenstruktur zu wählen, bietet sich immer dann an, wenn sich die wertvollen Informationen statt in den Entitäten, eher in deren Beziehung zueinander befinden [3]. Diese Graph Streams, oder auch dynamische Graphen genannt, finden Anwendung in der Analyse von Verhalten in den sozialen Medien, im Onlinehandel, oder der Kriminologie. Ein Beispiel ist das 2021 entwickelte *Convolutional Neural Network*, welches zur Gerüchteerkennung mit dynamischen Graphen arbeitet, die Gerüchteposts und deren Reaktionen abbilden [4].

Evaluationssoftware schließlich wird benötigt, um die Leistung eines Programms sichtbar zu machen, oder auf noch vorhandene Schwachstellen zu überprüfen. Unter Anwendung verschiedener Parameter und Eingaben wird die Performance anhand von vorher definierten Metriken gemessen [3]. So kann beispielsweise mittels eines Stresstests festgestellt werden, wie viel Entitäten gleichzeitig verarbeitet werden können, bevor es zu einem Eingabestau kommt. Oder man untersucht, bis zu welchem Grad sich eine Erhöhung der Verteilung lohnt, bzw. bis zu welchem Verteilungsgrad, sich die gemessenen Metriken merklich verbessern.

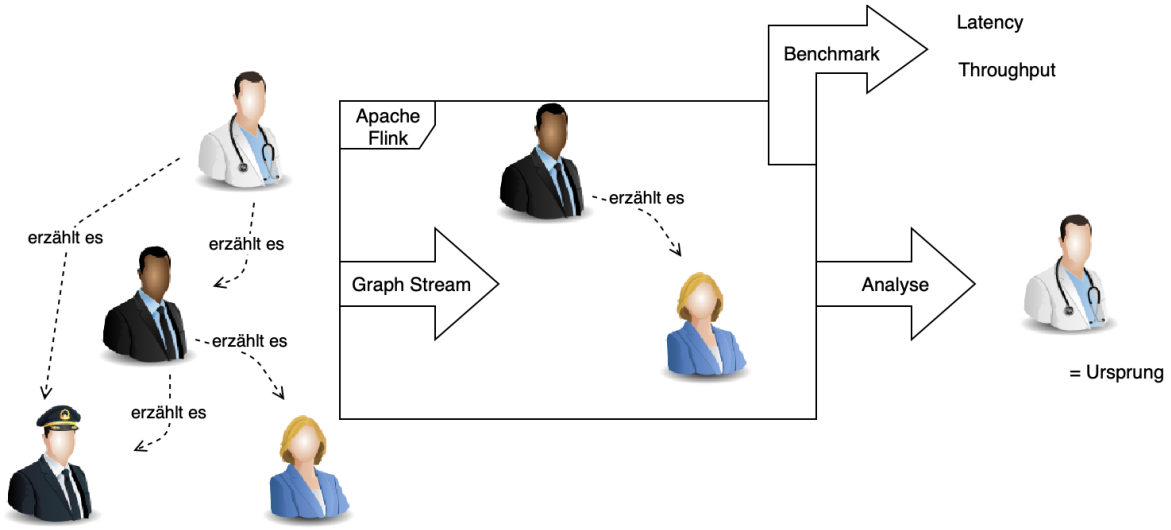

Abbildung 1.1.: Beispiel einer Graph Stream Analyse mit Benchmark

1.2. Zielsetzung

Ziel dieser Arbeit ist die Entwicklung einer Evaluationssoftware für Graph Stream Analyse Operatoren und Pipelines. Von der Anwendung wird erwartet, dass sie auf dem Framework Apache Flink aufbaut, eine hohne Anwendbarkeit aufweist, verschiedene Verteilungsgrade testet und mindestens die Metriken *Latency* und *Throughput*, welche vorher zu definieren sind, misst. Darüber hinaus soll ein Überblick über das Forschungsgebiet der Graph Streams, ihrer Analyse Pipelines, sowie deren Evaluation entstehen.

Angefangen mit einer kurzen Einführung in die Grundlagen des Themas, wird im Kapitel 3 ein kurzer Überblick der Architektur des Streamprozessor-Frameworks Apache Flink gegeben. Anschließend wird auf die Modellierung und Implementierung der entwickelten Software eingegangen. Den Abschluss dieser Arbeit bildet die Auswertung einer Evaluation am Analyse-Operator „Graph Stream Zoomer“ [5].

In dieser Arbeit wird keine Evaluation von Graph Data Management Systemen vorgenommen, da Apache Flink als Framework für diese Evaluationssoftware feststeht. Darüber hinaus wird auch kein eigener Analyse Operator geschrieben.

1.3. Verwandte Arbeiten

In der Forschung besteht eine große Nachfrage nach Software zur Evaluation von Programmen, die sich mit der Verarbeitung von Graph Streams beschäftigen. Das beweist unter anderem der Artikel *A parallel data generator for efficiently generating “realistic” social streams* [6], in dem ein Framework zur Generierung von synthetischen Social Media Streams vorgeschlagen wird. Man soll darüber hinaus den Generator so einstellen können, dass er verschiedene Arten, bzw. Muster von Verhalten der Nutzer simuliert.

Doch es gibt auch weitaus allgemeinere Ansätze, die sogar schon umgesetzt wurden. So ist 2018 *GraphTides* [7] erschienen, ein generisches Framework zur Analyse von Graph Streams verarbeitenden Programmen. GraphTides stellt einen Graph Stream Generator bereit, unterstützt die Analyse von allen Systemen, die einen unbegrenzten Stream an Graph Events über ein Netzwerk empfangen können und es bewertet Systeme mit Hilfe von Systemmetriken wie CPU-Last, Speichernutzung, Netzwerk I/O, sowie internen Metriken wie Durchsatzrate oder Kommunikationslatenz. Im Grunde hat GraphTides schon vieles umgesetzt, was mit dieser Arbeit ebenfalls umgesetzt wurde, mit dem Unterschied, dass die Evaluationssoftware dieser Arbeit auf Apache Flink aufbaut. *GraphTides* setzt bei seinen *Dependecies* dagegen auf das *Popper CLI tool* und Docker [7].

Ein weiteres Papier, das an dieser Stelle erwähnt werden soll, ist *Benchmarking Distributed Stream Data Processing Systems* [8], aus welchem die Definitionen für Latency und Throughput in den Abschnitten 2.4.1 und 2.4.2 stammen. Seine Autoren haben neben diesen Definitionen auch ein Framework zum Testen von Streaming Prozessoren entwickelt. Dabei ist es ihnen gelungen, ihr Benchmarking-Tool größtenteils vom *System Under Test (SUT)* zu trennen. Ermöglicht wurde dies, indem sie den Throughput in den Queues zwischen Datengenerator und SUT maßen, während

die Latency im Sinkoperator des SUT bestimmt wurde. Damit sei eine leichtgewichtete Lösung gefunden worden, ohne das zusätzliche Systeme von Dritten benötigt wurden.

Außerdem haben sie mit ihrem Framework die drei am weitesten verbreiteten Open Source Streaming Prozessoren getestet: *Apache Flink*, *Apache Spark*, sowie *Apache Storm*, wodurch sie jeweils deren Vor- und Nachteile bestimmen konnten. Bei Streams mit *skewed Data* hat Spark am Besten abgeschnitten. Bezogen auf das Handling von Schwankungen in den Eingangsraten von Daten bei Aggregation waren sowohl Flink, als auch Spark besonders robust. Flink war außerdem die geeignetste Wahl, wenn es um die durchschnittliche Latenz ging. Darüber hinaus hatte es auch den größten Throughput bei niedriger Latenz in der Anwendung von großen Windows und allgemein den besseren Throughput im Bezug auf Aggregationen und bei *join-Queries* [8].

Ein weiteres Papier, welches Aufmerksamkeit verdient, ist *Benchmarking Graph Data Management and Processing Systems: A Survey* [3]. Es gibt einen Überblick über die Forschung des Benchmarkings für Graph Managment und Processing Systeme zwischen den Jahren 2005 und 2019. Untersucht wurden dabei 20 verschiedene Benchmarks, darunter Datengeneratoren, Daten-Charakterisierungs-Software und Standardisierungsbemühungen. Auch das bereits genannte Graph-Tides findet Erwähnung.

Festgestellt wurde, dass 88% der Forschungen in ihrem Untersuchungszeitraum in den letzten acht Jahren, also von 2012-2019 stattgefunden haben. Das Thema ist also aktuell und von wachsender Bedeutung. Neben ihrer Studie über bereits bestehende Software, beschäftigten sich die Autoren auch mit der Frage, welche Aufgaben in Zukunft angegangen werden sollten. Dabei stellten sie z. B. fest, dass es noch kaum Benchmarks gibt, die mit realistischen Daten arbeiten, was für sie ein Problem darstellt, da Operatoren nicht darauf getestet werden, wie diese mit stark heterogenen Daten umgehen. Allgemein stellten sie auch ein Defizit an Graph Stream Benchmarks fest. Während die Zahl an entwickelten Analyse Operatoren Jahr für Jahr wächst, wird kaum an Performancetests für diese gearbeitet. Als letztes riefen sie auch dazu auf, neue Metriken ins Benchmarking zu integrieren. Genannt wurden dabei Werte wie Fehlertoleranz, Energieverbrauch und Sicherheit [3].

2. Grundlagen

2.1. Graphen

Zunächst sollen einige der in dieser Arbeit vorkommenden Begriffe definiert werden. Der Erste sei der des Graphen.

Definition 1. *Mathematisch wird ein **Graph** als Paar zweier endlicher Mengen $G = (V, E)$ definiert, wobei V die Menge der Knoten (Vertices) und E die Menge aller Kanten (Edges) bezeichnet [9].*

Je nach Anwendungsfall unterscheidet man zwischen gerichteten und ungerichteten Graphen, wobei bei einem ungerichteten Graphen jede Kante eindeutig durch seine zwei Knoten bestimmt werden kann, während bei einem gerichteten Graphen zusätzlich festgelegt werden muss, welcher Knoten als Start- bzw. Endpunkt gilt. Für zwei verschiedene Knoten *(a,b)* müsste man also zwischen den Kanten *(a,b)* und *(b,a)* unterscheiden. In der Informatik ist ein Graph darüber hinaus auch als Datenmodell zu verstehen, wobei in dieser Arbeit das Modell des Property Graphen Anwendung findet [10].

Definition 2. *Im Fall des **Property Graphen** wird der Graph G um die drei Funktionen ρ, λ und σ erweitert, sodass $G = (V, E, \rho, \lambda, \sigma)$, wobei*

- $\rho : E \to (V, V)$ *die Funktion ist, die jeder Kante $e \in E$ genau zwei Knoten $v_1, v_2 \in V$ zuordnet. Bei gerichteten Graphen nennt man v_1 Start- und v_2 Zielknoten.*
- $\lambda : (V \cup E) \to L$ *eine partielle Funktion ist, die einem Knoten, oder einer Kante ein Label L zuordnet. Aufgrund ihrer Funktionsweise nennt man λ auch Labelfunktion.*
- $\sigma : (V \cup E \times P) \to SET^+(V)$ *eine partielle Funktion ist, die eine Eigenschaft P (Property) einem Set aus Werten $SET^+()$ zuordnet [10].*

Eigenschaften P können sowohl von einfachem, als auch von komplexem Datentyp sein. Typischerweise gehören dazu *Integer*, *Strings*, *Timestamps*, aber auch *Sets*, *Maps* und eigene Objekte, wie beispielsweise das einer maßgeschneiderten ID Klasse. Es gibt Modelle, die es ermöglichen, dass Kanten und Knoten auch über mehrere Labels verfügen können. Da in der Implementierung dieser Arbeit aber darauf verzichtet wurde, wurde auch das Modell des Property Graphen an dieser Stelle angepasst.

2.2. Batch und Stream Processing

Bei der Verarbeitung von Streams wird zwischen zwei Paradigmen unterschieden: Batch- und Stream Processing [11]. Während man beim **Batch Processing**, auch Stapelverarbeitung genannt, durch begrenzte, zwischengespeicherte Daten zahlreiche Möglichkeiten der Datenvorverarbeitung, wie z.B. Sortieren, Auszählen, usw. hat, muss man beim **Stream Processing** Daten sofort zur Laufzeit verarbeiten. Dennoch bringt Stream Processing zahlreiche Vorteile mit sich. So kann z. B. die Verarbeitung beginnen, ohne dass die Quelle bereits komplett erschlossen wurde. Da der Strom an neu ankommenden Daten theoretisch endlos ist, wird Stream Processing im Monitoring, für Web-Clickstream-Daten, Machine Learning-Systemdaten, sowie beim Speichern und Verarbeiten von Daten von Sensoren, Servern und ähnlichem verwendet [12].

2.3. Stream Prozessor Frameworks vs Graphendatenbanken

Der Fokus eines **Stream-Prozessor Frameworks** liegt auf der Verarbeitung von vielen Updates, die möglichst schnell vorgenommen werden sollen. Im Gegensatz zu **Graphdatenbanken** sind ihre Suchanfragen eher auf die Struktur gerichtet, als auf die Daten, die den Knoten und Kanten angehängt sind [13].

Ein weiterer Unterschied liegt in der Unterstützung von Transaktionen. Während Graphdatenbanken oftmals Transaktionen mit ACID Eigenschaften zulassen, ist dies bei Stream-Prozessor Frameworks unüblich [13]. Dies liegt unter anderem daran, dass die Updates von Streaming Events eher leichtgewichtet sind. Das heißt, meistens kommt nur eine Kante dazu, oder fällt weg. Allgemein liegt der Fokus von Stream-Prozessor Frameworks auf leichtgewichtigen Updates, die möglichst schnell und skalierbar umgesetzt werden sollen [13].

Graphdatenbanken gewährleisten eine große Auswahl an komplexen und reichhaltigen Graph Modellen, in denen sowohl Kanten als auch Knoten und ihre Eigenschaften von verschiedensten Typen sein können, u. a. Springs, Arrays und Daten BLOBs wie Bilder oder Audiodateien [13]. Die Datentypen in Stream-Prozessor Frameworks sind dagegen einfach gehalten, was noch einmal zeigt, dass der Fokus hier wirklich eher auf den Beziehungen zwischen Entitäten, bzw. der Struktur des Graphen liegt.

Ein weiterer interessanter Unterschied liegt im Support von *data replication* und *data sharding*. Data replication bedeutet, dass mehrere Kopien einer Datenbank auf unterschiedlichen Maschinen laufen. Dadurch kann z. B. die Erreichbarkeit einer Datenbank garantiert werden, auch wenn es mal zu einem Serverausfall kommen sollte [14]. Database sharding hingegen verteilt eine meist sehr große Datenbank auf mehrere Maschinen, wodurch ihre Performance verbessert werden kann [14]. Es ist zu beobachten, dass Stream-Prozessor Frameworks, welche verteilte Berechnung unterstützen, auch das deutlich mächtigere, aber auch komplexere Data sharding anbieten können. Herkömmliche Graphendatenbanken tun sich damit schwerer. So hat der bekannte Anbieter Neo4j etwa erst kürzlich data sharding für einige seiner querys angeboten [13]. Daraus kann man schließen, dass Eigenschaften wie eine schnelle Performance und ständige Erreichbarkeit für Stream-Prozessor Frameworks einen höheren Stellenwert haben, als für die herkömmlichen Graphdatenbanken.

2.4. Metriken im Stream Processing

In ihrem Papier *"Benchmarking Distributed Stream Data Processing"* [8] haben die Verfasser die Metriken Latency und Throughput im Bezug auf die Evaluation von Streaming Systemen neu definiert. Jene Definitionen, welche im Folgenden erläutert werden, sollen auch für diese Arbeit gelten.

