UNIVERSITÄT
LEIPZIG

Institut für Informatik
Fakultät für Mathematik und Informatik
Abteilung Datenbanken

Flexible Graphstream-Analyse mittels Apache Flinks DataStream-API

Masterarbeit

vorgelegt von:
Simon Bordewisch

Matrikelnummer:
3684379

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

Zusammenfassung

In dieser Masterarbeit wird ein Modell für Graphstreams mit Eigenschaften für die Graphelemente und Graphzugehörigkeit entwickelt, welches eine Near-Realtime-Verarbeitung von solchen Graphstreams ermöglichen soll. Darauf aufbauend wurde GRAFS entwickelt, ein Prototyp für ein Graphstream-Analyse-Framework basierend auf Apache Flinks DataStream-API. GRAFS nutzt als Kantenstream-Modell Triplets, welche eine Kante und die zugehörigen Quell- und Zielknoten, sowie deren Eigenschaften speichern. Es wurden entsprechende Schnittstellen entworfen, die eine leichte Erweiterung des Systems durch Graph-Operatoren erlauben. Darauf aufbauend wurden die Graphstream-Operatoren Grouping, Pattern Matching, Subgraph, Transformation und Reduce mit Hilfe der in der DataStream-API befindlichen Methoden implementiert. Beim Grouping und Pattern Matching wurden Flinks Window-Funktionalitäten genutzt, um Elemente in einem Window zu sammeln und darauf die Graphalgorithmen auszuführen. Es wurde gezeigt, dass die reine Graphstream-Analyse-Verarbeitung mit Graphelement-Eigenschaften möglich ist und sich für Near-Realtime-Analysen eignet.

Simon Bordewisch
3684379

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1. Einleitung

Mit der in den Jahren immer schneller wachsenden Menge von gesammelten Daten ist es nötig geworden, Methoden zu entwickeln, die parallele Verarbeitung von Daten erlauben. Gleichzeitig steigt das Bedürfnis nach *Fast Data*, in welchem die generierten Daten in Echzeit oder Nahezu-Echtzeit (engl. *Near-Realtime*) analysiert werden [1]. Ein populärer Ansatz dafür ist die Datenstreamverarbeitung. Bei der Datenstreamverarbeitung werden die zu analysierenden Daten in sogenannte Ströme aufgeteilt, die dann elementweise in das Analysesystem eingegeben werden. Im Gegensatz zu herkömmlichen Daten kann dieser Strom kontinuierlich sein, d. h. er kann unendlich viele Daten liefern. Ziel ist es große Mengen von Daten mit möglichst geringer Latenz zu verarbeiten, ohne dass alle Daten zwischengespeichert werden müssen. Beispiele für Datenströme sind der Aktienhandel oder Sensordaten von IoT-Geräten. Apache Flink ist eines der Datenanalyse-Frameworks, welches mit der `DataStream`-API eine Plattform für Stream-Analyse bietet, da dort Transformationen definiert sind, welche auf einen Datenstrom angewendet werden können.

Gleichzeitig ist Graph-Analyse ist ein wichtiger Teil der Forschung, da viele der in Forschung und Wirtschaft untersuchten Daten als Graphen dargestellt werden können, um Zusammenhänge besser sichtbar zu machen. Auch an der Universität Leipzig in der Datenbank-Abteilung wird die verteilte Analyse von großen Graphdatenmengen erforscht. Hieraus ist bereits das erweiterte Graphmodell EPGM, sowie der Forschungsprototyp *Gradoop* [2] entstanden, welcher auf herkömmlichen Daten arbeitet.

Als Kombination von Datenstreaming und Graphen sind die sogenannten Graphstreams enstanden. Die Forschungsarbeit über die Analyse von Graphstreams hat sich in den letzten Jahren vestärkt. Beispiele für Graphstreams sind unter anderem Konversationen von Twitter, in denen die Nachricht eine Verbindung zwischen den Nutzern anzeigt, oder Daten von Vehikel-Miet-Plattformen wie Citi Bike, in der die Knoten die Stationen darstellen und die Fahrten zwischen den Stationen als Kanten interpretiert werden können. Genau wie herkömmliche Datenstreams können Graphstreams kontinuierlich sein.
Bisherige Graph-Streaming-Verarbeitungssysteme arbeiten auf den strukturellen Daten der Graphen oder nutzen ein Zwischenspeichermodell und arbeiten anschließend mit herkömmlichen Graph-Analyse-Techniken auf dem zwischengespeicherten Graphen. Dementsprechend gibt es bisher wenig Forschung im Bereich der reinen Graphstream-Analyse, im Speziellen auf sogenannten Property-Graphen, die weitere Informationen für die Elemente des Graphen bereitstellen können. Diese ist jedoch notwendig, um eine Near-Realtime-Analyse zu gewährleisten.

Das Ziel der Arbeit ist eine Implementierung eines End-zu-End Graphstream-Analyse-Frameworks mit Apache Flink. Das System soll in Java geschrieben werden. Für die Umsetzung ist es nötig ein Stream-Datenflussmodell für Property Graphen zu konzipieren und dieses auf Basis der `DataStream`-API umzusetzen, Referenzimplementierungen für gängige Graph-Operatoren zu entwerfen und eine Evaluation durchzuführen, um die Skalierbarkeit des Systems zu überprüfen. Um weitere Arbeiten möglichst zu erleichtern, soll sich dabei soweit wie möglich an der API von *Gradoop* orientiert werden, welches auf Flinks `DataSet`-API aufgebaut ist.

Die Arbeit ist wie folgt strukturiert. Zunächst werden die für die Arbeit nötigen Grundlagen erklärt. Hierfür werden Graphen und Streaming formal definiert und erklärt, sowie die genutzten Systeme vorgestellt. Anschließend werden verwandte Arbeiten im Bereich der Graph- und Stream-Analyse vorgestellt. Danach werden im Entwurf das Property Graph Streaming Modell sowie die Operatoren formal definiert. Kapitel 5 stellt die Implementierung des Modells sowie der Operatoren vor. Diese werden anschließend mit den im Streaming gängigen Metriken evaluiert. Schließlich werden die Ergebnisse der Arbeit beurteilt und ein Ausblick auf mögliche Optimierungen und Forschung in dieser Thematik gestellt.

2. Grundlagen

Zunächst werden die theoretischen Grundlagen von Graphen und das *Property Graph Model* als erweiteres Graphmodel definiert. Danach wird auf Streaming und die Konzepte um Streaming eingegangen und die Kombination von Graphen und Streams vorgestellt. Anschließend wird das Framework *Apache Flink*, im Speziellen die *DataStream*-API und ihre Transformationen vorgestellt und erläutert. Schließlich wird kurz auf *Apache Kafka* eingegangen, welches als Stream-Quelle für die Evaluation dient.

2.1. Graphentheoretische Grundlagen

In der Graphtheorie ist ein Graph eine mathematische Struktur, die Objekte und ihre Beziehungen zueinander beschreibt. In dieser Masterabeit wird im Speziellen auf die *gerichteten Graphen* eingegangen.

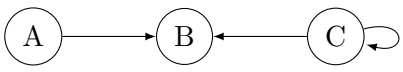
Definition 1. *Ein **gerichteter Graph** ist ein Paar $G = (V, E)$, mit*

- $V = \{v_1, v_2, \dots\}$ *ist eine Menge von Knoten* v_i *mit* $i \in \mathbb{N}$
- $E = \{e_1, e_2, \dots\}$ *ist eine Menge von Kanten* $e_j = (v_k, v_l)$ *mit* $v_k, v_l \in V$ *und* $j \in \mathbb{N}$

Um die Definition der einzelnen Tupelelemente von Graphen zu verkürzen, werden im Folgenden Tupelelemente aus einem Graphen G teilweise mit $\cdot(G)$ abgekürzt, zum Beispiel ist $V(G)$ die Knotenmenge von G.

In grafischen Darstellungen werden die Knoten häufig als Punkte oder Kreise mit Bezeichnungen, die den Graphen identifizieren, und die Kanten als Pfeile zwischen den jeweiligen Knoten dargestellt, die die Richtung der Beziehung anzeigen (vgl. Abbildung 2.1a). Die Definition lässt *Multigraphen* zu, d. h. zwei Knoten können durch mehrere Kanten verbunden sein, die dieselbe Richtung haben, solange die Identifikatoren der Kante unterschiedlich sind. Knoten und Kanten werden im Folgendem auch zusammenfassend *Graphelemente* genannt.
Eine wichtige Beziehung in der Graphentheorie ist der *Teilgraph* (engl. *Subgraph*, Beispiel siehe Abbildung 2.1b).

Definition 2. *Sei G ein Graph. Dann ist der Graph G' ein **Teilgraph** von G ($G' \sqsubseteq G$), wenn gilt $\forall v \in V(G') : v \in V(G)$ und $\forall e \in E(G') : e \in E(G)$.*



(a) Beispiel eines einfachen Graphen.

(b) Beispiel eines Teilgraphen des linken Graphen.

Abbildung 2.1.: Graphenbeispiel mit Teilgraph.

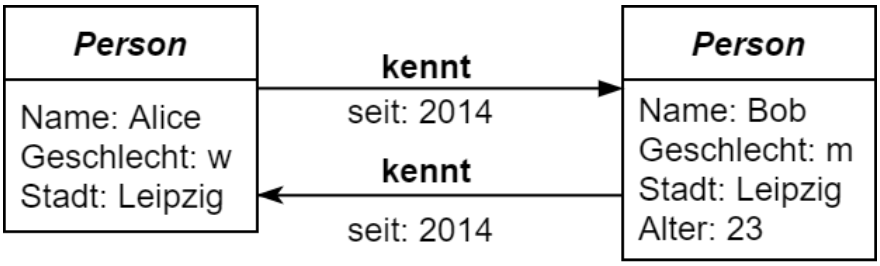


Abbildung 2.2.: Beispiel eines Property Graphen.[1]

Da das Graph-Modell lediglich simple Beziehungsdarstellungen erlaubt, wurde es zum sogenannten *Property Graph Model* (kurz *PGM*) erweitert [3, 4, 5]. Hierbei werden den Graphelementen Eigenschaften (engl. *Properties*) zugeordnet, die diese näher beschreiben.