2.4.1. Latency

Die Latency einer Anwendung wird in Zeiteinheiten angegeben. Da moderne Streaming Prozessoren aber zwei Arten von Zeit, nämlich *event-time* und *processing-time*, kennen, muss folglich auch die Latency bezüglich dieser zwei Arten unterschieden werden.

Definition 3. *Die **event-time** ist definiert als der Zeitpunkt, in dem ein Ereignis stattfindet.*

Definition 4. *Die **processing-time** beschreibt die Zeitspanne, in der ein Ereignis von einem Operator verarbeitet wird.*

Aus diesen Definitionen lassen sich nun die entsprechenden Latency Metriken ableiten:

Definition 5. *Die **event-time Latency** ist das Zeitinterval zwischen der event-time eines Objekts und dem Zeitpunkt seiner Emission im Output Operator des zu testenden Programms.*

Definition 6. *Die **processing-time Latency** spiegelt das Zeitinterval zwischen dem Eintreffen eines Objekts im Operator und dem Zeitpunkt seiner Emission im Output des Operators wieder.*

Klarer wird der Unterschied beider Metriken anhand eines Beispiels:

Gegeben sei die Banküberweisung B eines Kunden A. Nachdem A alle für die Überweisung nötigen Angaben getätigt hat, klickt er auf Ausführen. Dieser Zeitpunkt sei für das Beispiel die event-time. Da es zufälligerweise in genau diesem Moment zu einer großen Anzahl an Überweisungsaufträgen kommt, muss B zwei Sekunden in einer Warteschlange verbringen, bevor sie anschließend innerhalb von drei Sekunden verarbeitet und durchgeführt wird. Zum Schluss wird auf dem Bildschirm von A die Rückmeldung Überweisung erfolgreich ausgegeben. Die event-time Latency wäre in diesem Fall fünf Sekunden, während die processing-time Latency nur die drei Sekunden Verarbeitungszeit wären.

Anhand des Beispiels kann man sehen, dass die event-time Latency das Warten von Objekten in Warteschlangen mit einbezieht, während die processing-time Latency dies nicht tut und damit eine Teilmenge ersterer ist. Trotzdem ist die Aussagekraft beider Metriken wichtig. Die event-time Latency beispielsweise gibt Ausschluss darüber, wie lange ein Nutzer mit einer Anwendung kommuniziert. Ihr Wert hilft, wenn man versucht eben jene Zeit zu reduzieren. In manchen Fällen aber, in denen die Entwickler keinen Einfluss darauf haben, wie lange Objekte in Warteschlangen verbingen müssen, z.B. wenn aus Synchronisationsgründen die Anfrage eines anderen Nutzers vorher

verarbeitet werden muss, ist die event-time Latency uninteressant. Stattdessen ist in diesen Fällen die reine Verarbeitungszeit, also die processing-time Latency, zur Bestimmung der Performance der Anwendung ausschlaggebend.

Eine weitere Schwierigkeit bei der Bestimmung der Latency Metriken ergibt sich aus der Arbeitsweise von Operatoren mit Windows und Aggregationsfunktionen. Wenn diese mehrere Objekte mit unterschiedlichen event- und processing-time Werten beinhalten, ist es meist nicht trivial, welche davon ihr Aggregat bekommen soll. Die Verfasser des anfangs erwähnten Papiers haben entschieden, dass in diesen Fällen, sowohl die event-time, als auch die processing-time des Outputs jeweils das Maximum aller beinhaltenden event-, bzw. processing-time Werte seien sollen. Während die Berechnung des Maximums der event-times kein Problem darstellt, muss für jedes im ersten Operator eingehende Element ein Timestamp angelegt werden, um überhaupt festhalten zu können, was die jeweilige processing-time der Elemente ist. Um anschließend die processing-time Latency zu berechnen, muss am Ende des Windows lediglich die Differenz zwischen der aktuellen Systemzeit und des Timestamps mit der maximalen processing-time gebildet werden. Die Entscheidung nicht das Minimum aller event-times als die event-time des Outputs zu nehmen, beruht auf der Ansicht, dass nur die vergeudetete Wartezeit des Windows, auf Objekte, die nicht mehr kamen, zur event-time Latency zählen soll. [8]

2.4.2. Throughput

Definition 7. *Der **Throughput** einer Anwendung ist definiert als die Anzahl an Objekten, die von eben jener in einer gegebenen Zeitspanne verarbeitet werden kann.*

Im Kontext dieser Arbeit sei diese Zeitspanne als eine Sekunde festgelegt. In Streaming Systemen ist der Dateneingang nicht selten starken Schwankungen ausgesetzt. Wenn die Anzahl eingehender Objekte steigt, muss eine Anwendung in der Lage sein, zu skalieren, um Datenstau zu verhindern. Um Streaming Operatoren auf diese Anforderung zu testen, wurde eine erweiternde Metrik eingeführt: der *Sustainable Throughput*.

Definition 8. *Der **Sustainable Throughput** sei definiert als der größtmögliche Dateneingang, den eine Anwendung verarbeiten kann, ohne dass es zu einem anhaltenden Datenstau kommt, bzw. ohne das die event-time Latency kontinuierlich steigt.*

Der Zusammenhang zwischen event-time Latency und Sustainable Throughput ergibt sich dabei aus den Definitionen der beiden Begriffe. Befindet sich ein Objekt in einer Warteschlange, weil ein Operator an seine Grenzen gestoßen ist und aktuell keine weiteren Objekte verarbeiten kann, ist die event-time des Objekts weiter in der Vergangenheit, wenn das Objekt im Ausgangsoperator der Anwendung ankommt. Als Folge dessen steigt die event-time Latency. Es ist wichtig, den Sustainable Throughput eines Programms zu kennen, da bei Überschreitung dessen, die event-time Latency sogar kontinuierlich ansteigt. Dies liegt darin begründet, dass wenn der Dateneingang nicht abnimmt, die Warteschlange immer länger wird. Beschränkt man die Evaluation auf den Throughput nach der herkömmlicher Definition, hat man diesen Zusammenhang nicht. So könnte man beispielsweise

Objekte vor der Streamverarbeitung zusammenfassen. Dies würde den Throughput der Operatoren erhöhen, obwohl dafür die event-time Latency ansteigt. Der Sustainable Throughput würde dagegen nicht ansteigen, da dies vorraussetzen würde, dass die event-time Latency mindestens gleichbleibt [8].

3. Apache Flink

3.1. Überblick

Apache Flink ist ein Framework der Apache Software Foundation für zustandsbehaftete Berechnungen über begrenzte und unbegrenzte Streams. Programme in Flink laufen parallel und verteilt [15]. So hat jeder Stream eine oder mehrere Partitionen und jeder Operator hat wiederum eine oder mehrere voneinander unabhängige Unteraufgaben, die sogar auf unterschiedlichen Maschinen laufen können. Die Anzahl der Unteraufgaben kann unter dem Begriff parallelism für jeden Operator festgelegt werden [11]. Flink ist als Open-Source Software verfügbar und wird unter anderem von bekannten Unternehmen wie Uber, Ebay und der Otto Gruppe eingesetzt [15]. Auch in der Forschung gilt Apache Flink als aufstrebendes Stream-Prozessing Framework [16].

3.2. Pipelines

Pipelines funktionieren nach dem sogenannten ETL Prinzip (extrahiere, transformiere, lade). Während herkömmliche ETL Jobs periodisch ausgeführt werden, operieren Pipelines kontinuierlich und sind dadurch perfekt für Streaming geeignet [17].

Der große Vorteil von Pipelines sind neben der geringen Latenz beim Verschieben von Daten auch die Vielseitigkeit der Anwendbarkeit. Beispielsweise könnte eine Anwendung einen Event-Stream in einer Datenbank materialisieren, oder eine anderes ein Datenverzeichnis überwachen und Veränderungen in einem Ereignisprotokoll festhalten. Apache Flink unterstützt Pipelines schon alleine durch seinen Aufbau. Mit Hilfe seiner DataSource API kann eine große Variation an Datensätzen eingebunden werden; durch seine Table- und DataStream API sind eine Vielzahl an Datentransformation ermöglicht und zur Speicherung bietet Flink ein reiches Kontingent an Schnittstellen zu verschiedenen Datenbanksystemen wie Kafka, Kinesis, Elasticsearch und JDBC an [17].

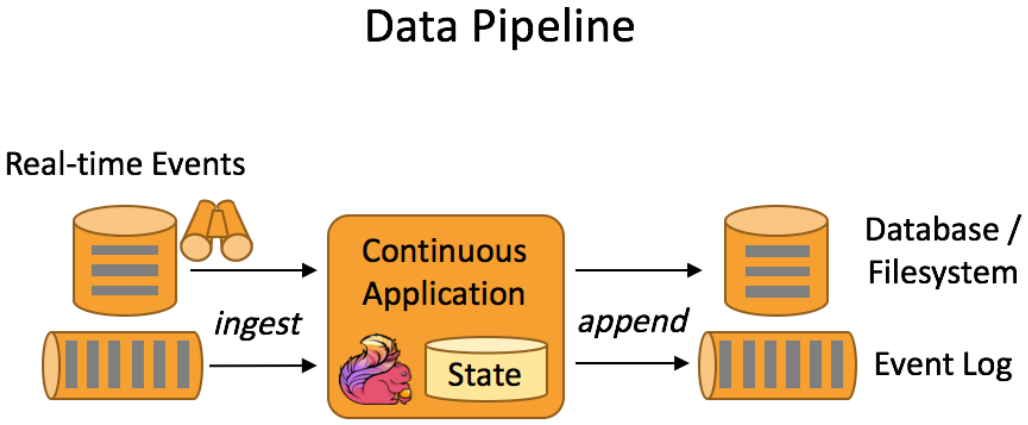

Abbildung 3.1.: Aufbau einer Data Pipeline in Apache Flink [18]

3.3. DataStream API

Apache Flinks DataStream API hat ihren Namen von der Klasse *DataStream*, welche im Stream Prozessor einen Datenstrom repräsentiert. Viele Funktionen von *DataStream* erinnern an eine herkömmliche Java Collection, jedoch gibt es einige entscheidende Unterschiede. Erstens ist der Inhalt des Streams immutable, was bedeutet, dass Elemente, wenn sie erst einmal erzeugt wurden, weder hinzugefügt, noch entfernt werden können. Auch ein gezieltes Herausnehmen und Sichten der Elemente ist nicht möglich. Stattdessen kann der Entwickler über bestimmte Operatoren wie *map()*, *filter()* usw. Einfluss auf den Stream nehmen. Diese werden auch Transformationen genannt und bilden den Kern einer jeden Analyse Anwendung. Schließlich ist es auch die Performance der Zusammenstellung jener Transformationen, die von der Evaluationssoftware dieser Arbeit getestet werden soll [19].

Um einen *DataStream* zu erzeugen, muss man seinem Programm eine Source hinzufügen. Über die Source entscheidet man auch welcher Datentyp initial gestreamt werden soll. Flink erlaubt neben dem Streamen von primitiven Datentypen wie *Integer*, *Char* oder *Boolean*, auch komplexere Strukturen wie *Tupel*, *POJOs* und *Scala Case* Klassen [11].

Syntax 1 (Tupel)**.**

$$Tuple_N < Datentyp_1, Datentyp_2, \ldots, Datentyp >$$

Ein Datentyp gilt als POJO, wenn die Klasse *public* ist, keine nicht-statische innere Klasse besitzt, einen *public* Konstruktor ohne Argumente hat und entweder alle nicht statischen Attribute *public* sind, oder Getter- und Settermethoden nach der Java beans Namenskonvention haben [11].

Ist ein DataStream erzeugt, durchläuft er die bereits erwähnten Transformationen, wodurch er verändert, aufgeteilt oder mit anderen DataStreams zusammengeführt werden kann. Darüber hinaus stellt Flink auch komplexe Aggrigationsfunktionen und Windowing zur Verfügung. Mit Windows teilt man einen unbegrenzten Stream in begrenzte Zeitfenster auf, um Zugriff auf mehr Berechnungsmöglichkeiten zu erhalten [19].

Sobald ein DataStream alle erforderlichen Transformationen durchlaufen hat und das gewünschte Ergebnis repräsentiert, kann er mittels eines DataSinks ausgegeben, oder an ein drittes System, wie z.B. eine Datenbank, übergeben werden. Dies stellt normalerweise den letzten Schritt in der Anatonmie eines Flink Progamms dar. Anschließend muss nur noch mittels der *execute()* Methode des StreamExecutionEnvironment die Programmausführung ausgelöst werden [19].

Jede Anwendung, die auf Apache Flink aufbaut, benötigt eine Ausführungsumgebung. Arbeitet das Programm mit Streams statt Batches wird speziell ein *StreamExecutionEnviroment* benötigt. Jeder Aufruf der DataStream API im Code wird in einem Graphen gespeichert, welcher an das *StreamExecutionEnviroment* angehängt ist. Sobald *.execute()* aufgerufen wird, wird dieser Graph entpackt und an den Job Manager gesendet, welcher dann die Aufgaben parallelisiert und an die Task Managers zur Ausführung verteilt. Diese verarbeiten die Aufgaben dann jeweils in einem sogenannten Task Slot [11] [19].

Jedes *StreamExecutionEnviroment* besitzt eine *ExecutionConfig*, mit deren Hilfe man aufgabenspezifische Einstellungen vor und während der Laufzeit festlegen kann. Zu den Einstellungsmöglichkeiten gehören unter anderem der Verteilungsgrad (parallelism) der Anwendung, die Aktivierung der Web UI, des Metrikreports, sowie des Latency Trackings. Und dies sind lediglich ein paar der über hundert verschiedenen möglichen Konfigurationen, die Flink anbietet [20].

3.4. DataSource API

Data Sources werden in Apache Flink durch seine DataSource API realisiert. Sie umfassen hauptsächlich drei Kernelemente: Splits, Source Readers und den Split Enumerator. Ein Split ist ein Teil der Daten, die aus einer Quelle ausgelesen, durch eine Funktion erstellt, oder von einem Connector übertragen wurden. Durch Splits kann die Arbeit aufgeteilt und das Lesen der Daten parallelisiert werden. Sie werden von Source Readern angefragt, welche sie daraufhin weiterleiten, sodass sie parallel verarbeitet werden können. Wie in Abbildung 3.2 zu sehen, ist jeder Source Reader Teil eines Task Managers, welcher wiederrum eine Instanz der Verteilung der Streamverarbeitung repräsentiert. Der Split Enumerator, zu guter Letzt, erstellt die Splits und weist sie auf Anfrage dem jeweiligen Source Reader zu. Es existiert immer nur ein Split Enumerator, der auf dem Job Manager, bzw. dem Master Prozess läuft[21].

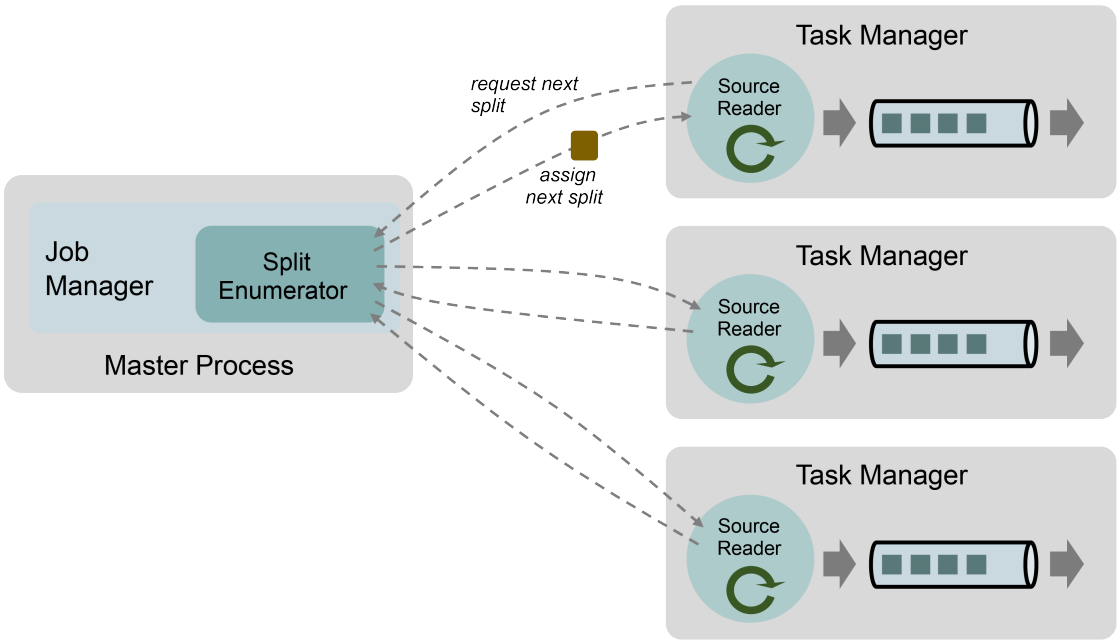

Abbildung 3.2.: Apache Flinks DataSource Architektur [22]

Wie bereits erwähnt, erlaubt Flink mehrere Arten von Quellen. Die einfachste Möglichkeit ist es, eine der bereits vorimplementierten *SourceFunctions* zu benutzen. Diese werden nach Herkunft der Daten unterschieden. File-based *SourceFunctions* lesen eine Datei, übergeben durch ihren Pfad, aus. Dieser Prozess findet in zwei Arbeitsschritten statt. Im ersten Schritt, auch directory monitoring genannt, wird das Verzeichnis und die darin enthaltenden Daten geladen, in Splits geteilt und an die Source Reader weitergeleitet. Im Stream Processing wird dieser Arbeitsschritt in einem

vom Entwickler festgelegten Intervall wiederholt, um zu sehen, ob neue Informationen hinzugefügt wurden. Beim Batch Processing wird das directory Monitoring nur einmal durchgeführt und danach beendet. Im zweiten Schritt werden die Informationen schließlich aus den Splits gelesen und als Stream weitergegeben. Während die Reader verteilt laufen, ist der Prozess des directory monitoring non-parallel. Die anderen vorimplementierten *SourceFunctions* funktionieren nach dem selben Prinzip, nur, dass die Splits, statt aus Daten von Sockets, Java.util.Collections, oder einem Connector wie Apache Kafka gewonnen werden [21].

Darüber hinaus erlaubt es Flink auch eigene *SourceFunctions* zu schreiben. Dazu werden dem Entwickler die Interfaces SourceFunction für nicht verteilte Sources, bzw. ParallelSourceFunction für verteilte Sources an die Hand gegeben. Alternativ kann man seine Source Funktion auch von RichParallelSourceFunction erben lassen, wodurch dem Entwickler unter Anderem Zugriff auf die Klasse RuntimeContext geben wird, mit deren Hilfe man z. B. Zugriff auf Flinks Metrik System bekommt oder Timer hinzufügen kann[19].

Um seinem Programm eine Data Source hinzuzufügen, benutzt man:

```
1 StreamExecutionEnvironment.addSource(sourceFunction);
```

Code 1: StreamExecutionEnviroment.addSource()

3.5. Metriksystem

3.5.1. Überblick

Apache Flink stellt Enwticklern ein umfangreiches System zur Verfügung, mit dem man Metriken in definierbaren Intervallen sammeln und ausgeben kann. Zugriff auf das System erhält man über das Objekt *RuntimeContext*, dass bereits im letzten Abschnitt über Data Sources erwähnt wurde. Doch kann es mithilfe der *getRuntimeContext()* Methode nicht nur in Sources, sondern innerhalb jeder Klasse aufgerufen werden, die von *RichFunction* erbt, wie beispielsweise die Transformationen *RichMapFunction*, *RichFilter*, usw. Dies macht es möglich in nahezu jedem Programmabschnitt Metriken zu registrieren oder zu aktualisieren [23].

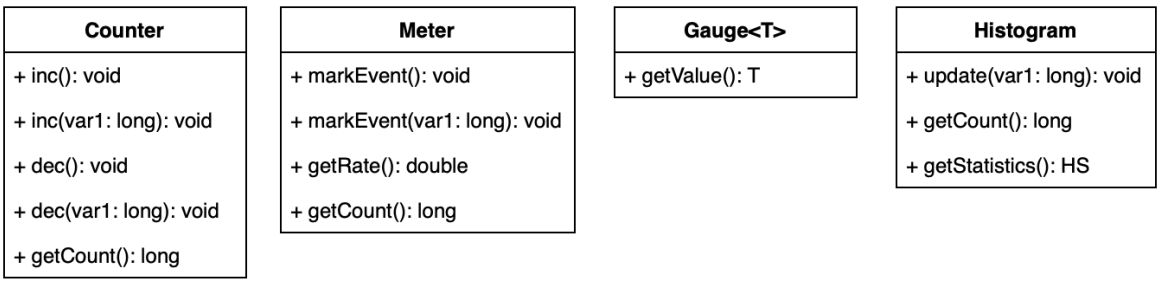

Abbildung 3.3.: Apache Flinks vier Metriktypen

Flink unterscheidet zwischen vier Metriktypen: *Counters*, *Meters*, *Gauges* und *Histograms*. Ein *Counter*, wie der Name schon vermuten lässt, zählt, wobei man sowohl hoch-, als auch runter-, in einer oder $n \in \mathbb{N}$ Schritten zählen kann. Ein *Meter* funktioniert genauso, ist aber noch um ein View erweitert, das anzeigt, wie viel in einem gewissen Zeitfenster gezählt wurde. Ein *Gauge* kann jeden beliebigen Wert T, was auch ein komplexes Objekt sein kann, annehmen. Das *Histogram* letztlich kann man als Sammlung von *long* Messwerten verstehen, mit denen man sich mit Hilfe statistischer Methoden Werte wie den Mittelwert, die Standardabweichung, das Minimum oder Maximum anzeigen lassen kann [23].

Neben den eigens registrierten Metriken gibt es eine ganze Reihe von vordefinierten Systemmetriken. Dazu gehören Messwerte der CPU, des Netzwerks, des Festplattenspeichers, der Ein- und Ausgabe; Anzahl der aktiven Live-Threads, sowie Werte die von Connectors wie Apache Kafka übermittelt werden. Apache Flink hat seine Systemmetriken gut dokumentiert. Man findet auf der offiziellen Website zu jeder Metrik eine kurze Beschreibung, den verwendeten Typ und welcher Kompontente sie hierarchisch direkt untergeordnet ist, also Job Manager, Task Manger, Operator oder Task [23].

Flinks Metriken werden in hierarchisch aufgebaute *Metric Groups* gegliedert. Kennt man den Namen und die *MetricGroup* einer Metrik, kann man sie damit eindeutig indentifizieren. Auf der obersten Ebene der Hierarchie differenziert man zwischen der *TaskManagerMetricGroup* und der *JobManagerMetricGroup*. Es gibt nur wenige Metriken die ausschließlich zur Gruppe des JobManagers zählen. Dazu gehören beispielsweise Werte wie *numRestarts*, welche die Anzahl an Neustarts beinhaltet, seit der jeweilige Job übergeben wurde, oder so etwas wie *totalNumberOfCheckpoints*. Die meisten Metriken, inklusive der selbst registrierten, befinden sich in der *OperatorMetricGroup*, welche eine Untergruppe der *TaskMetricGroup* darstellt, was wiederum eine Untergruppe der *TaskManagerJobMetricGroup* ist. Die vollständige Hierachie ist in Abbildung 3.4 zu sehen.

```
•TaskManagerMetricGroup
    •TaskManagerJobMetricGroup
        •TaskMetricGroup
            •TaskIOMetricGroup
            •OperatorMetricGroup
                •${User-defined Group} / ${User-defined Metrics}
                •OperatorIOMetricGroup
•JobManagerMetricGroup
    •JobManagerJobMetricGroup
```

Abbildung 3.4.: MetricGroup Hierachie [24]

3.5.2. Zugriff per REST API

Um auf die Messwerte zuzugreifen, werden dem Entwickler zwei Möglichkeiten geboten. Die auf dem ersten Blick zugänglichere der beiden funktioniert über Flinks REST API. Sie akzeptiert HTTP Requests und antwortet im JSON Format. Das Ganze läuft über einen Webserver, der voreingestellt

auf Port 8081 läuft, was aber über die *Config* beliebig geändert werden kann. Auf dem gleichen Port läuft übrigens auch Flinks WebUI, mit der man ebenfalls ein paar der Metriken einsehen könnte. Da das Web-Dashboard jedoch ohnehin ziemlich unübersichtlich ist, sei an dieser Stelle davon abgeraten. Den Kern der REST API bildet die Klasse *WebMonitorEndpoint*, welche den Server aufsetzt und das Routing der Anfragen steuert. Als Client-Server Framework werden die Bibliotheken Netty und Netty-Router verwendet. Flink begründet seine Wahl für Netty mit den lediglich leichtgewichteten Dependencies bei dennoch sehr guter Performance [25].

```
1  public interface MessageHeaders<R extends RequestBody, P extends ResponseBody, M
       extends MessageParameters> extends UntypedResponseMessageHeaders<R, M> {
2
3      Class<P> getResponseClass();
4
5      HttpResponseStatus getResponseStatusCode();
6
7      default Collection<Class<?>> getResponseTypeParameters() {
8          return Collections.emptyList();
9      }
10
11     String getDescription();
12 }
```

Code 2: Interface MessageHeaders [26]

Um eigene Requests an die API senden zu können, müssen 3 Schritte abgearbeitet werden. Zuerst muss durch eine eigene Klasse das Interface *MessageHeaders* mit seinen vier Methoden implementiert werden, wie sie in Codebeispiel 1 dargestellt sind, wobei *getResponseTypeParameters()* bereits eine default Implementierung besitzt. Im nächsten Schritt muss eine *AbstractRestHandler* Klasse geschrieben werden, in welcher geklärt wird, wie mit den Requests verfahren werden soll. Im letzten Schritt muss dann der Handler noch innerhalb der Methode

```
1  org.apache.flink.runtime.webmonitor.WebMonitorEndpoint\#initializeHandlers();
```

Code 3: initializeHandlers()

hinzugefügt werden. Um sich vor der Implementierung eigener Klassen einen Überblick zu verschaffen, hilft es sich ein Beispiel, wie den *JobExceptionsHeaders* und ihrem *JobExceptionsHandler* anzusehen [25].

3.5.3. Zugriff per Metrik Reporter

Die zweite Möglichkeit auf Flinks Metriksystem zuzugreifen, funktioniert über sogenannte Metrik Reporter. Metrik Reporter werden von Flink als Plugins geladen, was bedeutet, dass ihr Code vom Rest des Programms strikt getrennt ist. So kann man beispielsweise von einem Plugin nicht ohne weiteres auf eine Klasse aus einem anderen Plugin oder aus Flink zugreifen. Will man es doch, müsste man die Klasse, oder ihr Paket erst speziell freischalten. Diese werden dann als *whitelisted*

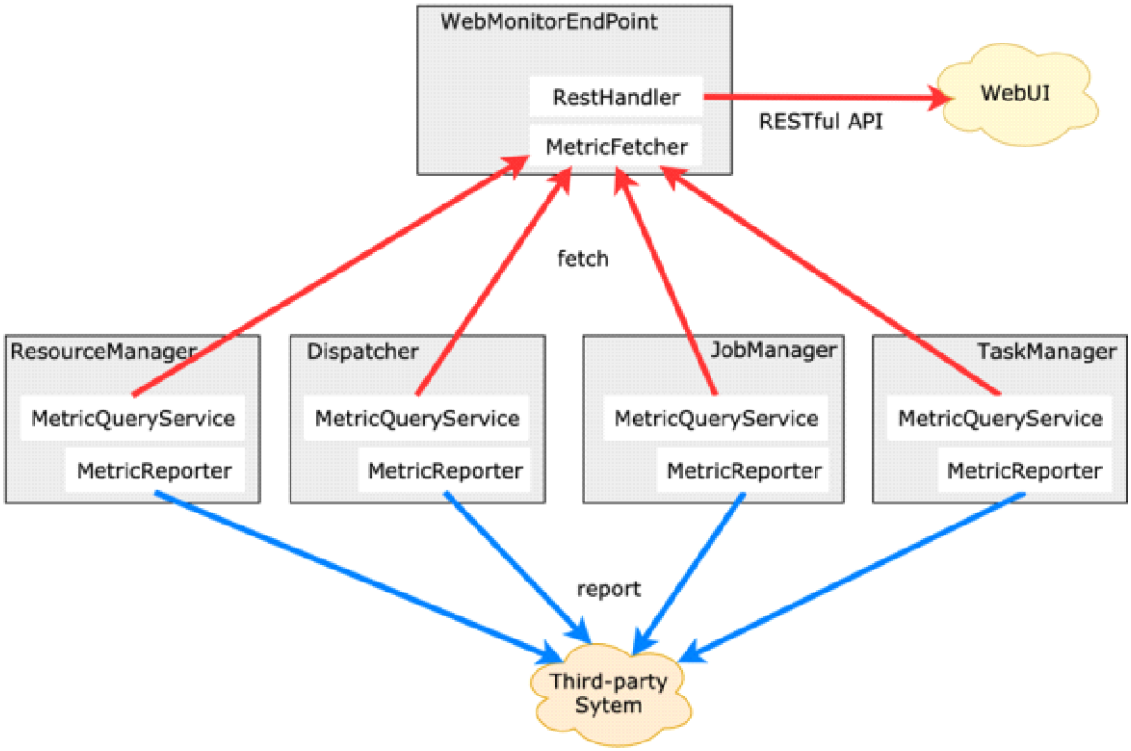

Abbildung 3.5.: Apache Flinks Metriksystem [27]

bezeichnet. Plugins bringen den Vorteil mit sich, dass sie Bibliotheken verwenden können, die im Konflikt mit bereits von anderen Programmteilen geladenen Bibliotheken stehen. Doch scheinen sie innerhalb von Flink noch nicht besonders verbreitet zu sein. Jedenfalls werden bisher nur Metrik Reporter und Datensysteme als Plugins geladen [28] [29].

Um einer Anwendung einen oder mehrere Metrik Reporter hinzuzufügen, müssen diese in der Flink *Config* unter dem Schlüssel *metrics.reporters* angegeben werden. Ist dies erfolgt, wird bei Start jedes Job- , bzw. *TaskManagers* auch eine Instanz der Reporter geladen. Sie arbeiten also verteilt, falls auch das Programm verteilt läuft, bzw. der *parallelism* größer als eins gesetzt wurde [29].

Apache Flink unterstützt von Haus aus acht verschiedene Metrik Reporter. Das sind JMX, Graphite, InfluxDB, Prometheus, PrometheusPushGateway, StatsD, Datadog und Slf4j. Sie alle werden über das *org.apache.flink.metrics.reporter.MetricReporter* Interface eingebunden, durch dessen Implementierung man auch seinen eigenen Reporter schreiben kann. Bei ihrer Arbeitsweise wird zwischen Push- und Pull-Prinzip unterschieden. Während Reporter, die nach dem Push-Prinzip arbeiten, normalerweise das *Scheduled* Interface implementieren und dadurch in regelmäßigen Intervallen die Metriken an ein externes System senden, werden nach dem Pull-Prinzip arbeitende Reporter von dritten Systemen aufgefordert, die aktuellen Metriken herauszugeben. Wenn das *Scheduled* Interface implementiert wurde, kann in der *Config*über den Key *metrics.reporter.<reporter_name>.interval* das passende Reporting-Interval definiert werden. Voreingestellt beträgt es 10 Sekunden [29] .