Definition 3 (vgl. [3, 4, 5])**.** *Ein* ***Property Graph*** *ist definiert als* $PG = (V, E, T, \tau, K, A, \kappa)$ *mit*

- V *und* E *sind äquivalent zur Definition 1*
- T *ist eine Menge von Bezeichnern (engl.* ***Label****)*
- $\tau : (V \cup E) \to T$ *ist eine Funktion, die die Graphelemente auf ein Label abbildet*
- K *ist eine Menge von (Eigenschafts-)Schlüsseln (engl.* ***(Property) Key****) und* A *ist eine Menge von (Eigenschafts-)Werten (engl.* ***(Property) Value****)*
- $\kappa : (V \cup E) \times K \to A$ *ist eine Funktion, die einem Tupel von Graphelement und Key einen Wert zuordnet*

Abbildung 2.2 zeigt ein grafisches Beispiel für einen solchen Graphen. *Person* und *kennt* sind jeweils die Label, *Name*, *Geschlecht* und *Stadt* sind Property Keys für die Knoten. Dahinter stehen die Property Values. Das PGM ist schemalos, sodass die Knoten und Kanten nicht nur unterschiedliche Values, sondern auch unterschiedliche Keys haben können. Im Beispiel wurde dem Knoten mit dem Namen *Bob* zusätzlich der Property Key *Alter* mit dem dazugehörigem Value zugeordnet. Die Kanten besitzen hier jeweils den Key *seit* mit dem Value *2014*.

Das PGM ist ein häufig genutztes Modell und wird unter anderem in der Grapdatenbank Neo4j [6] verwendet. Eine Einschränkung des PGM ist, dass nur einzelne Graphen repräsentiert werden können. Demnach ist es nicht möglich mehrere Graphen (sogenannte *Graphmengen*, engl. *Graph Collections*) darzustellen.

An der Universität Leipzig wurde hierfür das obige Modell zum *Extended Property Graph Model* (kurz *EPGM*) erweitert [2].

Definition 4 (vgl. [2])**.** *Ein* ***Extended Property Graph*** *ist definiert als* $EPG = (V, E, L, T, \tau, K, A, \kappa)$ *mit*

- V, E, T, K, A *sind äquivalent zur Definition 3*
- $L = \{G_1, G_2, \dots\}$ *ist eine Menge von logischen Graphen* $G_m = (V_m, E_m)$*, wobei* $m \in \mathbb{N}$*. Für jeden logischen Graphen gilt:* $V_m \subseteq V, E_M \subseteq E$*, sowie* $\forall (v_i, v_j) \in E_m : v_i, v_j \in V_m$

[1] Basierend auf [2, Abb. 1]

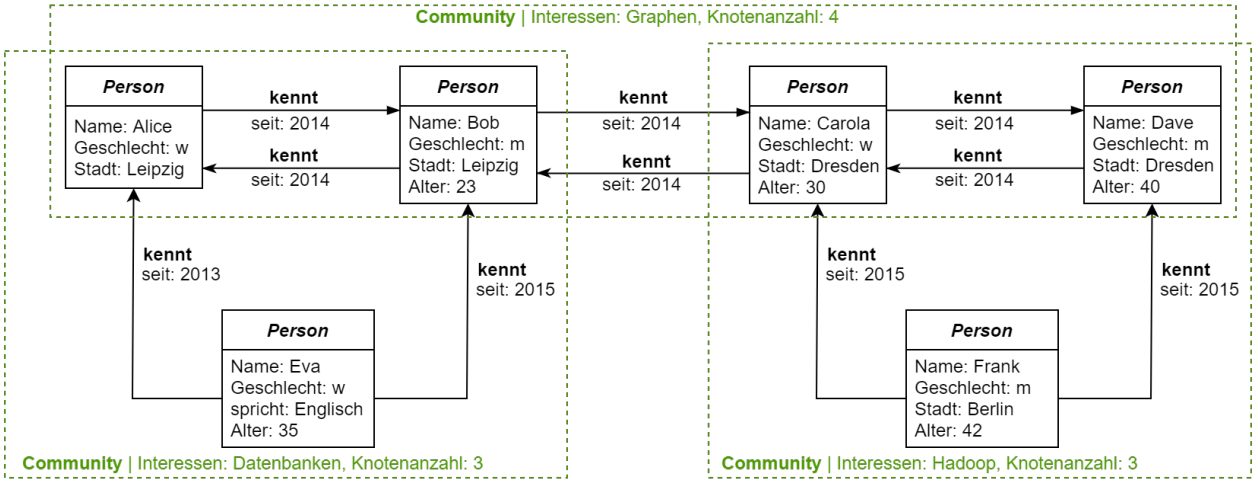


Abbildung 2.3.: Beispiel eines Extended Property Graph.[2]

- $\tau : (V \cup E \cup L) \to T$ *ist eine Funktion, die die Graphelemente oder einen Graphen auf einen Label abbildet*
- $\kappa : (V \cup E \cup L) \times K \to A$ *ist eine Funktion, die einem Tupel von Graphelement oder Graph und Property Key einen Property Value zuordnet*
- *Graph-Operatoren über dem geschlossenem Modell*

Abbildung 2.3 zeigt ein Beispiel eines solchen EPGs. Die Graph Collection besteht aus drei Graphen (grün markiert). Diese erhalten nun ebenfalls Label (hier: *Community*) und Properties, die jeweils die Interessen der Mitglieder und die Anzahl der in dem Graphen befindlichen Knoten darstellen.

2.2. Streams und Windows

In Abschnitt 2.1 wurden die Graph-Modelle vorgestellt. In welcher Form die Datenmodelle dabei zur Verarbeitung bereitgestellt werden, bleibt dabei zunächst offen.
Graphen werden meist vollständig als Stapeldatensatz bereitgestellt. Hier kann jederzeit auf jedes Element zugegriffen werden. Diese Form weist jedoch unter anderem das Problem auf, dass alle Daten jederzeit auch zugreifbar sein müssen. Dementsprechend müssen alle Daten gespeichert werden, was zu hoher Speicherauslastung führen kann. Zudem wird bei Stapeldaten von endlichen Datenquellen ausgegangen.

In diesem Abschnitt wird im Gegensatz dazu die Repräsentation (und Verarbeitungssicht) von Graphen als sogenannte *Datenströme* (engl. *(Data) Streams*) aufgezeigt. Datenströme sind zunächst Flüsse von Daten. Diese können endlich oder unendlich sein.

Definition 5 (vgl. [7])**.** *Ein* ***Datenstrom*** *ist ist eine Folge von Datenelementen* $D = (d_1, d_2, \dots)$*. Ein Datenstrom heißt* ***kontinuierlich****, wenn sein Ende zu Beginn der Verarbeitung nicht bekannt ist und er somit (theoretisch) unendlich ist. Ein Element aus dem Stream kann mit* $D[i] = d_i$ *adressiert werden. Es gilt:* $d \in D \Leftrightarrow: \exists i \in \mathbb{N} : d = D[i]$*.*

[2] Basierend auf [2, Abb. 1]

Diese Repräsentation zeigt Restriktionen für die Verarbeitung solcher Streams auf: Ein Streaming-System hat nur Zugriff auf ein Element des Streams und kann erst nach dessen Verarbeitung auf das nächste Element zugreifen. Um komplexere Analysen auf einem Datastream auszuführen, müssen zumeist jedoch Elemente zwischengespeichert werden. Eine Methode des Zwischenspeicherns ist das sogenannte *Windowing*, welches Teil des von Akidau u. a. vorgestellten *Dataflow Models* [8] ist. Beim *Windowing* wird der Stream in Teilfolgen unterteilt. In der Praxis sind diese häufig endlich und erlauben für die Verarbeitung eine Zwischenspeicherung der im Window befindlichen Elemente.

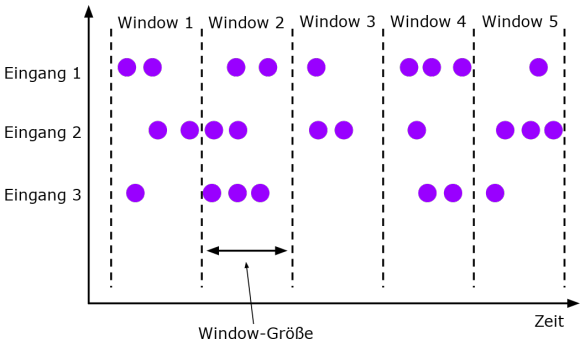
Definition 6 (vgl. [9, S. 277–279])**.** *Sei D ein Stream. Dann ist der Stream D' eine* ***Teilfolge*** *(oder Teilstream, engl.* ***Substream****) (bezeichnet mit $D' \sqsubseteq D$), wenn gilt $\forall k \in \mathbb{N} : D'[k] = D[n_k]$, wobei $n_1 < n_2 < \ldots$ eine aufsteigende Folge von Indizes ist.*

Definition 7 (vlg. [7, 10])**.** *Sei D ein Stream. Dann ist ein Fenster (engl.* ***Window****) W ein (endlicher) Substream von D. Die Zuordnung von Elementen aus dem Stream erfolgt durch einen* ***Window-Assigner*** *$w_D : \mathbb{N} \to P(\mathcal{W})$, wobei $\mathcal{W}$ die Menge aller Windows und $P(\mathcal{W})$ die dazugehörige Potenzmenge ist.*
Ein Window über alle Elemente des Streams $GW = D$ heißt ***Global Window****.*

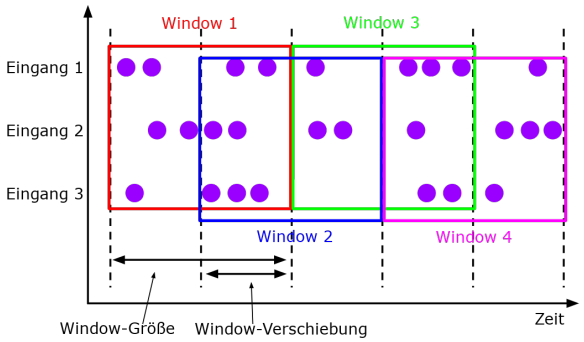
Durch die Definition des Window-Assigners kann auch die Größe des Windows (engl. *Window Size*) definiert werden In der Praxis werden verschiedene Arten von Window-Assigner genutzt, um die Elemente Windows zuzuordnen. Beispielhaft werden hier *Tumbling Windows* und *Sliding Windows* näher erläutert.
Bei Tumbling Windows (auch *Fixed Windows* genannt) wird eine statische Window Size definiert. Jedes Element wird genau einem Window zugeordnet. Dementsprechend überlappen sich die Elemente in verschiedenen Windows nicht (vgl. Abbildung 2.4a).
Im Gegensatz dazu wird beim *Sliding Window* nicht nur die Window Size definiert, sondern auch eine Verschiebung des Windows über den Stream. Dieser Parameter kontrolliert somit, wie oft ein neues Sliding Window gestartet wird. Wenn die Verschiebung kleiner ist als die Window Size, gibt es Überlappungen in den Windows und dementsprechend werden Elemente mehreren Windows zugeordnet (vgl. Abbildung 2.4b). Eine Verschiebung, die größer ist als die Window Size, ist in der Praxis eher unüblich.