3.5.4. Latency Marker

Apache Flink ermöglicht Entwicklern über sogenannte Latency Marker die Zeit zu messen, die ein Objekt braucht, um die jeweiligen Subtasks eines Operators zu durchlaufen. Aus Performancegründen ist dieses Feature, welches selbst einiges an Rechenleistung benötigt, deaktiviert. Um es zu aktivieren, muss das *latencyTrackingInterval* per *Config* auf einen positiven Wert geändert werden. Dies sollte allerdings wirklich nur zu Evaluationszwecken passieren. Der eingetragene Wert gibt gleichzeitig an, in welchen Zeitabständen, gemessen in Millisekunden, Latency Marker emittiert werden. Anschließend durchlaufen sie das Programm. Da sie keine echten Objekte, sondern nur Dummys, ausgestattet mit einem *Timestamp*, sind, überspringen sie die Transformationen und werden somit nicht verarbeitet. Der *Timestamp* steht dabei für den Zeitpunkt ihrer Emission. Um trotz der fehlenden Verarbeitung die Latenz der Subtasks messen zu können, wurden sie dahingehend eingeschränkt, dass es ihnen nicht möglich ist, ihren Vorgänger im Datenstrom zu überholen. Steckt der Vorgänger beispielsweise in einer Warteschlange fest, so muss auch der Latency Marker warten. Ausgenommen von der Messung ist dagegen die Zeit, die Objekte in der Verarbeitung verbringen, dies ist besonders auch dann zu beachten, wenn die Transformation Windows enthält, die Verzögerungen verursachen. Diese bleiben von den Latency Markern unberücksichtigt. Widergespiegelt wird stattdessen die Zeit, in denen das Programm und im speziellen die Subtasks überlastet sind und sich ein Datenstau bildet [23] .

Hat ein Latency Marker einen Subtask überwunden, wird die Differenz zwischen diesem Zeitpunkt und dem Zeitpunkt als der vorhergehende Subtask überwunden wurde, in einem *Histogram* abgelegt. Gab es keinen vorhergehenden Subtask, gilt der Zeitpunkt der Emission des Markers.

Über *metrics.latency.granularity* kann man in der *Config* die Granularität der Messungen einstellen. Standardmäßig ist diese auf *operator* gestellt, was bedeutet, dass für jede Source pro Subtask ein eigenes *Histogram* erstellt wird. Auf *single* gestellt, werden ungeachtet der Source, alle Werte der Latency Marker pro Subtask in ein *Histogram* gespeichert. Ist die Granularität auf *subtask*, wird sowohl für jede Source, als auch darüber hinaus noch für jeden Verteilungsgrad pro Subtask ein eigenes *Histogram* angelegt [30].

Im Hinblick auf die im Kapitel 2.4 eingeführten Definitionen, handelt es sich bei dem von den Latency Markern gemessenen Wert, um eine Sonderform der event-time Latency. Der Unterschied besteht darin, dass die Latency Marker, die Verarbeitungszeit der Transformationen, also die processing-time Latency, nicht misst. Vermutlich wurde dies so konzipiert, um es dem Entwickler zu überlassen, wie er das nicht triviale Problem der Messung von event-, bzw. processing-time Latency in Operatoren mit Windows lösen will. Dafür würde auch sprechen, dass gerade deren Verarbeitungszeit explizit nicht für die Latency Marker zählt.

Damit es bei den Messungen der Latency Marker zu keinen Fehlern kommt, sollte sichergestellt werden, dass die Uhren aller Maschinen im Cluster synchronisiert sind [23].

4. Konzeption

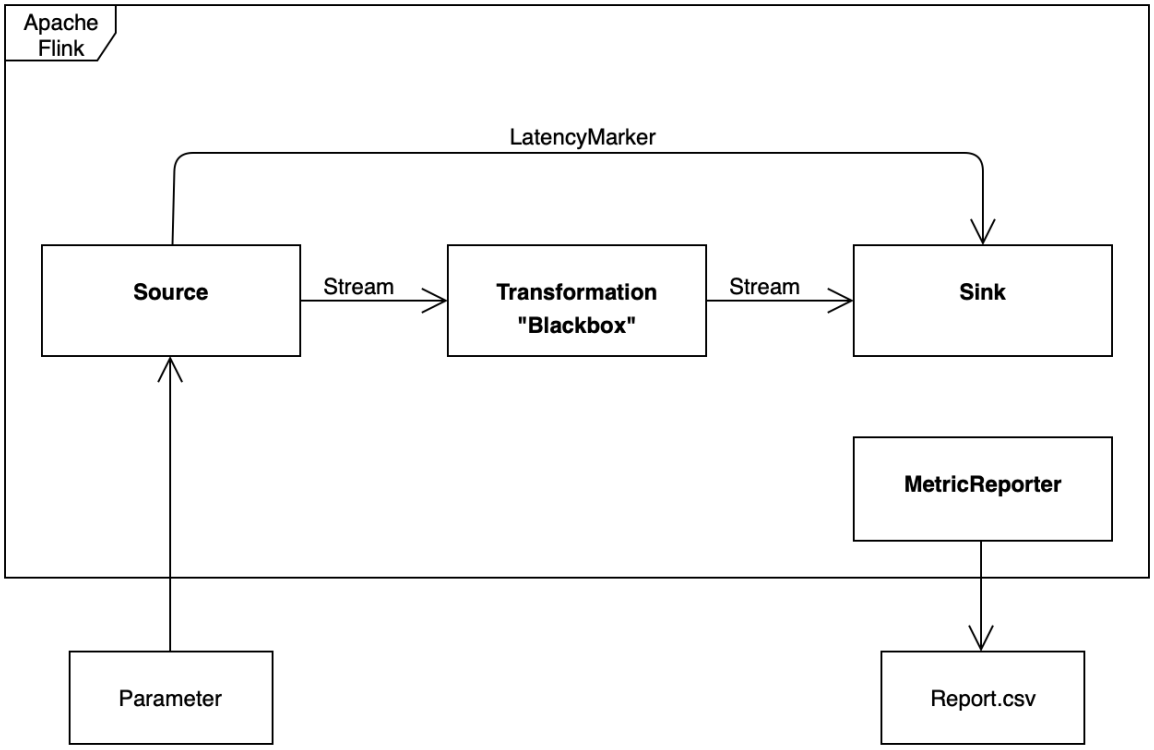

Abbildung 4.1.: Konzeption des Prototypen

4.1. Modell ETL Pipeline

Das Evaluationsprogramm wurde als ETL Pipeline konzipiert, sodass man dessen Arbeitsweise in drei Schritte unterteilen kann. Im ersten Schritt, der Extrahierung, geht es darum, dem zu testenden Operator geeignete Daten in Form eines *DataStreams* zur Verfügung zu stellen, wobei der Nutzer die Wahl zwischen künstlich generierten und realen Daten bekommen soll. Den zweiten und normalerweise wesentlichen Schritt, die Transformation, übernimmt anschließend ausschließlich die zu evaluierende Analyse. Diese soll vom Nutzer, möglichst ohne viel Änderungen im Code, ins Evaluationsprogramm eingebunden werden können. Den letzten Schritt, das Laden, teilen sich Evaluationstool und Operator, indem Ersteres die berechneten Metriken ausgibt, während der Operator den transformierten *DataStream* weitergibt.

Bei der Erarbeitung des Modells wurde von Beginn an größte Aufmerksamkeit auf die Wiederverwendbarkeit (*Reusability*) der Software gelegt. Das Projekt soll gerade keine auf die Evaluation eines Operators zugeschnittene Lösung sein, sondern universell als Performancetest aller mit Apache Flinks DataStream API laufenden Programme genutzt werden können. Um dies zu gewährleisten, durfte die Arbeitsweise des *textitSystems Under Test* keinen Einfluss auf die Art der Evaluation haben. Sie stellte also für die Entwicklung des Prototypen dieser Arbeit, sowohl in der Modellierung,

als auch in der Implementierung eine Blackbox dar. Darüber hinaus wurde, um die Wiederverwendbarkeit zusätzlich zu erhöhen, der Ansatz der Modularen Programmierung verfolgt. Durch möglichst viele Schnittstellen und Parameter soll es dem Nutzer ermöglicht werden, die Evaluation seinen individuellen Bedürfnissen anzupassen. Man kann die Anwendung durchaus auch als Framework verstehen, welches erst durch die individuellen Einstellungen des Nutzers, sein ganzes Potential ausschöpft.

4.2. Datengenerierung

4.2.1. Datentyp *StreamTriple*

In Kapitel 2.1 wurde die Datenstruktur des Graphen, bzw. Property Graphen vorgestellt. Nun soll die Modellierung des letztlich angewandten Datentypen geklärt werden: des *StreamTriple*. Ein *StreamTriple* umfasst genau eine Beziehung zwischen zwei Enitäten, wobei jeder Entität, entweder die Rolle des Start-, oder der des Endknoten, eindeutig zugeteilt wird. *StreamTriple* repräsentieren also gerichtete Graphen, weshalb zwei von ihnen gebraucht werden, falls eine ungerichtete Beziehung dargestellt werden soll. Die Entitäten werden alle durch ein und die selbe Klasse *StreamVertex* modelliert, welche neben einer *ID* und einem Label vom Typ *String*, noch einen *Timestamp* und beliebig viele Eigenschaften in einem Objekt *Properties* speichert. Die Klasse *Properties* ist im Grunde eine skalierbare *HashMap*, welche als Schlüssel *Strings* verlangt, während ihre Werte einen beliebigen Datentyp aufweisen können. Im *StreamTriple* werden neben den beiden *StreamVertex*, ebenfalls eine *ID* und ein Label vom Typ *String*, sowie ein *Timestamp* und ein *Properties* Objekt hinterlegt.

Die Summe aller *StreamTriple* bilden den Graph Stream, wobei es durchaus Sinn macht, nur einzelne Beziehungen, statt komplette Graphen zu streamen. So muss bei einer kleinen Veränderung beispielsweise, nur die betroffene Kante, anstatt des kompletten Graphen losgeschickt und anschließend verarbeitet werden. Insbesondere im Anwendungsfeld, welches in der Motivation erschlossen wurde, kommt es häufig zu vielen, leicht gewichteten Änderungen, was die Modellierung des Graph Streams mit *StreamTriplen* besonders attraktiv macht.

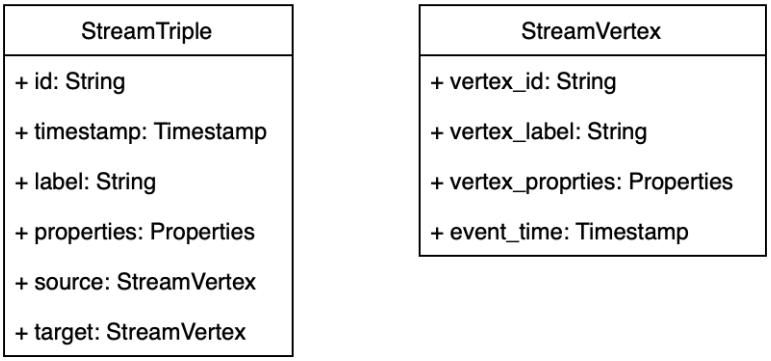

Abbildung 4.2.: Modelle der Klassen *StreamTriple* und *StreamVertex*

Das Modell des *StreamTriple* wurde auch deswegen gewählt, weil es vom *GraphStreamZoomer*, dem Graph Stream Grouping Algorithmus, der zum Abschluss dieser Arbeit evaluiert wurde, benutzt wird [5]. Definiert wurde das Modell in der Masterarbeit *Distributed Grouping of Property Graph Streams* von Rana Noureddin[31, S. 34].

4.2.2. Generische Datenquelle

Gemäß dem Ansatz der Modularen Programmierung, soll es dem Nutzer der Evaluationssoftware frei stehen, welche *SourceFunction* er für seinen Test verwenden möchte. Es soll lediglich feststehen, dass Objekte des Typs *StreamTriple* gestreamt werden. Dies reduziert zwar die Anwendbarkeit der Analyse auf Graph Stream Operatoren, davon wurde aber auch in der Zielsetzung ausgegangen. Um dem Nutzer Arbeit zu ersparen, sollen mit der Software zwei verschiedene *SourceFunctions* mitgeliefert werden, welche zusammen schon viele Anwendungsfälle abdecken können. Erfordert der zu testende Operator darüber hinaus dennoch eine individuellere Lösung, kann er mit Hilfe der in Kapitel 3.4 erwähnten Interfaces, seine eigene *SourceFunction* implementieren.

Die erste der beiden mitgelieferten *SourceFunctions* soll synthetische Daten generieren, wobei es dem Nutzer möglich sein muss, Einfluss auf die Wertebereiche der Attribute der zufällig generierten *StreamTriple* zu nehmen. Dies könnte beispielsweise, über Parameter in den Konstruktoren der Source ermöglicht werden.

Die Motivation für synthetische Daten sind schnell umsetzbare Tests, bei denen es ausreicht, realitätsnahe Informationen zu verarbeiten. Gerade bei Performancetests spielen die Resultate der Analyse, im Gegensatz zu den gemessenen Metriken, eine untergeordnete Rolle. Ist es dem Nutzer sogar völlig gleichgültig, welche Daten verarbeitet werden sollen, wird auch ein Test mit Platzhalterwerten wie $\{A, B, C, ...\}$ für die Label der Knoten, sowie $\{a, b, c, ...\}$ für die Label der Kanten möglich sein. Dies kann z.B. dann nützlich sein, wenn man nur einen Stresstest mit möglichst hohem Input durchführen will. Auf Properties wird bei diesen Tests völlig verzichtet.

Als letztes Feature soll es dem Nutzer ermöglicht werden, für die Source eine Ausgabebegrenzung festzulegen. Einerseits kann man den Datenstrom damit für einige Anwendungen realitätsnaher simulieren, andererseits sind dadurch spezifischere Test dahingehend möglich, dass der Nutzer sehen kann, wie sich die zu evaluierende Software unter verschiedenen, oder gleichbleibenden Eingangsgrößen verhält.

4.2.3. Reale Datenquelle

Es ist vorstellbar, dass synthetische Daten für einige Analyseprogramme zu unspezifisch sind, um sie effektiv testen zu können. Aus diesem Grund soll es für den Nutzer möglich sein, seinen eigenen Datensatz zu verwenden. Dafür wird eine zweite *SourceFunction* implementiert werden, die eine *Collection* aus *StreamTriplen*, je nach Wahl, entweder zufällig, oder mit Reihenfolge streamt. Der Nutzer kann dabei entscheiden, ob das Programm, nachdem die *Collection* durchgearbeitet wurde, beendet wird, oder ob sich der Datenstrom wiederholt. Hat man sich für die zufällige Reihenfolge

der Eingabe entschieden, kann so, trotz Wiederholung, eine gewisse Unvorhersehbarkeit des Streams simuliert werden.

Steht der eigene Datensatz jedoch nicht in Form einer *Collection*, sondern als *File* oder Ähnlichem zur Verfügung, benötigt man einen *Parser*, der die gegebenen Informationen in Form von *StreamTriple*n ablegt. Ein solcher textitParser wird am Beispiel eines Datensatzes des New York Citi Bike Services mitgeliefert. Dieser dient jedoch lediglich der Orientierung, da er ausschließlich mit Datensätzen des Citi Bike Services arbeiten kann. Für seine Quelle kann der Entwickler entweder einen der online verfügbaren textitParser nutzen, einen Eigenen schreiben, oder alternativ auch eine eigene *SourceFunction* implementieren, welche die *StreamTriple* aus einem *File* oder Ähnlichem generiert.

4.3. Evaluation

4.3.1. Latency

Nachdem im Abschnitt 2.4 Latency im Kontext von Stream Processing definiert wurde, wird nun ein geeignetes Modell zur Messung dieser diskutiert. Im Mai 2022 erschien auf der offiziellen Flink Webseite der Blogpost *"Getting into Low-Latency Gears with Apache Flink"*, der genau diese Thematik behandelte [32]. Die Autoren erläutern darin ihren Ansatz der Messung von End-To-End Latency, welche nach der Definition dieser Arbeit auch event-time Latency genannt wird, anhand von zwei verschiedenen Szenarien. In Szenario 1. gilt es die Latenz einer simplen Pipeline zu bestimmen, die ohne *Windows*, *Timer* oder andere komplexe, zeitkritische Logik auskommt und für jedes eingehende Objekt auch eines ausgibt. In diesen Fällen bestimmen sie die Differenz *t2-t1* zwischen dem Zeitpunkt der Emission *t1* des zu verarbeitenden Objekts und seiner Ankunft im *Sinkoperator* der Pipeline *t2*. Im zweiten Szenario werden alle übrig geblieben Fälle, in denen man keine Einschränkungen voraussetzen kann, abgedeckt. In diesen Fällen messen die Autoren den *Event Time Lag*, der sich aus der Differenz der aktuellen Systemzeit und dem *end Timestamp* des *Windows* ergibt. Sein Wert spiegelt den Unterschied zwischen dem erwarteten Ausgabezeitpunkt und dem Tatsächlichen. Als Watermarkstrategie nutzten sie *forBoundedOutOfOrderness*.

Für den Prototyp des Evaluationstools dieser Arbeit ist der Ansatz des ersten Szenarios auf Grund seiner zahlreichen Einschränkungen leider nicht ausreichend. Um eine hohe Verwendbarkeit des Performancetests zu gewährleisten, kann man nicht davon ausgehen, dass nur Operatoren evaluiert werden, die für jeden Input, auch einen Output generieren, oder auf komplexere Programmlogik verzichten. Szenario 2. hingegen wäre zwar eine geeignete Methode, um auch die Latenz von Operatoren, die mit komplexerer Zeitlogik arbeiten, zu messen, fällt aber ebenfalls als Lösung weg, da man für diesen Ansatz Zugriff auf das *System Under Test* benötigt, was jedoch wie in 4.1 festgelegt, eine Blackbox darstellt.

Nach bestem Wissen funktioniert die einzige universell anwendbare Methode Latenz zu messen, ohne dabei Zugriff auf die Operatoren zu haben, über die von Apache Flink bereitgestellten Latency Marker, die bereits in Kapitel 3.5.4 vorgestellt wurden. Allerdings spiegelt der von ihnen

gemessene Wert nur einen Teil der End-to-End Latency wieder, weshalb dem Nutzer der Evaluationssoftware zumindest die Möglichkeit gegeben werden muss, selbst eine Metrik zu registrieren, welche die Messungen der Latency Marker ergänzt. Einen geeigneten Weg zu finden, vom Nutzer angelegte Metriken in der Ausgabe zu berücksichtigen, war deshalb eine der Hauptaufgaben der Implementierung des Prototypen.

4.3.2. Throughput

Der Throughput des textitSystems Under Test soll direkt in der jeweils verwendeten *SourceFunction* gemessen werden. Dafür muss dort eine Metrik vom Typ *Meter* registriert werden. Das Meter bieten sich als Metriktyp an, da es neben seiner Funktion als Zähler, auch ausgeben kann, wie viel in einer gegeben Zeiteinheit gezählt wurde. Den Umstand den Throughput direkt in der Datenquelle messen zu können, ist der Arbeitsweise von Apache Flinks DataSource API geschuldet. Wie in Kapitel 3.4 erklärt, verteilt der Split Enumerator, was im Fall dieses Modells z. B. eine der beiden mitgelieferten *SourceFunctions* sein kann, auf Anfrage die von ihm erstellten Splits an die Source Reader der Task Manager. Da dieser Vorgang nur auf Anfrage stattfindet, kann niemals mehr Input generiert werden, als die Gesamtheit der Taskmanger verarbeiten kann. Die Messung spiegelt also den Sustainable Throughput der Anwendung wieder.

Die Messung des Throughputs einzelner Transformationen, z. B. um den Flaschenhals einer Anwendung zu ermitteln, ist universell nicht möglich, da man dafür Zugriff auf das *System Under Test* benötigen würde. Allerdings sollte es dem Nutzer auch in diesem Fall möglich sein, solche Metriken im Nachhinein zu ergänzen.

4.3.3. Ausgabe

In Kapitel 3.5 wurden bereits zwei Möglichkeiten aufgezeigt, mit denen es Apache Flink ermöglicht, auf sein Metriksystem zuzugreifen: per HTTP Request an seine *REST API* oder per Metrik Reporter. Die Wahl für diese Arbeit fiel auf den Metrik Reporter. Hauptsächlich dafür ausschlaggebend war erneut, eine hohe Wiederverwendbarkeit des Evaluationstools zu gewährleisten.

Ein Metrik Reporter trägt alleine durch seine Konzeption schon dazu bei. Als *Plugin* arbeitet er losgelöst vom zu testenden Operator und kann sogar Bibliotheken verwenden, die in Konflikt mit denen des Operators stehen. Er braucht, anders als das HTTP Request, mit welchem die Metriken unter *http://localhost:8081/jobs/<id>/metrics* angefragt werden, keine TaskManager oder Subtasks zu kennen. Stattdessen arbeitet automatisch eine Instanz pro Task Manager, welcher die Metriken bei deren Registrierung über die Methode

```
1 notifyOfAddedMetric(Metric metric, String metricName, MetricGroup metricGroup)
```

Code 4: notifyOfAddedMetric

übergeben werden. Zu guter Letzt spricht für einen Metrik Reporter, dass, im Kontext dieser Arbeit, eine regelmäßige Ausgabe der Performance Evaluation ausreicht und keine Anfrage durch

ein bestimmtes Ereignis getriggert werden muss. Genau diese Funktionalität erfüllt ein Metrik Reporter, welcher das *Scheduled* Interface implementiert.

Die Ausgabe der Metriken soll in Form einer Datei im *CSV* Format erfolgen, in welcher der Verlauf der gemessenen Werte über die Zeit ersichtlich ist. Im Header der Datei sollen dabei die wichtigsten Rahmenbedingungen der Evaluation, wie z.B. Datum der Ausführung und deren Verteilungsgrad, abgebildet werden. Im *Body* sollen anschließend die gemessenen Werte aufgelistet werden. Benutzt man einen der vorimplementierten Metrik Reporter, werden einfach alle registrierten Metriken, inklusive der vielen Systemmetriken von Apache Flink ausgegeben. Es muss eine Implementierung entwickelt werden, die nur die vom Nutzer gewünschten Metriken in übersichtlicher und zweckmäßiger Form ausgibt.

5. Implementierung

Auf die in diesem Kapitel beschriebenen Implementierungen kann über die CD im Anhang A zugegriffen werden.

5.1. CSV Metrik Reporter

Um einen ersten Eindruck von Apache Flinks Metriksystem zu gewinnen, wurde zunächst der vorimplementierte JMX Metrik Reporter in einen ersten Prototypen eingebunden, welcher eine einfache Pipeline evaluierte, die zufällig generierte *Long* Werte so filterte, dass nur gerade Zahlen weitergeleitet wurden. Um die ausgegeben Metriken zu visualieren, wurde die *GUI* von *VisualVM* benutzt [33]. Da dies jedoch viele Limitierungen offenbarte, galt es im nächsten Schritt, einen eigenen Metrik Reporter zu schreiben. Dafür musste das Interface *org.apache.flink.metrics.reporter.MetricReporter* implementiert werden. Damit die vom Metrik Reporter gemessenen Werte regelmäßig ausgegeben werden, wurde zusätzlich noch *Scheduled* eingebunden. Beide Interfaces sind im Codebeispiel 5 dargestellt.

```
1  public interface MetricReporter {
2
3      void open(MetricConfig var1);
4
5      void close();
6
7      void notifyOfAddedMetric(Metric var1, String var2, MetricGroup var3);
8
9      void notifyOfRemovedMetric(Metric var1, String var2, MetricGroup var3);
10 }
11
12 public interface Scheduled {
13     void report();
14 }
```

Code 5: MetricRepoter [34] und Scheduled [35] Interface

Wird der Metrik Reporter beim Aufsetzen eines Flink Clusters gestartet, wird zunächst die Methode *open()* ausgeführt. Dabei werden aus der als Parameter mitgegebenen *MetricConfig* der eingestellte Verteilungsgrad und die zusätzlich vom Nutzer registrierten Metriken ausgelesen und gespeichert. Da in der *MetricConfig* nur *Key-Value* Paare als *Strings* abgelegt werden können, stehen die nutzerdefinierten Metriken vorerst nicht als Objekte, sondern lediglich ihre Namen und Typen, also *Counter*, *Meter*, usw., zur Verfügung. Diese Informationen reichen jedoch aus, um später auf die jeweiligen Objekte zugreifen zu können. Bis dahin werden sie in einer *HashMap*, mit dem Metrik-Titel als *Key* und ihrem Metrik-Typ als *Value* abgelegt.

Die Methode *notifyOfAddedMetric()* spielt eine der wichtigsten Rollen in der Funktionsweise des CSV Reporters. Sie wird immer dann aufgerufen, wenn im Cluster eine neue Metrik registriert

wurde. Wie in Codebeispiel 5 zu sehen, erhält man durch ihre Parameter nun auch Zugriff auf das betreffende Metrik Objekt, bzw. bei Bedarf auch auf die gesamte *MetricGroup*.

Wird durch die Registrierung einer neuen Metrik *notifyOfAddedMetric()* aufgerufen, wird zuerst anhand des Namens, übergeben durch den Parameter *var2*, geprüft, ob sie zu den zu messenden Metriken gehört. Heißt die neu registrierte Metrik *latency*, handelt es sich um eines der *Histogramme*, welches die Latenzen der Latency Marker speichert. Da die Granularität der Marker in dieser Arbeit fest auf *single* eingestellt ist, wird für jeden Suboperator im Flink Cluster ein eigenes *Histogram* angelegt. Da in komplexen Analyse Pipelines sehr viele Suboperatoren existieren können, würde es den Rahmen sprengen, die Latenz jedes Einzelnen auszugeben. Stattdessen wird jedes *Histogram* zusammen mit seinem Scope in einer *HashMap* gespeichert, um damit zum Schluss die durchschnittliche Latenz eines Suboperators bestimmen zu können.

Handelt es sich bei der eingehenden Metrik dem Namen nach um den Throughput, wird sie als Attribut des Typs *Meter* gespeichert.

Kann die Metrik weder der Latenz, noch dem Throughput zugeordnet werden, wird ihr Name mit denen der zusätzlichen Metriken verglichen. Kommt es zu einem Treffer, wird das dazugehörige Metrik Objekt, je nach seinem Typ in einer der als Attribute initialisierten *HashMaps*, *addCounter*, *addMeter*, oder *addHistogram*, als *Value* abgelegt. Den *Key* bildet der jeweilige Name der Metrik. Kommt es zu keinem Treffer, passiert nichts.

Diese Herangehensweise wurde von der Implementierung des *AbstractReporter* Objekts, welches Apache Flink mitliefert [36], abgeleitet.

Die Methode *notifyOfRemovedMetric()* entspricht der Umkehrung von *notifyOfAddedMetric()*. Wird ein Flink Job gecancelt, wird sie für jede betroffene Metrik einmal ausgeführt und die betroffene Metrik aus dem Attribut, in dem sie gespeichert wurde, gelöscht.

Beim ersten Aufruf der Methode *report()* wird eine neue Datei im *CSV* Format erstellt und ein *BufferedWriter* initialisiert. In einer frühen Version des Prototypen wurde noch jeder Report, mit seinen Metriken in der linken Spalte und deren Werten in den rechten Spalten, in jeweils einer eigenen Datei gespeichert. Da dies jedoch die Auswertung erschwerte, wurde der Metrik Reporter so umgestellt, dass es nun nur noch eine Datei gibt, die in Intervallen aktualisiert wird, sodass der Verlauf der Messungen ersichtlich ist.

Nach Erstellung der Datei wird der *Header* des Reports geschrieben, welcher sich aus dem aktuellen Datum inklusive Uhrzeit in der ersten Zeile, der Angabe über die Parallelität in der zweiten Zeile, der Angabe über den Scope in der dritten Zeile und dem Tabellenkopf in der vierten Zeile zusammensetzt.

Der Tabellenkopf wiederum besteht aus den Titeln der gemessen Metriken. Dazu gehörten einerseits die Standardwerte *Time* (Zeit in Sekunden), *Throughput*, *Throughput(s)*, sowie *MeanLatencyOfSubtask*, welche von den Latency Markern stammt. Andererseits muss, aufgrund der von Test zu Test schwankenden Zahl an nutzerdefinierten Metriken, der Tabellenkopf teilweise dynamisch erstellt werden. Dies geschieht, indem in einer *for*-Schleife alle in der *HashMap* abgelegten *Key-Value* Paare durchlaufen werden und für jeden *Counter* eine Spalte (Anzahl), für jedes *Meter* zwei (Anzahl,

Throughput/s) und für jedes *Histogram* vier Spalten (*Min*, *Max*, *Mean* und *Last*) dem Tabellenkopf hinzugefügt werden.

Auf die Einbindung von zusätzlichen Metriken des Typs *Gauge* wurde verzichtet, da diese ein Objekt von beliebiger Klasse darstellen können und sich daher unmöglich vorhersagen lässt, wie eine angemessene Darstellung aussehen könnte.

Darüber hinaus werden bei jedem weiteren Aufruf von *report()* die aktuellen Werte der Metriken ausgelesen und in die jeweiligen Spalten der Tabelle eingetragen. Jeder Aufruf erzeugt folglich genau eine neue Zeile. In welchen Zeitabständen die Methode aufgerufen wird, ist in der Config unter *metrics.reporter.<name>.interval:* festgelegt.

Mit der *close()* Methode wird der initialisierte *BufferedWriter* geschlossen. Bei anderen Metrik Reportern, die sich mit Datenbanken oder ähnlichem verbinden, kann diese Methode genutzt werden, um diese Verbindungen wieder zu trennen.

Im Codebeispiel 6 ist zur Veranschaulichung und einem besseren Verständnis der Inhalt eines Metrikreports abgebildet. Das Reportinterval war in diesem Fall auf fünf Sekunden eingestellt.

```
1  Report generated on 24-01-2023 at 12-58-26
2  Parallelism: 2
3  Time;#StreamTriple;Throughput(s);#LatencyMarkers;Latency(Mean);Latency(Last)
4  12-58-31;0;0;0;0;0
5  12-58-36;0;0;0;0;0
6  12-58-41;1;0.0;1;30.0;30
7  12-58-46;245;0.0;5;6.0;0
8  12-58-51;579;9.566666666666666;10;3.0;0
9  12-58-57;1005;14.583333333333334;13;2.3076923076923075;0
10 12-59-02;1170;19.5;14;2.142857142857143;0
11 12-59-07;1603;25.083333333333332;18;1.6666666666666665;0
12 12-59-12;2078;30.483333333333334;19;1.5789473684210524;0
13 12-59-17;2671;41.0;26;1.1538461538461535;0
14 12-59-22;2908;48.45;26;1.1538461538461535;0
15 12-59-27;3135;50.516666666666666;26;1.1538461538461535;0
```

Code 6: Beispiel eines Reports des CSV Metrik Reporters

5.2. Source Functions

5.2.1. RandomStreamTripleSource

Die *RandomStreamTripleSource* ist die Implementierung der im Kapitel 4.2.2 konzipierten generischen Datenquelle. Die Source Function erweitert die abstrakte Klasse *RichParallelSourceFunctionWithCap*, welche wiederum die abstrakte Klasse *RichParallelSourceFunction* erweitert, wodurch sie einerseits Zugriff auf das Objekt *RuntimeContext* erhält und andererseits verteilt ausgeführt werden kann. Somit wird die Datengenerierung nicht zum Flaschenhals, sondern kann gemeinsam mit dem Verteilungsgrad der zu testenden Anwendung skalieren. Das Anlegen der erweiternden abstrakten

Klasse *RichParallelSourceFunctionWithCap* war wiederum nötig, um die Source Functions, die von ihr erben, um die Methode *resetOutputCount()* zu ergänzen, die von einem Timer getriggert wird. Auf die Frage, wozu ein Timer nötig ist, wird im Laufe dieses Kapitels eingegangen.