Auf diese Windows können Funktionen angewendet werden, die die Elemente des Windows und damit auch den Stream verändern. Für diese Windows ist eine Zwischenspeicherung von Teilergebnisse erlaubt. Es können sogar alle Elemente im Window vor der Verarbeitung gesammelt werden ($\{d | \exists i \in \mathbb{N} : d = W[i]\}$. Anwendungsbeispiele für solche Windows sind unter anderem Aggregationen über die Elemente des Streams. Abbildung 2.4c zeigt ein solches Beispiel, in dem die im Window gesammelten Zahlenelemente summiert werden. Die Summen werden nach der Berechnung als Stream ausgegeben.

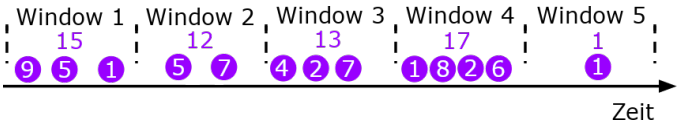
Graph-Streaming In der Literatur gibt es verschiedene Definitionen für *Graph-Streaming*, die die Repräsentation von Graphen als Datenstrom beeinflussen (Übersicht in [13]; vgl. auch Kapitel 3). Im Rahmen dieser Arbeit wurde der sogenannte *Kantenstream* (engl. *Edge Stream*, hier im Speziellen



(a) Beispiel eines Tumbling Windows (nach [11]). Elemente werden genau einem Window zugeordnet.



(b) Beispiel eines Sliding Windows (nach [12]). Elemente können hier mehreren Windows zugeordnet werden (hier biszu zwei), da die Window-Größe doppelt so groß ist wie die Verschiebung.



(c) Beispiel für Summierung über Tumbling Windows. Die Summe eines jeden Windows (in lila über den Zahlenkugeln) wird als Stream ausgegeben.

Abbildung 2.4.: Windows in Flink und ihrer Anwendung.

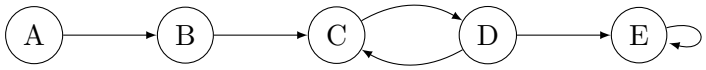


Abbildung 2.5.: Graph-Repräsentation des Streams $S = \langle (A, B), (B, C), (C, D), (C, E), (D, E) \rangle$.

Insert-Only Edge Streams, in denen nur Elemente eingefügt werden können) als Grundlage genutzt, der als Repräsentationsfundament für die Framework-Implementierung dient.

Definition 8 (nach [14]). *Sei $G = (V, E)$ ein Graph. Dann kann dieser Graph als **Edge Stream** ES repräsentiert werden mit $ES = \langle e_1, e_2, \dots \rangle$, wobei $\{e_1, e_2, \dots\} = E$.*

Die Knotenmenge ergibt sich aus den Kanten (da jede Kante aus zwei Knoten besteht). Dementsprechend beschränkt Definition 8 die Darstellung von Graphen auf diejenigen, in denen jeder Knoten mindestens eine Kante besitzt.

2.3. Apache Flink

Apache Flink[3] ist ein Framework für die verteilte Verarbeitung von sowohl Stapeldaten als auch kontinuierlichen Datenströmen [10]. In Flink wurde hierfür eine verteilte Stream-Engine entwickelt (Übersicht in Abbildung 2.6). Darauf aufbauend wurden die `DataSet`-API und die `DataStream`-API implementiert. Darüber hinaus bietet Flink domänenspezifische Bibliotheken und APIs, die auf den beiden vorherigen genannten aufbauen.

[3] `https://flink.apache.org`

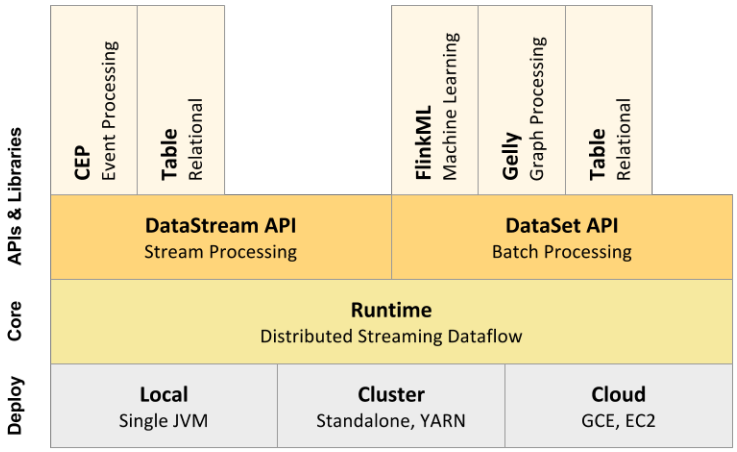


Abbildung 2.6.: Ebenen der Abstraktion von Flink [15].

Transformation	Beschreibung
Map	Nimmt ein Element und wendet auf dieses die gegebene UDF an. Das Ergebnis wird anschließend ausgegeben.
FlatMap	Identisch zu Map, nur das beliebig viele Elemente ausgegeben werden können.
Filter	Evaluiert für jedes Element eine benutzerdefinierte boolsche Funktion. Der daraus resultierende `DataStream` enthält genau die Elemente, für die die Funktion *wahr* ausgegeben hat.
KeyBy	Der Stream wird anhand des angegebenen Schlüsselfunktion in logische Partitionen unterteilt. Alle Elemente mit demselben Schlüssel werden zur selben Partition zugeteilt.
Window/WindowAll	Der Stream wird anhand des angegebenen Window-Assigners in Windows unterteilt. Alle Elemente desselben Windows werden vom selben Worker verarbeitet.
Reduce (für KeyBy)	*Rollender* Reduce. Kombiniert anhand der UDF das aktuelle Element mit dem letzten reduzierten Wert und gibt das Ergebnis aus.
Reduce (für Windows)	Reduziert alle Elemente anhand der UDF im Window zu einem Ergebnis und gibt dieses aus.

Tabelle 2.1.: Ausschnitt aus den Transformationen der Flink `DataStream`-API (aus [16]).

Im Rahmen dieser Arbeit wurde die `DataStream`-API als Grundlage für die Implementierung genutzt, da sie im Speziellen für die Verarbeitung von Datenströmen in Echtzeit gedacht ist.
Innerhalb der API sind Transformationen definiert. Eine Transformation ist eine Funktion, die auf die Elemente des Streams angewendet wird. Hierfür werden zumeist nutzerdefinierten Funktionen (engl. *User Defined Functions*, kurz *UDF*) genutzt. Tabelle 2.1 zeigt exemplarisch einige Transformationen.

Transformationen dienen dabei als Schnittstelle für die Anwendungslogik: Der Nutzer kann selbst eine auf die Transformationsschnittstelle passende UDF definieren, die auf den Stream angewendet wird. Da jede Transformation wieder einen Stream zurückgibt, können mehrere Transformationen hintereinander ausgeführt werden.

Mit Hilfe der `DataStream`-API können somit Datenanalyse-Workflows programmiert werden (vgl. Abbildung 2.7): Der Nutzer erstellt einen Datenfluss, in dem definiert wird, woher die Daten kom-

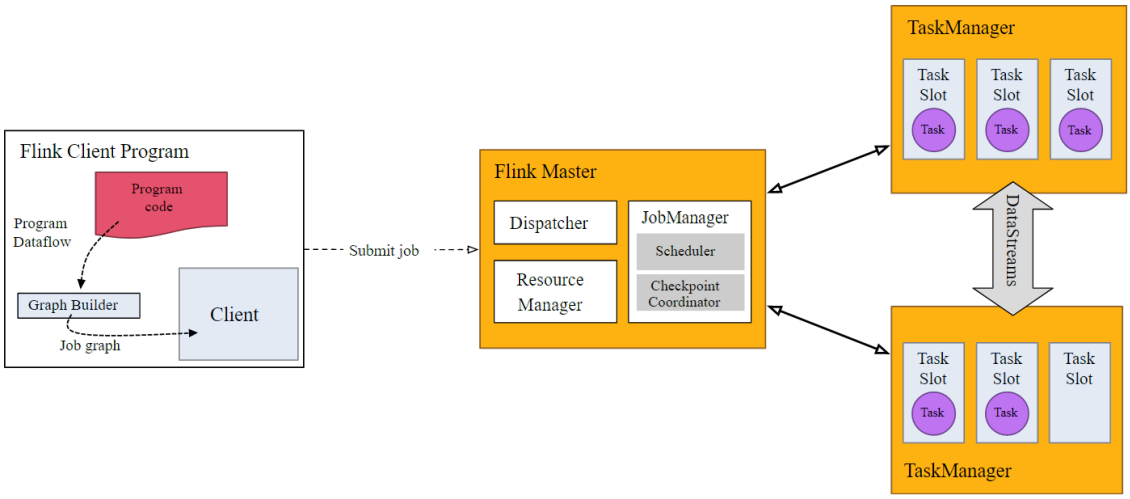


Abbildung 2.7.: Prozess des Job-Generation und Taskzuweisung in Flink.

men (*Datenquellen*; z. B. aus einer Datei oder von einem Websocket), eine beliebige Anzahl auf den Stream anzuwendende Transformationen, sowie wohin die Daten fließen (*Datensenken*). Dieser Datenfluss wird anschließend von Flink optimiert und in ein Runtime-Programm umgewandelt, welches einem gerichteten azyklischem Graphen von Knoten entspricht, deren Kanten den Datenfluss aufzeigen. Die Knoten stellen dabei Subtasks dar, die durch die angewendeten Transformationen definiert werden.

Abbildung 2.8 zeigt ein Beispiel eines solchen Datenflusses. Im obigen Teil des Bildes sieht man dem vom User definierten Transformationsfluss. Der untere Teil zeigt einen von Flink definierten Datenfluss-Graph mitsamt Parallelität. Die Daten können dabei eins zu eins von einem Subtask zum nächsten fließen (z. B. von den *Sources* zu den *map*-Transformationen), um möglichen Overhead durch Neuverteilung der Daten zu vermeiden. In Flink wird hierfür ein Shared-Nothing-Ökosystem genutzt, beidem der Datenautausch über das Netzwerk stattfindet. Der Grad der Parallelität kann von Flink festgelegt oder vom Nutzer bestimmt werden.