Die Klasse *RandomStreamTripleSource* umfasst insgesamt 11 verschiedene Attribute, die sich nach Art ihrer Verwendung in fünf verschiedene Kategorien unterteilen lassen. Der boolean Wert *cancelled* gehört zu den Attributen, die die Funktionsweise der Klasse sicherstellen und wird zu Beginn auf *false* gesetzt. Durch das Aufrufen der Methode *cancel()* wird ihr Wert auf *true* geändert und die Source Function hört auf, *StreamTriple* zu produzieren. Die Listen *sources* und *labels*, sowie der *idCounter* werden zur Datengenierung benötigt. In den Listen werden alle möglichen Label der Knoten (sources) und Kanten (labels) gespeichert, aus denen bei Erstellung der StreamTriple per Zufall Werte gezogen werden. Der *idCounter* hingegen, inkrementiert pro erstelltem Objekt um eins, sodass jeder Kante eine einzigartige ID zugewiesen wird. Zur Kategorie der Metriken gehört das *Meter* "throughput". Dieses wird in der *open()* Methode registriert und zählt für jedes ausgegebene *StreamTriple* um eins hoch. Da Meter lediglich ein Interface ist, wurde noch eine geeignete Implementierung benötigt. Die Wahl fiel dabei auf die von Apache Flink mitgelieferte Klasse *MeterView*. Dieses implementiert das Interface *View*, um den Throughput pro Sekunde zu messen und benötigt als Parameter im Konstruktor einen *Counter*. Die Wahl für diesen fiel wiederum auf den *SimpleCounter*, der ebenfalls standardmäßig bei Apache Flink dabei ist.

Zur nächsten Kategorie gehören drei Attribute, die zur Erstellung der *Timestamps* der *StreamTriple* benutzt werden. Die ersten Beiden sind *Long* Werte, die den Beginn (start) und das Ende (end) des Wertebereichs festlegen, aus welchem die zufällig generierten *Timestamps* stammen sollen. Die *RandomStreamTripleSource* stellt dem Nutzer insgesamt acht verschiedene Konstruktoren zur Verfügung, um die Datengenerierung den individuellen Anforderungen so gut wie möglich anzupassen. Beispielsweise kann man wählen, ob sich die *Timestamps* innerhalb eines gegebenen Intervalls befinden (default: Jahr 2021), oder sie den Zeitpunkt ihrer Erstellung darstellen sollen. Gesteuert wird das Ganze über das dritte Attribut der Kategorie: *realTimestamp* vom Typ *boolean*. Auf *false* gesetzt, kann man den Wertebereich selbst wählen, während auf *true*, der *Timestamp* auf den Zeitpunkt der Erstellung des Objekts fällt.

Die Attribute der letzten Kategorie werden im Zusammenhang mit der Ausgabebegrenzung (Cap) der *SourceFunction* benötigt. Das Ganze funktioniert über einen *Timer*, der jede Sekunde die Methode *resetOutputCount()* aufruft, welche den Wert *outputCount* auf null zurücksetzt. Hochgezählt wird *outputCount* dagegen durch jedes erstellte *StreamTriple*. Erreicht der Counter den Wert des *outputCapPerSeconds*, welcher wiederum als Parameter im Konstruktor der *SourceFunction* angegeben werden kann, werden bis der Timer wieder abgelaufen ist, keine *StreamTriple* mehr erstellt. Wird der *outputCapPerSeconds* auf null gesetzt, gibt es keine Ausgabebegrenzung. Die Ausgabebegrenzung wurde auf pro Sekunde und nicht etwa auf pro Minute festgelegt, da der Throughput der Anwendung ebenfalls pro Sekunde bestimmt wird.

Kommen wir nun zu den Methoden der *SourceFunction*. Wie bereits erwähnt, gibt es acht verschiedene Konstruktoren, die es dem Nutzer ermöglichen sollen, Instanzen der Klasse anwendungsbezogen, sowie möglichst unkompliziert erstellen zu können. So werden beispielsweise bei einem Konstruktor die Labels der Knoten als *String-Array* verlangt, bei einem Anderen kann man sie

als Liste übergeben. Gleiches gilt für die Eingrenzung des Wertebereichs der *Timestamps*. Start- und Endzeitpunkt können entweder als Instanzen vom Typ *Long*, oder als *Timestamps* übergeben werden. Andere Konstruktoren nehmen einem die Entscheidung über die Wertebereiche ab, indem sie default Werte, wie z. B. *A, B, ..., Z* für Knotenlabels, oder a,b,c als mögliche Labels für die Katen übergeben. So gibt es beispielsweise sogar einen Konstruktor, der ohne jede Parameter aufgerufen werden kann, oder einen, bei dem lediglich die Ausgabebegrenzung angegeben werden muss. Dies soll es ermöglichen, das Evaluationstool dieser Arbeit auch ohne viel Vorbereitung bzw. Einlesezeit zu nutzen. Für welchen Weg der Entwickler sich auch entscheiden mag, am Ende rufen alle Konstruktoren den Hauptkonstruktor auf, wie er in Codebeispiel 7 abgebildet ist.

```
1  public RandomStreamTripleSource(List<String> labels, List<String> sources, Long
       start, Long end, int oCap, boolean realTimestamp) {
2
3    idCounter = 0;
4    outputCapPerSeconds = oCap;
5    if (!realTimestamp) {
6      this.labels = new ArrayList<>(labels);
7      this.sources = new ArrayList<>(sources);
8      this.start = start;
9      this.end = end;
10     this.realTimestamp = false;
11   }
12   else {
13     this.labels = new ArrayList<>(labels);
14     this.sources = new ArrayList<>(sources);
15     this.realTimestamp = true;
16   }
17 }
```

Code 7: Hauptkonstruktor der RandomStreamTripleSource

Die Methoden *open()* und *cancel()* wurden ja bereits in den Ausführungen über die Attribute der Source Function analysiert, weshalb nun nur noch die Funktionsweise von *run(SourceContext ctx)* zu klären ist. An deren Anfang steht eine *if*-Klausel, die prüft, ob eine Ausgabebegrenzung initialisiert wurde. Ist dem so, wird solange der Boolean *cannceled* nicht auf true setzt wird, eine *while* Schleife ausgeführt, welche *synchronized StreamTriple* erstellt und dabei jedes Mal den Counter der Metrik Throughput um eins erhöht. Ist dagegen eine Ausgabebegrenzung eingestellt, wird vorher noch abgefragt, ob der *outputCount* kleiner als der *outputCapPerSeconds* ist. Außerdem muss in diesem Fall neben dem Throughput auch der *outputCount* um eins erhöht werden. Die Anwendung von *synchronized* wird von Apache Flink empfohlen, um keine Fehler durch das asynchrone Arbeiten mehrerer Instanzen der *SourceFunction* zu erhalten. Bei Verteilungsgraden größer als eins kann dadurch z. B. verhindert werden, dass zwei unterschiedliche StreamTriple die gleiche ID zugeteilt bekommen, oder die Ausgabebegrenzung überschritten wird, weil der *outputCount* noch eins niedriger als der Cap war, woraufhin zwei Threads jeweils noch ein *StreamTriple* erschaffen.

Die Generierung der *StreamTriple* erfolgt durch eine ausgelagerte Hilfsfunktion namens *createTriple()*, die ein *StreamTriple* zurückgibt und keine Parameter benötigt. Dazu legt sie zuerst eine siebenstellige ID an. Als Wert dient dazu der *idCounter*, welcher anschließend um eins inkrementiert

wird. Danach wird, je nachdem ob *realTimestamp* true gesetzt wurde oder nicht, ein Timestamp aus der aktuellen Systemzeit, oder über die Formel in Codebeispiel 8, zufällig innerhalb des gesetzten Wertebereichs erstellt.

```
1 Timestamp time;
2 long diff = end - start + 1;
3 time = new Timestamp(start + (long) (Math.random() * diff));
```

Code 8: Generiung eines zufälligen Timestamps aus dem Wertebereich zwischen *start* und *end*

Danach werden aus den bereits erwähnten Listen *sources* und *labels* zufällige Einträge gewählt, wobei darauf geachtet wurde, dass Ziel und Endnoten unterschiedliche Einträge haben müssen. Die *SourceFunction* erstellt also keine Schleifen-Kanten. Für die *Properties* werden über *Properties.create()* lediglich leere Hüllen erzeugt. Die Notwendigkeit der Erstellung von realistischeren *StreamTriplen* leitet zum nächsten Punkt dieser Arbeit: der Implementierung der *RealStreamTripleSource*.

5.2.2. RealStreamTripleSource

Für die *RealStreamTripleSource* sind im Gegensatz zur generischen Datenquelle keine Attribute zur Steuerung der Datengeneration nötig. Folglich gibt es auch nur einen Konstruktor, der als Parameter lediglich eine Liste von *StreamTriplen*, einen *Boolean* und ein *int* verlangt. Die Liste sollte die *StreamTriple* beinhalten, die letzlich gestreamt werden sollen. Ähnlich wie der Flink Connector *.fromCollection(...)* es verlangt. Über den *Boolean* wird die Reihenfolge gesteuert, mit der die StreamTriple gelesen werden. Auf *true* gesetzt, ist die Reihenfolge zufällig, während auf *false* gesetzt, die Liste per Index aufsteigend iteriert wird. Mit dem *int* kann, genauso wie bei der *RandomStreamTripleSource*, die Ausgabebegrenzung pro Sekunde festgelegt werden. Auf null gesetzt, gibt es auch bei dieser Source keine Ausgabebegrenzung.

Im Konstruktor wird die übergebene Liste einmal als Attribut *workingList* abgelegt und einmal in eine neue, zweite Liste *list*, die als Backup fungiert, kopiert. Arbeitet die *SourceFunction* wird bei jedem Durchlauf der *while-Schleife* innerhalb der *run()* Methode, ein *StreamTriple* aus der *workingList* entnommen, weitergegeben und anschließend aus der Liste entfernt. Ist die Liste schließlich leer, wird sie über die zweite Liste, die als Backup angelegt wurde, wieder aufgefüllt. Dadurch wird sichergestellt, das dem *System Under Test* permanent Daten geliefert werden und die Anwendung nicht, wie etwa im Batch Processing, endet.

Da es den Operator verwirren könnte, wenn die *Timestamps* der neu ankommenden *StreamTriple* plötzlich wieder in der Vergangenheit liegen, wurde implementiert, dass sich die *Timestamps* mit jedem Wiederauffüllen der *workingList* neu berechnen.

Formel 1 (Neuberechnung der Timestamps)**.** *Der neu berechnete Timestamp t'_n des StreamTriple T_n an Position $n \in \mathbb{N}$ in der Liste L ergibt sich wie folgt:*

$$t'_n = t_{end} + (t_n - t_{start}),$$

wobei

- t_n *der alte Timestamp von* T_n *ist,*
- t_{end} *der Timestamp des letzten StreamTriple aus L ist und*
- t_{start} *der Timestamp des ersten StreamTriple aus L ist.*

Diese Neuberechnung ermöglicht einen Stream mit realen Daten und kontinuierlich zunehmenden *Timestamps*. Das Verhältnis, bzw. der zeitliche Abstand zwischen den *Timestamps* bleibt dabei erhalten. Die Variablen t_{start} und t_{end} werden als Attribute der Klasse einmal im Konstruktor initialisiert und danach zum Abschluss jeder Neuberechnung aktualisiert.

Bei der Implementierung der Source musste zudem beachtet werden, dass bei einem Verteilungsgrad > 1 mehrere Instanzen der *RealStreamTripleSource* erschaffen werden. So befinden sich kritische Aufrufe wie in Codebeispiel 9 innerhalb eines *synchronized()* Blocks, sodass sie zu jeder Zeit nur von jeweils einem Thread ausgeführt werden können. Dies verhindert unter anderem, dass ein Index von *workingList* gezogen werden kann, der gar nicht mehr existiert, weil bereits eine andere Instanz der Source Objekte aus der Liste entfernt hat. Außerdem besitzt nicht jede Source-Instanz seine eigene *workingList*, sondern jede greift auf eine Referenz der gleichen Speicheradresse zurück.

```
1 synchronized (ctx.getCheckpointLock()) {
2   int r = new Random().nextInt(workingList.size());
3   StreamTriple graph = workingList.get(r);
4   ctx.collect(graph);
5   workingList.remove(r);
6 }
```

Code 9: *synchronized()* Block

Die restlichen Methoden der Source verhalten sich wie bei der *RandomStreamTripleSource*. Über einen *Timer* wird sichergestellt, dass die Ausgabebegrenzung eingehalten wird. In der *open()* Methode wird die Metrik für den Throughput registriert, während *cancel()* den *Boolean cancelled* auf *true* setzt und damit die *while* Schleife in *run()* abbricht.

Die *RealStreamTripleSource* wurde so implementiert, dass es irrelevant ist, woher die Liste von *StreamTriplen* kommt. Sie kann beispielsweise der Output einer vorangestellten Streaming Pipeline, oder eines Datengenerators sein. Im Fall dieser Arbeit wurde ein Datensatz des New York Citi Bike Services eingelesen, mit Hilfe eines eigens dafür geschriebenen *Parsers* in einzelne *StreamTriple* übersetzt und in einer *ArrayList<>* zusammengeführt.

Wie in Abbildung 5.1 dargestellt, wird dem *Parser* im Konstruktor der Pfad zum Datensatz übergeben. Über die Methode *read()* kann der Datensatz anschließend ausgelesen werden, wobei die gewonnenen Informationen zunächst als Objekte der Klasse *CitiBikeTripData* gespeichert werden. Erst durch den anschließenden Aufruf von map(), erhält man schließlich die gewünschte Liste mit *StreamTriplen*. Die Entscheidung das Auslesen der Daten von ihrer Interpretation zu trennen, beruht auf der Heterogenität des verwendeten Datensatzes.

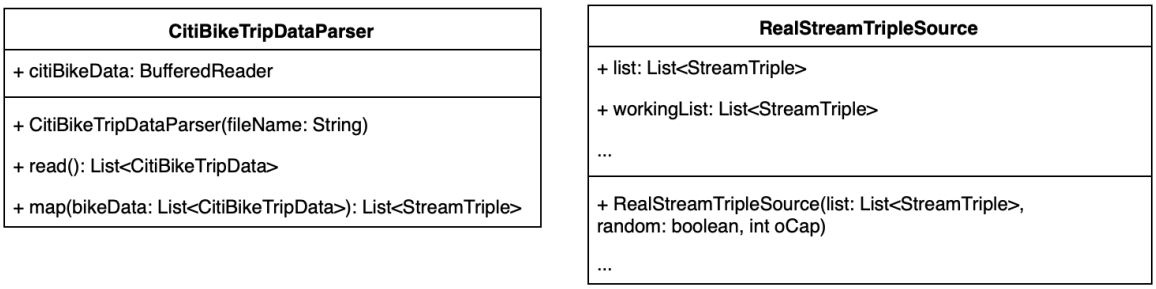

Abbildung 5.1.: Modelle des Parsers und der RealStreamTripleSource

Der größte Unterschied zur *RandomStreamTripleSource* besteht darin, dass die von der *RealStreamTripleSource* gestreamten Objekte *Properties* besitzen können. So haben beispielsweise, die aus dem Citi Bike Datensatz gewonnenen Knoten, welche die Fahrradstationen repräsentieren, Angaben über ihren Längen- (*longitude*) und Breitengrad (*latitude*, während die aus dem Datensatz resultierenden Kanten, welche die Fahrten darstellen, die Eigenschaften *duration*, *usertype*, *birthyear* und *gender*.