Neuverteilung der Daten ist bei bestimmten Transformationen notwendig (z. B. bei `KeyBy` oder `window`) oder kann z. B. durch die Nutzung der *Rebalancing*-Transformation forciert werden, um eine bessere Last-Balancierung zu gewährleisten, welche zu einer effizienteren Verarbeitung führen kann. Sollen die Daten nicht neu verteilt werden, so können zudem weitere Optimierungen stattfinden, bspw. können ein oder mehrere Subtasts zu einem Task zusammengefasst werden, die auf einem *Task-Manager* innerhalb eines Threads ausgeführt werden. In Abbildung 2.8 wäre es z. B. denkbar, dass die Subtasks *Source* und *map* zu einem Task zusammengefasst werden. Flink versucht dabei die Subtasks sinnvoll zu bündeln, um den Kommunikationsaufwand durch den Datenfluss gering zu halten.

Das Runtime-Programm kann anschließend an den Flink-Master weitergegeben werden, der für das Planen der Flink-Jobs zuständig ist. Dabei kommuniziert dieser mit den Task-Managern des Shared-Nothing-Clusters und weist die generierten Tasks zu. Jeder Task-Manager weist ein oder mehrere *Task Slots* auf, die bei der Clustergenerierung konfiguriert werden. Die Task-Manager führen anschließend die für den Slot zugeteilten Tasks auf dem Stream aus und senden diesen, wenn nötig, an andere Task-Manager weiter. Dabei können diese abhängig von der Transformation

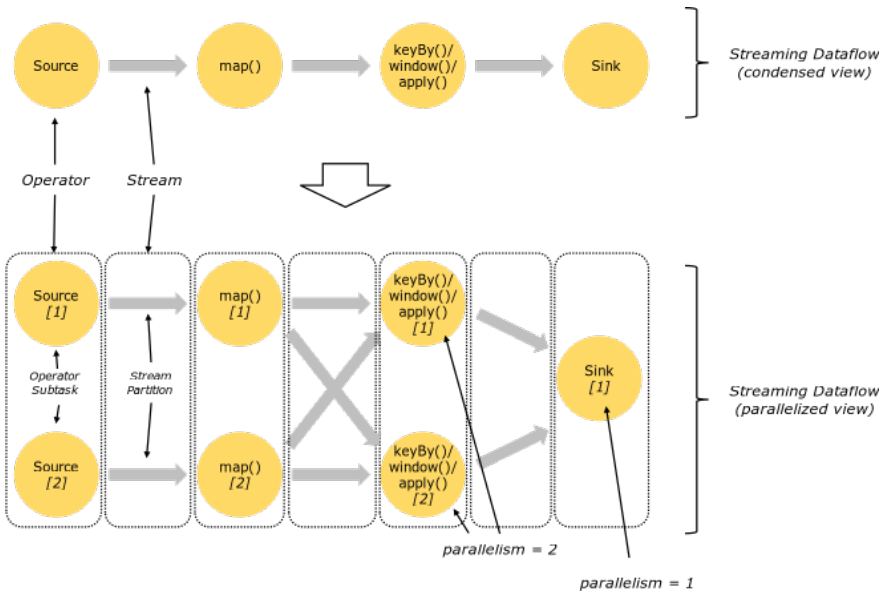


Abbildung 2.8.: Datenfluss von Flink [17].

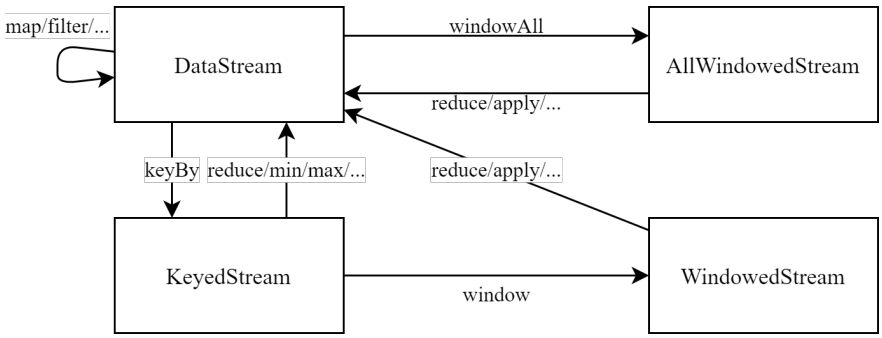


Abbildung 2.9.: Ausschnitt der Stream-Klassen und Übergänge in Flink.

in verschiedenen Threads und möglicherweise auf verschiedenen Maschinen parallel auf dem Stream ausgeführt werden, um eine möglichst hohe parallele Verarbeitung der Daten zu gewährleisten.

Abhängig davon ob und wie der Stream durch die Transformationen partitioniert wird, lässt er sich in eine von mehreren Stream-Klassen einteilen. Abbildung 2.9 zeigt einen Ausschnitt der Klassen und die jeweiligen Methoden, die für die Überführung eines Streams von der einen in die andere Klasse verantwortlich sind.

Ausfallsicherheit Flink nutzt sogenanntes *Checkpointing*, um zu gewährleisten, dass nach einem Ausfall eines Cluster-Mitglieds die Streamverarbeitung fortgesetzt werden kann. Hier werden die Zustände der Task-Manager (*Snapshot*) periodisch gespeichert. Fällt ein Task-Managers aus, kann durch die Wiederherstellung eines vorher gespeicherten Zustandes (bspw. auf einem anderen Task-Manager) die Verarbeitung fortgesetzt werden. Je nach Datenquelle und -senke ergeben sich zudem verschiedene Maximal-Garantien für die Verarbeitung der Elemente. Ein Socket-Stream bietet bespielsweise nur eine *At-Most-Once*-Garantie, in der nicht Garantiert ist, dass alle Elemente verarbeitet werden können. Grund dafür ist, dass die Daten, die während der Wiederherstellungsphase (in der bspw. ein neuer Task-Manger zum Lesen des Socket-Streams beauftragt wird) vom Socket-Stream gesendet werden, vom Cluster nicht aufgenommen werden können.

Bei der Nutzung von Kafka als Quelle kann eine *Exactly-Once*-Verarbeitung ermöglicht werden, sodass jedes Element im Stream genau einmal verarbeitet und ausgegeben wird.

Zeit und Windowing in Flink Um einer Transformation mehrere Elemente des Datenstroms gleichzeitig als Eingabe zur Verfügung zu stellen, müssen die Daten zunächst in Windows zusammengefasst werden. Windows sind, wie bereits erwähnt, eine Möglichkeit um mehr als ein Element auf einmal betrachten zu können. Die Daten werden in Flink innerhalb der Windows anhand der Zeit partitioniert. Im Dataflow Model [8] unterscheidet man zwischen zwei unterschiedlichen Zeitbegriffen: *Processing Time* und *Event Time*.

Definition 9 (aus [8])**.** ***Ereigniszeit*** *(engl.* ***Event Time****) ist die Zeit, zu welchem das Element (als Event) selbst aufgetreten ist, bzw. das Element erstellt wurde.*

Definition 10 (aus [8])**.** ***Verarbeitungszeit*** *(engl.* ***Processing Time****) ist die Zeit, zu welcher das Element von dem Verarbeitungsmaschine aufgenommen wurde, also die derzeitige Zeit der Systemuhr der Verarbeitungsmaschine. Dabei werden keine Annahmen über die Synchronisation der Uhren innerhalb eines verteilten Systems gemacht.*

Definition 11. *Ein Window, welches von einem auf Event Time oder Processing Time basierenden Window-Assigner generiert wurde, heißt* ***Time Window****.*

Den Definitionen zufolge bleibt die Event Time konstant und die Processing Time verändert sich ständig, während das Element durch die Verarbeitungspipeline fließt. Die Nutzung der Processing Time funktioniert automatisch, hat aber den Nachteil, dass die Reihenfolge der Elemente im Stream ggf. nicht nachvollzogen werden kann, da sich während der Verarbeitung und Verteilung der Elemente die Reihenfolge ändern kann, was aufgrund der stetigen Neugenerierung der Processing Time, dazu führt, dass die originale Reihenfolge nicht mehr abgeleitet werden kann.

Windowing in Flink kann mit beiden Zeitdefinitionen verwendet werden. Während die Processing Time vom System selbst generiert wird, muss für die Erkennung der Event Time eine sogenannte *Watermark-Strategy* erstellt werden, also eine Funktion, die jedem Element im Stream die Event Time zuweist. Dies geschieht bspw. durch das Auslesen eines Feldes innerhalb des Datenelements.

Flink unterstützt die bereits im vorherigen Kapitel besprochenen Arten von Window-Assigner. Intern nutzt Flink dafür *Trigger* und *Evictor*, um zu entscheiden, wann ein Window geschlossen wird und wann Elemente aus einem Window entfernt werden (vgl. [18]).
Trigger können beispielweise auslösen, wenn ein bestimmtes Element im Window angekommen oder eine bestimmte Zeit erreicht ist. Beim Auslösen des Triggers wird entweder die Verarbeitung gestartet (`FIRE`), alle Elemente aus dem Window entfernt (`PURGE`) oder beides (`FIRE_AND_PURGE`). Somit können verschiedene Arten von Trigger erstellt werden. Bereits in Flink enthalten sind bspw. der `EventTimeTrigger`, dessen Auslösen von der Eventzeit abhängt, oder der `CountTrigger`, der auslöst, sobald eine bestimmte Anzahl von Elementen im Window gesammelt wurde.
Ein Evictor kann Elemente nach dem Auslösen eines Trigger aus dem Window entfernen, bevor die Verarbeitung des Windows mit einer Window-Funktion beginnt.

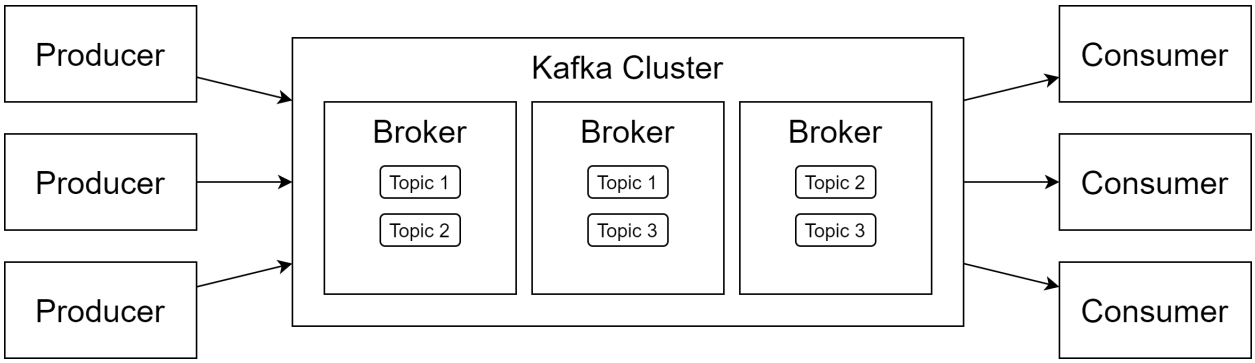


Abbildung 2.10.: Grundlegender Datenfluss mit Kafka

Iterative Streams In Flink gibt es neben den oben genannten Arten von Streams auch den `IterativeStream`, der es zulässt, dass auf Elemente des Streams Transformationen kontinuierlich angewendet werden, solange diese eine definierte Bedingung erfüllen.