5.2.3. GraphStreamBenchmarkSuite

Durch die Implementierung der Klasse *GraphStreamBenchmarkSuite* wurden schließlich alle Programmbausteine zu einer funktionierenden Anwendung zusammengesetzt. Ihre insgesamt sechs Attribute, dargestellt in 5.2, werden über die Parameter des Konstruktors gesetzt. Darüber hinaus besitzt die Klasse nur noch die Methode *execute(String pathName)*, mit welcher die Evaluation gestartet wird. Der Parameter *pathName* übergibt der Anwendung den gewünschten Speicherpfad für die Reports der Instanzen des Metrik Reporters.

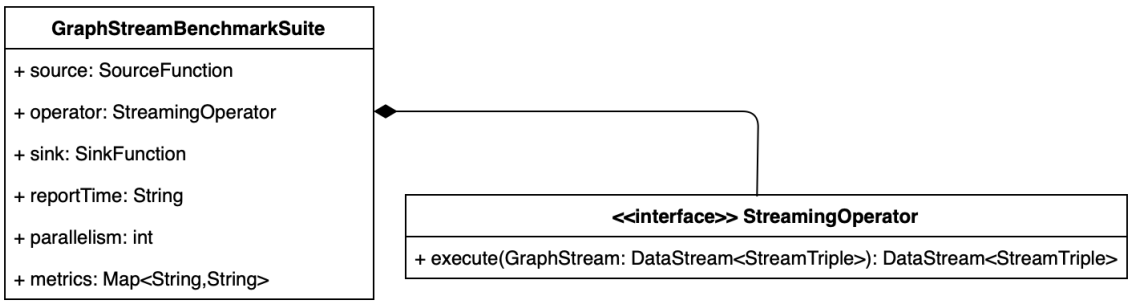

Abbildung 5.2.: Modell der *GraphStreamBenchmarkSuite*

Wie viele Schritte *execute()* umfasst, hängt davon ab, auf welche Form von Cluster die Anwendung laufen soll. Startet man Apache Flink über eine Entwicklungsumgebung, wird ein Flink Minicluster ausgeführt, welches keinen Zugriff auf die Datei *flink-conf.yaml* hat. Stattdessen müssen

Anpassungen des Clusters über ein Objekt der Klasse *Configuration* vorgenommen werden, welches anschließend dem *StreamExecutionEnviroment* übergeben wird. Startet man das Flink Cluster hingegen per *Bash Script*, kann man auf diese Weise keine Änderungen vornehmen, da der Flink Job, welcher das *Configuration* Objekt beinhaltet, erst nach Aufsetzen des Clusters übergeben wird. In diesem Fall muss man etwaige Änderungen zwangsweise in der *flink-conf.yaml* vornehmen. Da die Tests im folgenden Kapitel auf einem klassischen Flink Cluster ausgeführt wurden, wurden die Programmzeilen, welche die Konfiguration des Miniclusters vornahmen, auskommentiert und durch entsprechende Zeilen in der *flink-conf.yaml*, dargestellt in Codebeispiel 10, ersetzt. Dadurch verloren zwar die Attribute *reportTime* und *metrics* ihre Funktion, doch blieben sie trotzdem implementiert, um dem einzelnen Nutzer die freie Wahl zu lassen, welches Cluster er für seine Test benutzen möchte.

```
1  metrics.reporters: csv
2  metrics.reporter.csv.class: org.apache.flink.metrics.csv.CSVReporter
3  metrics.latency.granularity: single
4  metrics.latency.history-size: 150
5  metrics.reporter.csv.interval: 5s
6  metrics.reporter.csv.path: ../output/report
7  metrics.reporter.csv.additional.processingTimeLatency: Histogram
8  metrics.reporter.csv.additional.filteredOut: Counter
```

Code 10: flink-conf.yaml language

Wie man in Codebeispiel 10 sehen kann, betreffen alle gesetzten Konfigurationen Flinks Metriksystem. Es sei anzumerken, dass lediglich die ersten drei Zeilen für jeden Anwendungsfall so zu setzen sind. Durch sie wird die Klasse *CSVReporter* als Metrik Reporter festgelegt und die Granularität der *LatencyMarker* auf *single* eingestellt, was vom *CSVReporter* vorausgesetzt wird. Für die restlichen Konfigurationen können die Werte beliebig angepasst werden, wobei die Zeilen sieben und acht komplett optional sind. Sie sollen hier lediglich als Beispiel dafür dienen, wie man dem Metrik Report weitere Metriken hinzufügt. In diesem Fall wären dies ein *Histogram* namens *processingTimeLatency* und ein *Counter* namens *filteredOut*.

Die nächsten Schritte in *execute()* sind das Laden des *StreamExecutionEnvironments*, das Setzen des *RuntimeExecutionMode* auf *STREAMING* und das Anpassen der *Execution* Parameter. Dazu gehören der vom Nutzer gewünschte Verteilungsgrad, sowie das *LatencyTrackingInterval*, welches fest auf 1000 Millisekunden gesetzt wird. Es wird also jede Sekunde ein *LatencyMarker* emittiert.

Nachdem all diese notwendigen Einstellungen getroffen wurden, wird schließlich die Pipeline der Evaluation definiert. Dabei wird als Erstes ein *DataStream* erzeugt, in dem die im Konstruktor übergebene *SourceFunction* dem *StreamExecutionEnvironment* zugeteilt wird. Damit das *System Under Test* eingebunden werden kann, muss dieses zunächst das Interface *StreamingOperator* implementieren, welches wiederum die Einbettung einer einzigen Methode verlangt:

```
1  public DataStream<StreamTriple> execute(DataStream<StreamTriple> graphStream)
```

Code 11: *execute()* Methode des *StreamingOperatos*

Diesen Schritt muss der Nutzer der Evaluationssoftware selbst erledigen. Aus diesem Grund wurde die Einbindung so unkompliziert wie nötig gehalten. Der *execute()* Methode des *StreamingOperators* wird ein *DataStream* aus *StreamTriplen* übergeben und als *return* wird wieder ein eben solcher erwartet. Dazwischen können alle möglichen Transformationen stattfinden, die Apache Flinks *DataStream API* zulässt.

Bei der Einbindung des Operators kann es vorkommen, dass die vom Benchmark kommenden *StreamTriple* nicht als solche erkannt werden, da sie in einem anderen Paket liegen, als die *StreamTriple* des Operators. Für diesen Fall werden zwei *Map Functions* bereitgestellt, die ein neues Objekt des jeweils anderen Typs erstellen und anschließend alle Informationen übertragen. Die Mapper liegen unter dem Pfad *src/main/java/edu/dbsleipzig/benchmark/application/functions/mapper* und werden zum Zeitpunkt der Fertigstellung dieser Arbeit dafür genutzt, die StreamTriple des *GraphStreamZoomers* mit denen der Evaluationssoftware zu mappen. Damit der Nutzer sie für sein eigenes System verwenden kann, muss lediglich der Pfad der Operator internen *StreamTriple* mit dem Pfad zu denen des *GraphStreamZoomers* ausgetauscht werden.

Die Pipeline endet in der im Konstruktor übergebenen *SinkFuntion*. Für die Evaluation des *GraphStreamZoomers* wurde die von Apache Flink bereitgestellte *DiscardingSink* verwendet, da diese die Performance der Pipeline nicht einschränkte. Für den Fall, dass die Ausgabe des Operators anschließend ebenfalls evaluiert werden soll, sind jedoch auch andere *SinkFunctions* denkbar.

5.2.4. Main

Der Prototyp dieser Arbeit umfasst zum Zeitpunkt seiner Fertigstellung drei Klassen mit einer *main()* Methode, die alle einen relativ ähnlichen Aufbau teilen. Die Klassen *GenericDataExecution* und *RealDataExecution* sollen dem Nutzer als Einstieg in die Anwendung dienen. Beide evaluieren Testoperatoren, die lediglich einen einfachen Filter anwenden. Die *GenericDataExecution* initialisiert dafür eine *RandomStreamTripleSource*, während die *RealDataExecution* eine *RealStreamTripleSource* erzeugt. Die dritte Klasse mit einer *main* Methode heißt *GraphStreamZoomerTest* und wurde benutzt, um die verschiedenen Tests des nächsten Kapitels durchzuführen.

Es wird empfohlen, aufbauend auf den bestehenden Implementierungen für jeden Test eine eigene *main()* zu schreiben. Trotzdem erlauben es alle drei Klassen, über die Parameter der Kommandozeile (*String[] args*) verschiedene Einstellungen vorzunehmen, sodass nicht für jede Evaluation Code verändert werden muss.

Syntax 2 (Kommandozeilenparameter)**.** *Folgende Einstellungen sind über die Kommandozeile zu treffen:*

- *args[0] : Namen der zusätzlichen Metriken*

  Bsp. processingTimeLatency,filteredOut

- *args[1] : Typen der zusätzlichen Metriken*

  Bsp. Histogram,Counter

- *args[2] : Verteilungsgrad*

  Bsp. 72

- *args[3] : Label für die Kanten | Pfad zum Datensatz*

  Bsp. a,b,c | ../input/201306-citibike-tripdata.csv

- *args[4] : Label für die Knoten | Reihenfolge oder zufällig*

  Bsp. A,B,C,D,E,F,G,H | false

- *args[5] : Beginn des Wertebereichs zur Timestamp Erstellung | -*

  Bsp. 0L (Angabe als Long)

- *args[6] : Ende des Wertebereichs zur Timestamp Erstellung | -*

  Bsp. 1000000L (Angabe als Long)

- *args[7] : Ausgabebegrenzung pro Sekunde*

  Bsp. 1000 (0 für Stresstest)

- *args[8] : Setzen der Timestamps auf Systemzeit | -*

  Bsp. true (Kommandozeilen Parameter 5 und 6 werden ignoriert)

- *args[9] : Speicherpfad des Reports*

  Bsp. null (Nur bei Nutzung eines Miniclusters anzugeben)

Ein Trennstrich (|) in der Darstellung verbildlicht, dass der Parameter, je nachdem welche *Source-Functions* verwendet wird, anders besetzt ist. Möchte der Nutzer eine bestimmte Einstellung nicht vornehmen, sollte er den jeweiligen Parameter auf *null* setzen.

6. Evaluation

Um die Funktionsweise des Prototypen der Evaluationssoftware zu prüfen, wurden drei verschiedene Performancetests am Beispiel des *GraphStreamZoomers* durchgeführt. Dafür wurde Apache Flink auf dem Athena Cluster der Uni Leipzig ausgeführt, welches insgesamt 16 Maschinen umfasst, wobei jede mit 376 Gbyte RAM, einer Intel(R) Xeon(R) Gold 6240 CPU @ 2.60GHz mit 72 Cores, sowie 11TB SSD Festplattenspeicher ausgestattet ist. Alle Maschinen sind mit einer Datenübertragungsrate von 1Gbit/s verbunden. Auf jeder Maschine läuft ein Task Manager, der seine Tasks bis zu einem Verteilungsgrad von 36 parallelisieren kann. Auf das gesamte Cluster übertragen, kann mit einem Verteilungsgrad von 576 gearbeitet werden.

6.1. Stresstest unter verschiedenen Verteilungsgraden

Beim ersten Test wurde evaluiert, wie sich der Throughput des Operators bei Erhöhung des Verteilungsgrads verhält. Als Datengenerator wurde die *RandomStreamTripleSource* ohne Ausgabebegrenzung verwendet. Es handelte sich also um einen Stresstest. Der *GraphStreamZoomer* wurde mit einer *WindowSize* von 10 Sekunden konfiguriert und der *GroupingKey* für Knoten und Kanten waren ihre *Labels*. Insgesamt wurde der Flink Job fünf mal ausgeführt, wobei die Verteilungs des Flink Cluster jedes mal um 36 erhöht wurde, was dem System bei jedem Durchgang eine weitere physische Maschine hinzufügte. Bei jedem Test wurde dem Cluster eine Minute Zeit gegeben, um sich einzulaufen, bevor die Metriken in die Evaluation einbezogen wurden.

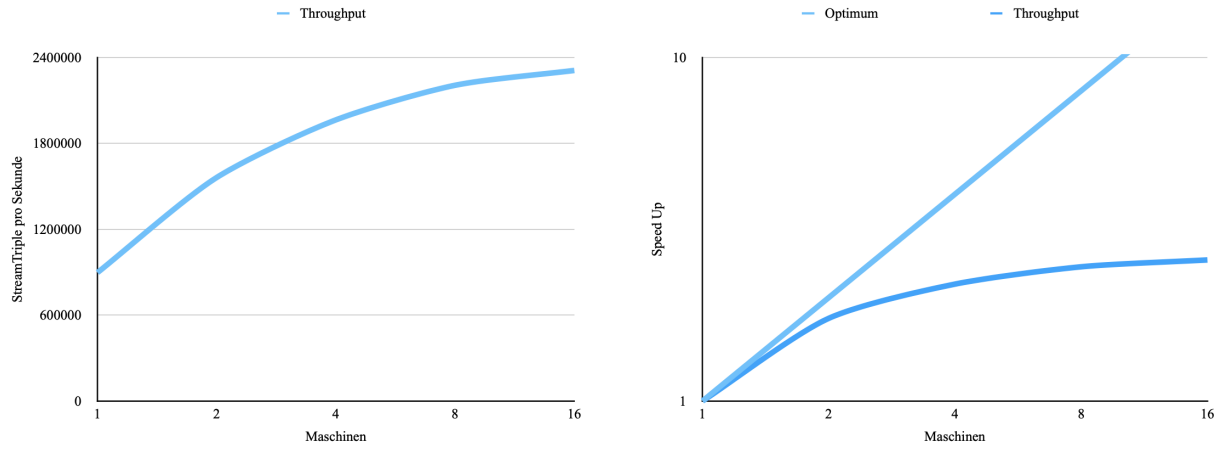

Abbildung 6.1.: Auswertung des ersten Tests

Es ist zu beobachten, dass der Throughput mit Zunahme des Verteilungsgrads ansteigt, der Anstieg jedoch immer weiter abflacht. Bei der niedrigsten Verteilung konnte der Operator 897.156 *StreamTriple* pro Sekunde verarbeiten. Bei maximaler Verteilung des Clusters lag der Wert bei 2.309.184 *StreamTriplen* pro Sekunde, was einer Steigerung um 257% entspricht.

6.2. Messung der Latency unter verschiedenen Verteilungsgraden

Beim zweiten Test wurde bei der verwendeten *SourceFunction* eine Ausgabebegrenzung von 1000 Objekten pro Sekunde eingestellt. Ansonsten galten die gleichen Konfigurationen wie im ersten Test. Da für jede weitere Verteilung des Clusters, auch eine Instanz der *SourceFunction* erstellt wird, werden bei maximaler Verteilung insgesamt 576.000 *StreamTriple* pro Sekunde erstellt. Im ersten Durchlauf gab es einen Verteilungsgrad von 36, was bedeutet, dass nur 36.000 *StreamTriple* pro Sekunde erzeugt wurden.