Execution Mode Flinks `DataStream`-API unterstützt seit Version `1.12` die Möglichkeit, die Art des Streams auszuwählen, sodass Flink eventuelle Optimierungen vornehmen kann. `STREAMING` ist der *klassische* Modus, welcher hauptsächlich für kontinuierliche Streams vorgesehen ist. Daneben gibt es noch den `BATCH`-Modus, welcher von einem endlichem Stream ausgeht. Beispielsweise werden dort die Taskjobs in Verarbeitungsphasen unterteilt und gewartet bis die erste Verarbeitungsphase für alle Daten abgeschlossen ist, bevor die nächste begonnen wird. Da in dieser Masterarbeit der Fokus auf kontinuierlichen Streams liegt, wird der (standardmäßig eingestellte) `STREAMING`-Modus genutzt.

2.4. Apache Kafka

Apache Kafka [19] ist eine verteilte Event-Streaming-Plattform. Mit ihr können Daten (*Events*) parallel an ein Cluster gesendet, von diesem gespeichert und von Clients (sogenannte *Consumer*) gelesen werden.

Abbildung 2.10 zeigt den grundlegenden Datenfluss innerhalb eines Kafka-Clusters. In Kafka werden die Daten in *Topics* eingeteilt. Die Regulation des Datenflusses geschieht mittels eines Push-Pull-Systems: Datenerzeuger (genannt *Producer*) können Daten für ein Topic an Kafka senden und die *Consumer* können vom Kafka-Cluster Daten für ein Topic erbitten. Dabei kann jedes Topic in mehrere Partitionen aufgeteilt werden. Ein Partition liegt dabei immer auf mindestens einem Worker (sogenannte *Broker*). Jeder Broker kann mehrere Partitionen haben. Eine höhere Anzahl von Replikationen der Partitionen erhöht die Ausfallsicherheit. Die Koordination unter den Brokern findet über *Apache ZooKeeper* statt (nicht im Bild enthalten). Für jede Partition gibt es einen Partition-Leader, der die Lese- und Schreibanfragen für die Partition verarbeitet. Sollte eine Partition mehr als einen Broker haben, so werden die Broker, die nicht der Partition-Leader sind, als Follower bezeichnet. Sollte der Leader ausfallen, wird vom ZooKeeper ein neuer Leader unter den Followern der Partition ausgewählt.

Im Rahmen dieser Masterarbeit wird Kafka als Datenquelle genutzt, um über die Erweiterung des Kafka-Connectors eine Geschwindigkeitsregulierung der Datenquelle zu gewährleisten.

3. Verwandte Arbeiten

Nach bestem Wissen gibt es bisher keinen Ansatz, der ein End-zu-End Streaming-Analyse-Framework für Property Graphen bietet. Dementsprechend werden hier Arbeiten vorgestellt, die sich mit Graphverarbeitung beschäftigten, sowie Arbeiten zu Datenstrom-Analyse und Graph-Streaming im Speziellen.

3.1. Grundlage für ein Graphstream-Framework

Neben dem bereits vorgestellten Apache Flink gibt es weitere Systeme, die Streamverarbeitung erlauben und dementsprechend als Grundlage für diese Masterarbeit dienen könnten. Das bereits erwähnte Apache Kafka bietet bspw. mit der `Streams`-API ebenso die Möglichkeit eingehende Streams zu verarbeiten [20]. Dabei unterstützt es wie Flink verschiedene Arten von Zeitinformationen und erlaubt Windowing auf dem Datensatz. Darüber hinaus ist eine Exactly-Once Verarbeitung möglich, sodass jedes Element im Stream genau einmal verarbeitet und ausgegeben wird. Die `Streams`-API sieht dabei einen Stream als Tabelle an (jedes Element im Stream ist ein Tabelleneintrag). Dementsprechend werden übliche Tabellen-Operationen wie Joins und Gruppierungen (innerhalb eines Windows) ermöglicht. Confluent, das Unternehmen, welches die Entwicklung von Kafka betreut, sieht Kafka mehr als einfache Echtzeit-Streamverarbeitung und Vorbereitung für Analyse-Pipelines [21]. Auch andere Arbeiten, die Streamverarbeitungsbenchmarks durchführen, nutzen Kafka zumeist nur als Datenquelle (bspw. [22, 23]) für andere Frameworks. Apache Kafka könnte in Zukunft eine Alternative als Grundlage für ein Graphstream-Analysesystem sein, die Betrachtung von Apache Flink wurde in dieser Arbeit aufgrund des bekannten Umfelds priorisiert.

Andere von der Apache Foundation betreute Streamverarbeitungsframeworks sind Apache Spark [24] und Apache Storm [25]. Beide bauen, wie Flink, aus dem definierten Datenfluss einen DAG. Apache Spark ist primär für die Verarbeitung von Stapeldaten gedacht. Dabei baut es auf partionierten Datensätzen, sogennanten *Resilient Distributed Datasets* (kurz *RDD*), auf. Auf den RDDs können Transformationen durchgeführt werden, die ebenso Exactly-Once garantieren. Sparks `DStream`-API und *Structured Streaming* bauen beide auf bereits vorhandenen Grundlagen in Spark auf. DStream basiert dabei auf den RDDs, Structured Streaming baut wiederum auf DStream und Spark SQL auf. Der Stream wird dabei in sogenannte *Micro-Batches* aufgeteilt. Diese sind kleinere Stapeldaten-Mengen und können dementsprechend wie ein RDD verarbeitet werden. Dies hat zur Folge, dass bei der Verarbeitung eine höhere Latenz zu erwarten ist als bei Flink [26, 23].
In Apache Storm besteht der DAG aus Knoten von *Spouts* und *Bolts*. Spouts sind äquivalent zu den Datenquellen in Flink, sodass sie Daten annehmen und an diese an die Bolts weitersenden, in denen die Verarbeitung stattfindet. Die Bolts sind dabei ein Interface, welches von dem Nutzer implementiert werden muss, um die gewünschte Transformation zu erreichen. Storm ermöglicht, wie Flink, Windowing auf den Daten. Außerdem bietet es zunächst nur eine *At-Least-Once*-Garantie. Exactly-Once kann aber für die meisten Verarbeitungen mit der Higher-Level API *Trident* ermöglicht werden, wobei die Elemente wie bei Spark in kleinere *Batches* gesammelt werden.

Eine genauere Übersicht über die Charakteristiken von Spark und Storm im Vergleich zu Flink kann in [27] gefunden werden, ein Benchmarking auf diesen in [23].

3.2. Graph-Analyse-Systeme auf Stapeldaten

Es gibt viele Systeme, die sich mit verteilter Analyse von Graphen in Form der Stapelverarbeitung oder in iterativen Prozessen beschäftigen. Im folgenden werden einige bekannte Systeme aufgelistet. Die meisten davon bauen auf MapReduce auf und sind dementsprechend dem in dieser Arbeit zu entwerfendem Prototypen nah verwandt.

Graphbasierte Analyse ist eine der Forschungsrichtungen an der Universität Leipzig in der Abteilung für Datenbanken. In [28] und [2] stellen Junghanns u. a. nicht nur das bereits erwähnte EPGM vor, sondern auch den Forschungsprototypen *Gradoop*, welcher einen End-zu-End Ansatz für Verwaltung und Analyse von Graphdaten bereitstellt. Gradoop ist in Java geschrieben und baut dabei auf Apache Flinks *DataSet*-API auf. Gradoop wurde so konzipiert, dass eine Erweiterbarkeit leicht gegeben ist. Soll ein neuer Operator implementiert werden, erbt er von einem der gegebenen Interfaces, bspw. von *UnaryGraphToGraphOperator*, wenn der Operator einen Graph als Eingabe annimmt und einen Graph ausgibt. Gradoop wurde über die Zeit erweitert, bspw. um den Operator *Pattern Matching* [29] und um die Verarbeitung von Graphen, die sich über die Zeit verändern (sogenannte *temporal graphs*) [30].
Die Verwendung zu vereinfachen, wurde für Gradoop die sogenannte *Graph Analytical Language* (*GrALa*) definiert. Die Syntax von GrALa ermöglicht die flexible Anwendung von Operatoren unter Nutzung von UDFs. Da bei vielen Operatoren als Ergebnis der verarbeitete Graph zurückgegeben wird, ist Operator Chaining möglich. Ein so vom Nutzer definierter Workflow wird von Gradoop anschließend mithilfe der *DataSet*-API in eine Flink-Runtime übersetzt und ermöglicht somit die verteilte Analyse der Graphdaten. Flinks DataSet-API bietet nur die Möglichkeit zur Stapelverarbeitung, sodass weder die Verarbeitung von kontinuierliche Streams noch eine Near-Realtime-Verarbeitung möglich ist.
Zusätzlich wurden an der Universität Repräsentationsalternativen des EPGMs in Masterarbeiten erforscht. Die Umsetzungen basieren dabei auf *Apache Spark SQL* [31] und Apache Flinks *Table-API* [32]. Die Table-API baut auf Flinks DataStream- und DataSet-API auf. Jedoch können nicht alle in der Table-API implementierten Transformationen auf Datenstreams angewendet werden. Die in der Masterarbeit implementierten Graphoperatoren nutzen dabei solche Transformationen, wie z. B. *Join* und *Intersect*. Somit ist diese Implementierung nicht für Graphstreams geeignet.

Eines der bekanntesten Systeme für eine Graphverarbeitung ist Pregel [33], welches von Google entwickelt wurde. Das Modell wurde im Jahr 2010 vorgestellt und ist durch das *Bulk Synchronous Parallel*-Modell inspiriert. In Pregel sind die Knoten konzeptuell auf verschiedenen Maschinen verteilt. Außerdem hält jeder Knoten seine ausgehenden Kanten. Anstatt das die Knoten und Kanten durch das System fließen und an jeder Station Verarbeitungsschritte durchgeführt werden, bleiben die Knoten auf der ihr zugeteilten Maschine. Stattdessen werden (abhängig von den Knoten generierte) Nachrichten durch das Netzwerk geschickt, die die anzuwendenden Funktion verändern. Genauer gesagt wird innerhalb eines Verarbeitungszyklus (*Superstep*) eine nutzerdefinierte Funktion auf einem Knoten ausgeführt, die das Verhalten des Knotens für den Superstep definiert. Ein

Knoten kann dabei Nachrichten aus dem vorherigen Superstep lesen, Nachrichten an andere Knoten für den nächsten Superschritt senden und seinen Zustand und den seiner Kanten verändern. Dieser Ansatz nennt sich *Knotenzentraler Ansatz* (engl. *vertex-centric*, »Think like a vertex«[33]). Eine Open Source-Umsetzung von Pregel findet sich in *Apache Giraph* [34], welches auf Apache Hadoop aufgebaut wurde. Die Graphelemente in Giraph halten (neben ihrer ID) einen Wert. Das Modell unterstützt neben einfachen Transformationen auch Algorithmen wie PageRank [35], das Finden von Zusammenhangskomponenten oder kürzesten Pfaden (vgl. [36, S. 129ff.]). Während in solchen Systemen zwar Daten (i. e. Nachrichten) gestreamt werden, sind sie jedoch nicht für Graphstreams anwendbar.

In der Apache-Software-Familie gibt es zudem weitere Ansätze für ein Graph-Analyse-System. Mit *Gelly* [37] stellt bspw. Apache Flink eine auf der DataSet-API basierenden Graph-API zu Verfügung. Genau wie bei Giraph hält jedes Graphelement in Gelly eine ID und einen Wert.
GraphX, welches eine Komponente von Apache Spark ist, ist eine weitere Apache-Implementation. GraphX baut dabei auf den in Spark verwendeten RDDs auf. Im Gegensatz zu Gelly arbeitet GraphX mit Property Graphen und bietet somit eine native Implementierung von Graph-Operatoren mit Properties, wie beispielsweise Transformationen (über die *mapVertices*- und *mapEdges*-Methoden). Beide Erweiterungen (Gelly und GraphX) bieten ebenfalls herkömmliche Operatoren an, wie Subgraph oder Transformation und Algorithmen wie z. B. PageRank (vgl. [38, 39]).

3.3. Graph-Streaming-Systeme

Graph Streaming deckt ein weites Gebiet ab, in dem Graphen in Form eines Streams verarbeitet werden. In [13] werden die gängigen Praktiken in Graph Streaming Frameworks vorgestellt und analysiert. Die Erkenntisse werden im Folgenden zunächst zusammengefasst.

Übersicht Neben dem bereits definierten Kantenstream findet auch der sogenannte *Knotenstream* Erwähnung (bspw. in [40]). In diesem besteht jedes Element des Streams aus einem Knoten sowie dessen Adjazenzliste. Der Kantenstream ist jedoch die am häufigsten genutzte Repräsentationsform [13]. Eine Abwandlung des bereits vorgestellten *Insert-Only* Kantenstream ist die Möglichkeit, dem Kantenstream die Information mitzugeben, ob die jeweilige Kante zum Graphen hinzugefügt (*Insert*) oder gelöscht (*Delete*) wird oder ob sie eine bereits bestehende Kante ersetzt (*Update*).

Die meisten Frameworks bieten laut der Analyse nur simple Datenmodelle, mit einem Typ oder einer Eigenschaft pro Graphelement und zumeist einem Zeitstempel, der anzeigt, wann die Kante eingefügt wurde [13].
Wie bereits diskutiert, ist das Zwischenspeichern für komplexere Analysen auf dem Stream häufig notwendig. Neben dem bereits vorgestellten Windowing wird von vielen Graphstream-Systemen das sogenannte *Snapshot Model* genutzt, welches alle eingehenden Kanteninformationen bis zu einem Zeitpunkt enthält, d. h. ein Snapshot S zu einem Zeitpunkt i ist eine Menge von Kanten $S = \{e | t_D(e) < i\}$. Dieses Konzept entspricht jedoch nicht dem vorgesehenem Anwendungsfall und die Snapshots können, je nach Datenquelle, sehr groß werden.

In [41] wird ein weiteres Zwischenspeichermodell vorgestellt: Das *Edge Decay Model* gibt neueren Kanten mehr Relevanz (und älteren weniger), da neuere Verbindungen häufig wichtiger sind, aber alte Verbindungen nicht verfallen (wie es in dem Windowing Modell der Fall ist).
Neben den in [13] genannten Modellen wird in der Literatur für Graphstreams auch das *Semi*-Streaming-Modell genannt (z. B. [14, 42, 7]). Hier darf bei der Verarbeitung eine Speicherkomplexität von $\mathcal{O}(n\mathsf{poly}(\log n))$ erreicht werden, wobei n die Anzahl der Knoten ist. Die Idee dahinter ist, dass die Anzahl der Kanten eines Graphen häufig deutlich größer ist als die Anzahl der Knoten. Graph-Algorithmen, die dieses Modell nutzen, benötigen meist ein oder mehrere Durchläufen des Streams, bevor ein Ergebnis feststeht.

In [13] wird ebenfalls aufgezeigt, dass bei dem Programmiermodell die meisten Frameworks auf existierende Modelle für statische Graphverarbeitung setzen oder diese erweitern. Die dabei genutzten Paradigmen sind unter anderem der in Pregel vorgestellte knotenbasierte Ansatz.

RDF-Stream-Verarbeitung Auch im Bereich der *Resource Description Framework*-Graphen (kurz *RDF-Graphen*) gibt es seit einiger Zeit Bemühungen mit einer Streaming-Variante der Graphen umgehen zu können. Ein RDF-Graph ist eine spezielle Graphform, in dem die Knoten URIs oder Werte (bspw. Strings oder Zahlen) und Kanten URIs repräsentieren (vgl. [43]). Ziel ist es eine wohldefinierte Semantik für das Web zu finden, sodass dieses maschinenlesbar wird. In RDF können Aussagen über Zusammenhänge zwischen Objekten mit Triplets ausgedrückt werden. Dabei besteht ein Triplet aus einem *Subjekt*, einem *Prädikat* und einem *Objekt* (in dieser Reihenfolge). Subjekt und Prädikat sind jeweils URIs das Objekt kann eine URI oder ein Literal sein. Eine Aussage kann wie folgt aussehen:

<http://dbpedia.org/resource/Barack_Obama> <http://dbpedia.org/property/spouse>
<http://dbpedia.org/resource/Michelle_Obama>.

In dem Beispiel wird ausgedrückt, dass Barack Obamas Ehefrau Michelle Obama ist. Barack und Michelle Obama repräsentieren hierbei die Knoten (Barack Obama ist das Subjekt, Michelle Obama ist das Objekt) und *Spouse* die Kante (und Prädikat) zwischen den beiden Knoten. Ein RDF-Stream wird zumeist als Stream solcher Tripel sowie einer Zeitrepräsentation (bspw. Event Time) formuliert (vgl. [44, 45]). Arbeiten in diesem Bereich beschäftigen sich zudem mit der Modellierung der Abfragesprache über einen Graphsstream (z. B. [44, 46, 47]). Abfrageergebnisse werden hier gestreamt, d. h. sie werden zur Verfügung gestellt, sobald ein neues Ergebnis gefunden wurde.

Graph-Streaming auf Apache Flink Eine weitere Masterabeit der Universität Leipzig beschäftigt sich mit der verteilten Verarbeitung des *Pattern Matching*-Operators auf Graphstreams [48]. Beim Pattern Matching werden innerhalb eines Graphen zu einem gegebenen Mustergraphen passende Teilgraphen gesucht und ausgegeben (mehr dazu siehe Unterabschnitt 4.3.2). Alkamel nutzt dabei Flinks DataStream-API und setzt Property Graphen als Kantenstream um. Hierfür nutzt er ein *StreamObject*, welches eine Kante und die zugehörigen Knoten kapselt. Die Knoten und Kanten halten dabei ihr jeweils zugehöriges Label und die Properties für das jeweilige Element, sowie eine ID. Die Implementierung des Operators erfolgt über Windows: Der Operator findet innerhalb des Windows den auf den Abfragegraphen passenden Teilgraphen. Dieser Teilgraph wird anschließend als Ganzes zurückgegeben, d. h. aus einem Kantenstream wird ein Stream von Teilgraph-Objekten.

In der Arbeit wird eine Dual Simulation-Implementation des Pattern Matching genutzt, da dieses im Gegensatz zu isomorphischem Matching eine geringe Komplexität aufweist. Die Evaluation ergab, dass die durchschnittliche Verarbeitungszeit pro Element von der Anzahl der im Window befindlichen Elementen abhängt und bei zu großer Window Size das System überlastet ist und es zu einem Rückstau von Elementen (sogenannte *Backpressure*) kommt.

Schließlich sei die Masterarbeit von Bali [49] und die daraus folgende Implementierung *Gelly-Streaming* [50] erwähnt, welche ein Graphstream-Analyse-Framework auf Basis von Flink vorstellt. Die Implementierung beruht dabei, genau wie diese Arbeit, auf Flinks `DataStream`-API. Ziel war es eine API zu entwickeln, welche für Graphstreams geeignet ist und der API von Gelly ähnelt. Die Implementierung in [49] beruht dabei auf einem Kantenstream eines einfachen Graphen. Bali nutzte das Graphdatenmodell von Gelly, sodass eine Kante des Streams aus Keys (die Knoten-Keys) und einem Wert besteht. Der Typ der Keys kann dabei beliebig sein, genau wie der Typ des Wertes. Die Knoten selbst sind in diesem Modell lediglich als IDs referenziert. Dies unterscheidet Balis Arbeit von dem geplanten Datenmodell in dieser Masterarbeit, da hier der Property Graph als Datenmodell-Grundlage dienen soll.
Dementsprechend richten sich auch die Operatoren in [49] an dieses einfache Datenmodell. Ein *Subgraph*-Operator (für Definition siehe Unterunterabschnitt 4.3.1.2), hier als *Filter* bezeichnet, kann über die Keys der Knoten und über den Kantenwert ausgeführt werden. *Knoteninduzierte Subgraphs* können über Eigenschaften der Knoten zwar durchaus abgebildet werden, in dem man die Knoten-Properties im Kanten-Wert speichert, dies ist aber zumindest nicht nativ unterstützt, im Gegensatz zu dem für diese Arbeit geplanten Modell. Darüber hinaus wurden einfache Transformationen auf den Kanten, verschiedene Arten von Aggregationen und verschiedene Single-Pass-Graph-Algorithmen eingebaut, bspw. für eine Approximation der Anzahl der Dreiecke im Graphstream und ein sich kontinuierlich verbessernder[4] Stream, der die Anzahl der Knoten oder Kanten des Streams mitteilt. Ähnlich wie in dieser Arbeit vorgesehen, wurde für die Aggregationen die Windowing-Funktionalität von Flink genutzt.
Insgesamt wurde auf eine Trennung von Daten und Funktionen verzichtet und der Fokus auf die Implementierung verschiedener Graph-Algorithmen gelegt, sodass sich zwar verschiedene UDFs für die Operatoren nutzen lassen, sich die Operator-Implementierung aber teilweise in den Daten-Klassen befindet. Diese Trennung ist ein zentraler Bestandteil dieser Masterarbeit, sodass sich das Modell mit anderen Operatoren erweitern lässt, ohne das Datenmodell vorher erweitert werden muss.