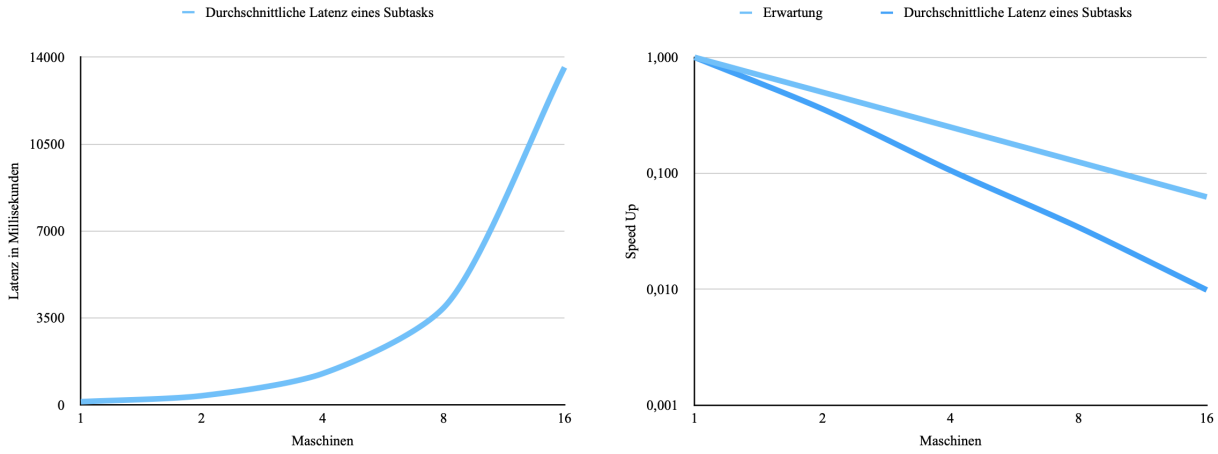

Abbildung 6.2.: Auswertung des zweiten Tests

Es ist zu beobachten, dass die Latenz im vierten und fünften Durchlauf stark anstieg. Es existiert also eine Grenze, ab welcher das *System Under Test* steigenden Dateneingang, nicht mehr durch eine Erhöhung der Verteilung kompensieren kann, ohne das die durchschnittliche Latenz pro Subtask deutlich ansteigt. Während die Latenz bei einem Verteilungsgrad von 144 nur eine Erhöhung der Latenz um das neunfache verursachte, stieg sie bei maximaler Verteilung um das 101-fache an. Es sei erwähnt, dass selbst beim letzten Durchgang, die Latenz nicht kontinuierlich stieg, sondern mit der Zeit sogar immer weiter abnahm. Der Sustainable Throughput des Operators wurde also nicht überschritten.

6.3. Messung von Latency und Throughput mit realen Daten

Die Messungen des letzten Tests wurden mit der *RealStreamTripleSource* und einem Datensatz des New York Citi Bike Services durchgeführt [37]. Es gab keine Ausgabebegrenzung, die Reihenfolge der *StreamTriple* wurde eingehalten und es gab keine Verteilung des Clusters. Die *WindowSize* des *GraphStreamZoomer* wurde auf zwei Sekunden eingestellt. Zusätzlich zu den *GroupingKeys* der letzten Tests, wurde noch *gender* als *Key* für das *Grouping* der Kanten hinzugenommen. Außerdem wurde eine *EdgeAggregateFunction* auf die Leihdauer der Fahrräder angewandt.

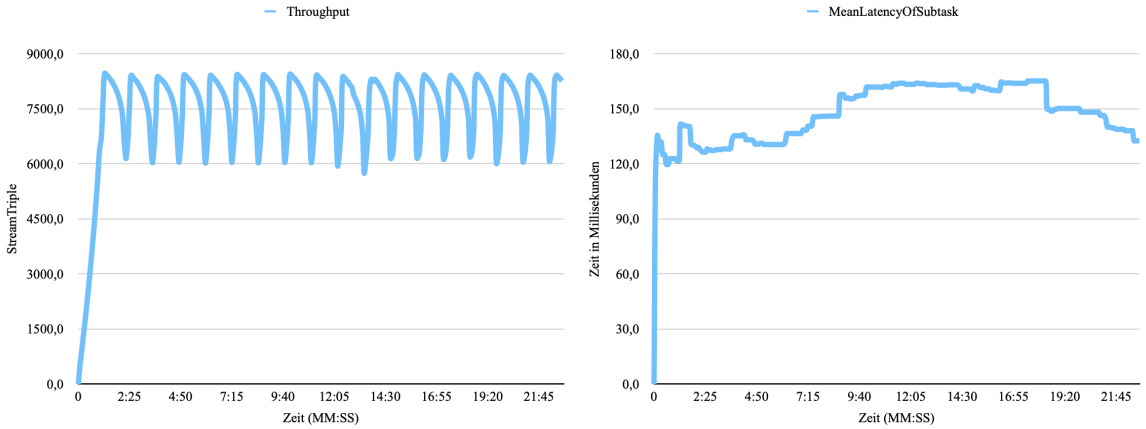

Abbildung 6.3.: Auswertung des dritten Tests

7. Zusammenfassung und Ausblick

Zu Beginn der Arbeit wurde die Motivation des Thema erörtert, die Zielsetzung definiert und ein Überblick zu verwandten Arbeiten gegeben. Nachdem in Kapitel 2 die wichtigsten Begriffe definiert und in Kapitel 3 das Apache Flink Framework vorgestellt wurde, sind in 4 die Konzeption und in 5 die Implementierung des Prototypen dieser Arbeit erläutert worden. Zum Abschluss sind drei Tests am Graph Stream Analyse Operator *Graph Stream Zoomer* durchgeführt und ausgewertet worden.

Mit dieser Arbeit wurde ein erster Grundstein zur Evaluation von Graph Stream Analyse Operatoren gelegt, die auf Apache Flink aufbauen. Darüber hinaus ist das Thema komplex und besitzt noch viel Potential für Erweiterungen. Beispielsweise erstellt Apache Flink bisher einen Metrik Reporter für jeden Task Manager, weshalb ein Report pro physischer Maschine und nicht fürs gesamte Cluster erstellt wird, was die Auswertung der Tests erschwert.

Darüber hinaus wurde noch kein Weg gefunden, die *End-To-End* Latency zu messen, ohne dabei Zugriff auf den Operator zu haben. Zukünftig könnte versucht werden, eine eigene Art Latency Marker zu implementieren, denen es möglich ist, diese zu messen. Alternativ könnte auch ein Benchmark entwickelt werden, welches auf *HTTP Requests*, statt auf dem Metrik Reporter Interface von Apache Flink fußt.

Literatur

[1] Braznev Sarkar. „Big Streaming Graph Analysis“. In: *Proceedings of the ACM India Joint International Conference on Data Science and Management of Data*. CoDS-COMAD '19. Kolkata, India: Association for Computing Machinery, 2019, S. 285–288. ISBN: 9781450362078. DOI: 10.1145/3297001.3297042. URL: https://doi.org/10.1145/3297001.3297042 (besucht am 09. 01. 2023).

[2] Mia Liang und Stephan Augsten. *Über den Sinn und Nutzen der Datenstrom-Verarbeitung*. 4. März 2021. URL: https://www.dev-insider.de/wo-spielen-streaming-datenbanken-ihre-staerken-aus-a-1002981/ (besucht am 05. 01. 2023).

[3] Toyotaro Suzumura Miyuru Dayarathna. *Benchmarking Graph Data Management and Processing Systems: A Survey*. 2020. DOI: 10.48550/ARXIV.2005.12873. URL: https://arxiv.org/abs/2005.12873 (besucht am 09. 01. 2023).

[4] Choi J u. a. „Dynamic graph convolutional networks with attention mechanism for rumor detection on social media“. In: *PLOS ONE 16(8)* (2021). URL: https://doi.org/10.1371/journal.pone.0256039 (besucht am 18. 12. 2022).

[5] Christopher Rost. *Graph Stream Zoomer: A window-based graph stream grouping system based on Apache Flink*. 5. Feb. 2023. URL: https://fosdem.org/2023/schedule/event/graph_grouping_zoomer/ (besucht am 14. 02. 2023).

[6] Chengcheng Yu u. a. „A parallel data generator for efficiently generating realistic social streams“. In: *Frontiers of Computer Science* (17. Juni 2019). URL: https://link.springer.com/article/10.1007/s11704-018-8022-z (besucht am 09. 01. 2023).

[7] Benjamin Erb u. a. „GraphTides: A Framework for Evaluating Stream-based Graph Processing Platforms“. In: *Proceedings of the 1st ACM SIGMOD Joint International Workshop on Graph Data Management Experiences, Systems (GRADES) and Network Data Analytics (NDA)*. Association for Computing Machinery, 2018, S. 1–10. ISBN: 9781450356954. DOI: 10.1145/3210259.3210262.

[8] Jeyhun Karimov u. a. *Benchmarking Distributed Stream Data Processing Systems*. 25. Okt. 2018. URL: https://hpi.de/fileadmin/user_upload/fachgebiete/rabl/publications/2018/Stream_BenchmarksICDE2018.pdf (besucht am 26. 02. 2023).

[9] Hanspeter Bieri und Felix Grimm. *Datenstrukturen in APL2*. 7. Jan. 2023. URL: https://link.springer.com/chapter/10.1007/978-3-642-77680-9_10 (besucht am 07. 01. 2023).

[10] Renzo Angles. *The Property Graph Database Model*. 7. Jan. 2023. URL: https://ceur-ws.org/Vol-2100/paper26.pdf (besucht am 07. 01. 2023).

[11] Apache Software Foundation. *Learn Flink: Hands-On Training*. 25. Okt. 2021. URL: https://ci.apache.org/projects/flink/flink-docs-release-1.13/docs/learn-flink/overview/ (besucht am 26. 02. 2023).

[12] Splunk. *Was ist Stream-Processing?* 12. Jan. 2020. URL: https://www.splunk.com/de_de/data-insider/what-is-stream-processing.html (besucht am 09. 01. 2023).

[13] Maciej Besta u. a. *Practice of Streaming Processing of Dynamic Graphs: Concepts, Models, and Systems*. 2019. DOI: 10.48550/ARXIV.1912.12740. URL: https://arxiv.org/abs/1912.12740 (besucht am 09. 01. 2023).

[14] awsthinkbox. *Advanced Features: Replication and Sharding*. 26. Mai 2017. URL: https://www.awsthinkbox.com/blog/replication-and-sharding (besucht am 10. 01. 2023).

[15] Nico Litzel Stefan Luber. *Was ist Apache Flink?* 28. März 2019. URL: https://www.bigdata-insider.de/was-ist-apache-flink-a-812389/ (besucht am 21. 02. 2023).

[16] Jan Oelschlegel. *Implementierung und Evaluierung einer Verarbeitung von Datenströmen im Big Data Umfeld am Beispiel von Apache Flink*. 17. Mai 2021. URL: https://nbn-resolving.org/urn:nbn:de:bsz:l189-qucosa2-748926 (besucht am 21. 05. 2022).

[17] Apache Software Foundation. *Use Cases*. 3. Okt. 2021. URL: https://flink.apache.org/usecases.html (besucht am 21. 02. 2023).

[18] Apache Software Foundation. *Data Pipeline*. URL: https://flink.apache.org/img/usecases-datapipelines.png (besucht am 21. 02. 2023).

[19] Apache Software Foundation. *Flink DataStream API Programming Guide*. URL: https://nightlies.apache.org/flink/flink-docs-master/docs/dev/datastream/overview/ (besucht am 21. 02. 2023).

[20] Apache Software Foundation. *Configuration | Basic Setup*. URL: https://nightlies.apache.org/flink/flink-docs-master/docs/deployment/config/ (besucht am 21. 02. 2023).

[21] Apache Software Foundation. *Data Sources*. URL: https://nightlies.apache.org/flink/flink-docs-master/docs/dev/datastream/sources/ (besucht am 21. 02. 2023).

[22] Apache Software Foundation. *Data Pipeline*. URL: https://nightlies.apache.org/flink/flink-docs-master/fig/source_components.svg (besucht am 21. 02. 2023).

[23] Apache Software Foundation. *Metrics*. 25. Okt. 2021. URL: https://nightlies.apache.org/flink/flink-docs-master/docs/ops/metrics/ (besucht am 22. 02. 2023).

[24] Liu Biao. *MetricGroup Hierarchie*. 16. Sep. 2020. URL: https://www.alibabacloud.com/blog/metrics-principles-and-practices-flink-advanced-tutorials_596634 (besucht am 21. 02. 2023).

[25] Apache Software Foundation. *REST API*. URL: https://nightlies.apache.org/flink/flink-docs-master/docs/ops/rest_api/ (besucht am 21. 02. 2023).

[26] Apache Software Foundation. *MessageHeaders Interface*. 21. Feb. 2023. URL: https://nightlies.apache.org/flink/flink-docs-release-1.15/api/java/org/apache/flink/runtime/rest/messages/MessageHeaders.html (besucht am 21. 02. 2023).

[27] Liu Biao. *Apache Flinks Metriksystem*. 16. Sep. 2020. URL: https://yqintl.alicdn.com/5b39904f365fbc997ccc992a3f856ea7b466cecf.png (besucht am 21. 02. 2023).

[28] Apache Software Foundation. *Plugins*. URL: https://nightlies.apache.org/flink/flink-docs-master/docs/deployment/filesystems/plugins/ (besucht am 21. 02. 2023).

[29] Apache Software Foundation. *Metric Reporters*. URL: https://nightlies.apache.org/flink/flink-docs-master/docs/deployment/metric_reporters/ (besucht am 21. 02. 2023).

[30] Apache Software Foundation. *Configuration / Metriken*. URL: https://nightlies.apache.org/flink/flink-docs-master/docs/deployment/config/#metrics-latency-interval (besucht am 21. 02. 2023).

[31] Rana Noureddin. *Distributed Grouping of Property Graph Streams*. 11. Mai 2020. URL: https://dbs.uni-leipzig.de/file/Master-Thesis-Rana-Noureddin.pdf (besucht am 22. 02. 2023).

[32] Jun Qin und Nico Kruber. *Getting into Low-Latency Gears with Apache Flink*. 18. Mai 2022. URL: https://flink.apache.org/2022/05/18/latency-part1.html (besucht am 22. 02. 2023).

[33] Jiri Sedlacek und Tomas Hurka. *Visual VM : All-in-One Java Troubleshooting Tool*. 11. Apr. 2022. URL: https://visualvm.github.io/index.html (besucht am 26. 02. 2023).

[34] Apache Software Foundation. *MetricReporter Interface*. 12. Apr. 2022. URL: https://github.com/apache/flink/blob/master/flink-metrics/flink-metrics-core/src/main/java/org/apache/flink/metrics/reporter/MetricReporter.java (besucht am 18. 02. 2023).

[35] Apache Software Foundation. *Scheduled Interface*. 12. Jan. 2022. URL: https://github.com/apache/flink/blob/master/flink-metrics/flink-metrics-core/src/main/java/org/apache/flink/metrics/reporter/Scheduled.java (besucht am 18. 02. 2023).

[36] Apache Software Foundation. *AbstractReporter Klasse*. 23. März 2022. URL: https://github.com/apache/flink/blob/master/flink-metrics/flink-metrics-core/src/main/java/org/apache/flink/metrics/reporter/AbstractReporter.java (besucht am 18. 02. 2023).

[37] New York Citi Bike. *New York Citi Bike Datensatz*. 30. Apr. 2018. URL: https://s3.amazonaws.com/tripdata/201306-citibike-tripdata.zip (besucht am 27. 02. 2023).

Erklärung

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

„Evaluation von Graph Stream Analyse Pipelines mit Apache Flink“

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

Leipzig, den 28.02.2023

Maximilian Martin

A. Anhang