[4] Zitat: »[...] continuously improving [...]«[49, S. 30]

4. Entwurf

In diesem Abschnitt werden zunächst die Anforderungen an das zu entwickelnde Datenmodell und Streamingsystem vorgestellt. Anschließend wird das Streaming-Modell für Property Graphen definiert. Schließlich werden die in der Arbeit umgesetzten Operatoren definiert und an Beispielen erläutert.

4.1. Anforderungen

Im Rahmen dieser Masterarbeit soll zunächst ein Graph-Stream-Modell entwickelt werden, welches sowohl einzelne Graphen als auch Graph Collections repräsentieren kann. Außerdem soll es, ähnlich wie das PGM, Informationshaltung für die Graphelemente unterstützen können. Dabei soll bei der Entwicklung des Systems darauf verzichtet werden den Stream durch zusätzliche Datenquellen anzureichern (z. B. indem alle Label und Properties in einer Datenbank gespeichert werden und diese von den Operatoren für die aktuell zu verarbeitende Kante abgefragt werden können). Das Modell muss zudem die Anwendung von Operatoren ermöglichen (bspw. die Veränderung von Informationen an den Graphelementen), wobei auch die Verkettung mehrerer Operatoren möglich sein soll.

Anschließend soll ein Prototyp entworfen und implementiert werden, welcher für Analysen auf dem entwickelten Modell geeignet ist. Hierfür sollen exemplarisch die Graph-Operatoren *Subgraph*, *Transformation*, *Grouping* und *Pattern Matching* implementiert werden. Bei dem Entwurf ist darauf zu achten, dass das System auf Apache Flinks DataStream-API aufbauen soll und Operatoren gegebenfalls auf das Stream-Modell angepasst werden müssen.
Der entwickelte Prototyp soll anschließend hinsichtlich Performance und Skalierbarkeit gemessen und evaluiert werden. Hierfür sind vorher Benchmark-Konzepte zu entwerfen. Als Evaluierungsdatensatz dient dabei ein Datensatz von *Citi Bike* [51].

4.2. Graphstream-Modell

Um ein System auf einem Property-Graphen zu implementieren, der als Kantenstream repräsentiert wird, muss zunächst ein mathematisches Modell eingeführt werden, welches beide Konzepte vereint.

Definition 12. *Ein* $PGS = (ES, L, T, \tau, K, A, \kappa)$ *ist ein* ***Property-Graphstream (PGS)*** *mit*

- *ES ist ein Edge Stream nach Definition 8*
- T, τ, K, A *und* κ *sind wie beim PGM definiert (Definition 3)*
- *L ist wie beim Extended Property Graph Model definiert (Definition 4)*

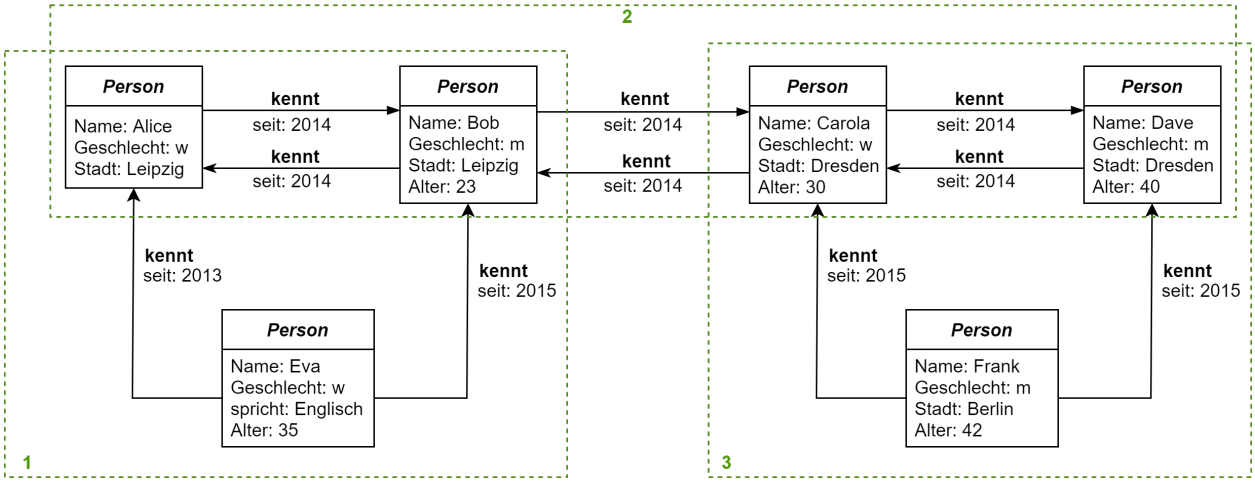


Abbildung 4.1.: Aus einem Property-Graphstream entstehender Graph.

Ein Property-Graphstream erlaubt die Darstellung von Property-Graphen als Kantenstream. Zusätzlich ist durch die Nutzung von L aus dem EPGM die Zuordnung von Graphelementen zu mehreren Graphen möglich. Dies ist notwendig, da der zu implementierende Operator *Pattern Matching* als Ergebnis mehrere Graphen zurückgeben kann und die Graphelemente innerhalb eines Streams sonst nicht unterschieden werden können. Abbildung 4.1 zeigt, wie ein aus einem PGS entstehender Graph aussieht. Der einzige Unterschied sind die fehlenden Properties und Label für die Graphen (stattdessen sind diese lediglich mit IDs gekennzeichnet).
Das Datenmodell der einzelnen Elemente im Stream ist Teil der Implementierung und wird demensprechend im nächsten Kapitel erörtert.

Windows auf Graphstreams Ähnlich zum Datenstrom wird bei einem Graphstream davon ausgegangen, dass bei der Anwendung eines Operators nur auf ein Element gleichzeitig zugegriffen werden kann. Wie beim Datenstrom können Windows erzeugt werden, auf denen dann Funktionen angewendet werden können. Ein Window kann dabei als Teilgraph gesehen werden.

Definition 13. *Sei S ein Stream für einen Graphen G. Sei $S' = \langle e_1, e_2, \dots \rangle$ ein Substream auf S. Dann kann daraus ein Graph $G' = (V', E')$ gebildet werden, wobei $\{e_1, e_2, \dots\} = E'$ und $(u, v) \in E \Leftrightarrow u, v \in V'$. G' ist Teilgraph von G. Ist S' ein Window heißt G' auch **Windowed Graph**.*

Die Definition lässt sich auch auf das PGSM erweitern (vgl. Unterunterabschnitt 4.3.1.2). Der entstehende Graph ist dann ein EPG (ohne Label und Properties für die logischen Graphen).

4.3. Operatoren

Genau wie das EPGM enthält das PGSM Operatoren für einzelne Graphstreams und Graph-Collection-Streams. Operatoren können einzelne Graphen oder Graph Collections zurückgeben. Tabelle 4.1 zeigt eine Übersicht der im PGSM definierten Operatoren.

Operator	Eingabe	Ausgabe
Subgraph	Graphstream	Graphstream
Transformation	Graphstream	Graphstream
Windowed Grouping	Graphstream	Graphstream
Windowed Pattern Matching	Graphstream	Graph-Collection-Stream
Reduce	Graph-Collection-Stream	Graphstream

Tabelle 4.1.: Übersicht über die PGSM-Operatoren.

Im Folgendem werden die Operatoren vorgestellt, die Teil der Framework-Implementierung sind. Die Implementierung der Operatoren wird in Abschnitt 5.2 erläutert. Zunächst werden die Graph-zu-Graph-Operatoren vorgestellt. Anschließend wird *Windowed Pattern Matching* vorgestellt, der ein Graph-zu-Graph-Collection-Operator ist. Schließlich wird der Hilfsoperator *Reduce* vorgestellt.

4.3.1. Graph-zu-Graph

Graph-zu-Graph-Operatoren nehmen einen Graphen als Eingabe und geben einen Ergebnis-Graphen zurück. Zu den hier implementierten Operatoren gehören *Windowed Grouping*, *Subgraph* und *Transformation*, die im Folgenden definiert werden.

4.3.1.1. Windowed Grouping

Der *Windowed Grouping-Operator* (im Folgendem kurz *Grouping-Operator*) ist eine Funktion, bei der die Knoten und Kanten des Streams aufgrund ihrer Eigenschaften zusammengefasst werden. Dadurch ist es möglich Informationen über die generelle Struktur des Graphen zu gewinnen und gleichzeitig die Komplexität zu reduzieren. Beim Windowed Grouping wird Windowing auf dem Stream angewendet und die Knoten und Kanten innerhalb des Windows aggregiert. Die folgenden Definitionen zu Grouping basieren auf den Definitionen in [52] und wurden an das PGSM angepasst.

Definition 14. *Sei ES ein Edge Stream und $WG = (V, E)$ ein Windowed Graph eines Windows auf ES. Dann bildet der* ***Grouping-Operator*** *einen zusammengefassten Window-Graphen $WG' = (V', E')$ aus WG, einer nichtleeren Menge von* ***Grouping Keys für Knoten*** *$K_\nu \subseteq K$ und einer Menge von* ***Grouping Keys für Kanten*** *$K_\epsilon \subseteq K$, sowie einer Menge von* ***Aggregatfunktionen*** *Λ_ν und Λ_ϵ. Dabei bilden V' eine Menge von Superknoten und E' eine Menge von Superkanten.*

Der resultierende Graph wird anschließend von dem Operator als Stream ausgegeben. Wie bereits erwähnt, findet das Grouping nur innerhalb eines Windows statt. Elemente außerhalb des Windows haben keine Auswirkungen auf den zusammengefassten Windowed Graphen.

Für den zusammengefassten Windowed Graphen werden die Knoten anhand eines oder mehrerer angegebener Property Keys gruppiert, sodass alle Knoten in derselben Gruppe dieselben Property Values für die angegeben Keys besitzen. Um die Gruppierung über die Label zu ermöglichen und die Defintionen zu vereinfachen, wird das PGM so erweitert, dass die Label der Graphelemente in

der Property Key-Menge sind ($T \subseteq A$) und diese durch einen speziellen Key t durch κ gewonnen werden können, d. h. $\forall g \in V \cup E : \kappa(g, t) = \tau(g)$.

Definition 15. *Sei G ein Property-Windowed Graph, G' ein zusammengefasster Property-Windowed Graph und $s_\nu : V(G) \to V(G')$ eine surjektive Funktion. Dann heißt v'_i* ***Superknoten*** *und $\forall v \in V(G)$ ist $s_\nu(v)$ der Superknoten von v. Knoten in $V(G)$ werden auf ihren Property Values gruppiert, sodass für eine nichtleere Menge von Grouping Keys für Knoten $K_\nu \subseteq K(G)$ gilt: $\forall u, v \in V(G) : s_\nu(u) = s_\nu(v) \Leftrightarrow \forall k \in K_\nu : \kappa(u, k) = \kappa(v, k)$.*

Die Gruppen werden demnach zu Superknoten zusammengefasst und speichern dabei die Eigenschaften, die die Gruppe repräsentieren, d. h. $\forall k \in K_\nu : \forall v \in V(G) : \kappa(s_\nu(v), k) = \kappa(v, k)$.

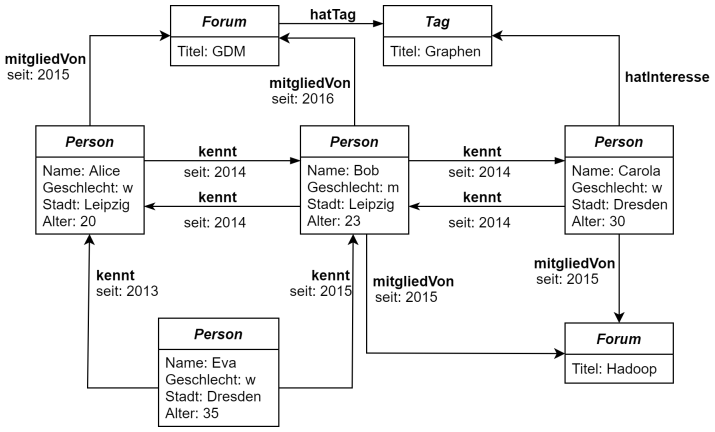
Anschließend werden die Kanten zu sog. *Superkanten* gruppiert. Hierbei ist nicht nur zu beachten, dass die Kanten nur dann gruppiert werden, wenn sie dieselben Property Values für die angegeben Key aufweisen und wenn sich jeweils die Startknoten der Kante in derselben Gruppe befinden und selbiges für die Ziel-Knoten gilt:

Definition 16. *Sei G ein Windowed Property-Graph, G' ein zusammengefasster Windowed Property-Graph und $s_\epsilon : E(G) \to E(G')$ eine surjektive Funktion. Dann heißt e'_i* ***Superkante*** *und $\forall e \in E(G)$ ist $s_\epsilon(e)$ der Superknoten von e. Superknoten sind dabei sowohl von den Superknoten als auch den Grouping Keys für Kanten $K_\epsilon \subseteq G$ abhängig: $\forall \langle u, v\rangle, \langle x, y\rangle \in E(G) : s_\epsilon(\langle u, v\rangle) = s_\epsilon(\langle x, y\rangle) \Leftrightarrow s_\nu(u) = s_\nu(x) \wedge s_\nu(v) = s_\nu(y) \wedge \forall k \in K_\epsilon : \kappa(\langle u, v\rangle, k) = \kappa(\langle x, y\rangle, k)$*

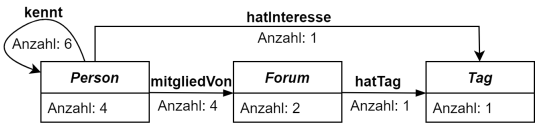
Analog zu den Superknoten speicheren Superkanten dabei die Eigenschaften, die die Gruppe repräsentieren, d. h. $\forall k \in K_\epsilon : \forall e \in E(G) : \kappa(s_\epsilon(e), k) = \kappa(e, k)$.
Die in den gruppierten Graphelementen befindlichen Property Values können mit sogenannten Aggregatfunktionen zu einem Wert zusammengefasst werden.

Definition 17. *Seien $\Lambda_\nu = \{\alpha_\nu : P(V(G)) \to A\}$ und $\Lambda_\epsilon = \{\alpha_\epsilon : P(E(G)) \to A\}$ Mengen von assoziativen und kommutativen Knoten- bzw. Kanten-Aggregatfunktionen. Die resultierenden Werte der Funktionen werden in einem Property gespeichert, dessen Key durch $f\alpha : \Lambda_\nu \cup \Lambda_\epsilon \to K(G)$ bestimmt wird, d. h. $\forall \alpha_\nu \in \Lambda_\nu : \forall v' \in V(G') : \kappa(v', f_\alpha(\alpha_\nu)) = \alpha_\nu(\{v \in V(G) | s_\nu(v) = v'\})$ und $\forall \alpha_\epsilon \in \Lambda_\epsilon : \forall e' \in E(G') : \kappa(e', f_\alpha(\alpha_\epsilon)) = \alpha_\epsilon(\{e \in E(G) | s_\epsilon(e) = e'\})$*

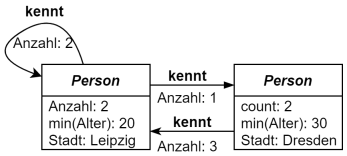
Typische Aggregatfunktionen sind zum Beispiel Summen-, Maximum- und die Durchschnittsbildung oder auch das Zählen der Graphelemente. Abbildung 4.2 zeigt einen Windowed Graphen und zwei aus der Anwendung des Grouping-Operators resultierende Graphen. Für 4.2b wurde sowohl für Knoten als auch Kanten nur nach den Labels gruppiert, d. h. $K_\nu = K_\epsilon = \{t\}$. Zusätzlich werden die Aggregatfunktionen $\alpha_{\nu_{count}} : V \mapsto |V|$ und $\alpha_{\epsilon_{count}} : E \mapsto |E|$ verwendet und die resultierenden Werte dem Key $f_\alpha(\alpha_{\nu_{count}}) = f_\alpha(\alpha_{\epsilon_{count}}) =$ »*Anzahl*« zugewiesen. Bei dem Graphen in 4.2c wurde für die Knoten auf dem Property *Stadt* gruppiert ($K_\nu = \{Stadt\}$). Die Kanten werden wieder auf dem Labels gruppiert. Da die *Forum*-Knoten und der *Tag*-Knoten das Property *Stadt* nicht enthalten, fallen diese weg. Übrig bleiben die auf den Städten gruppierten Personen



(a) Der Ausgangsgraph.



(b) Auf den Label zusammengefasster Graph.



(c) Auf dem Property Key *Stadt* zusammengefasster Graph.

Abbildung 4.2.: Beispiel für Grouping (Basierend auf [52, Abb. 1]).

und die Kanten, die mit Personen-Knoten verbunden sind. Zusätzlich zu der oben definierten Zähl-Aggregation $\alpha_{\nu_{count}}$ und $\alpha_{\epsilon_{count}}$ wird hier auf den Knoten mit der Funktion $\alpha_{\nu_{minAge}}(v) = i$ mit $i \in \mathbb{N} \wedge \exists v \in V : i = \kappa(v, Alter) \wedge \forall u \in V : i \leq \kappa(u, Alter)$ ein Minimum ermittelt und dem Key $f_\alpha(\alpha_{\nu_{minAge}}) =$ »min(Alter)« zugewiesen.

4.3.1.2. Subgraph

Der Subgraph-Operator erzeugt aus einem Eingabegraphen einen Teilgraphen anhand von Eigenschaften der Knoten und Kanten. Da der Subgraph-Operator auf PGS angewendet wird, muss die Definition von Teilgraphen entsprechend angepasst werden.

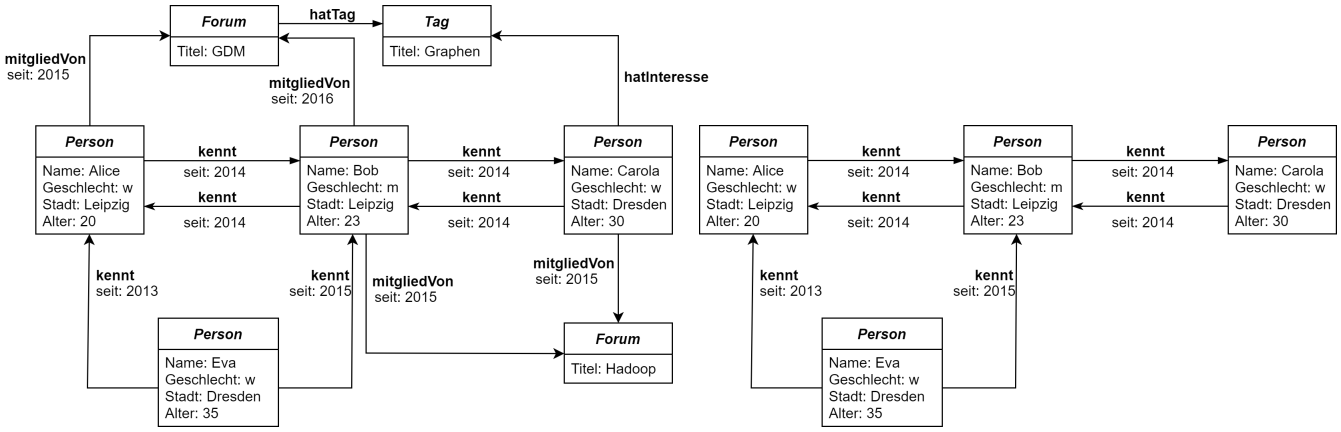
Definition 18. *Sei $PGS = (ES, L, T, \tau, K, A, \kappa)$ ein Property-Graphstream. Sei ES' Teilfolge von ES. Dann heißt $PGS' = (ES', L, T, \tau, K, A, \kappa)$ Teilgraph (oder Teilfolge/-stream) von PGS.*

Die Teilfolge eines Kantenstreams repräsentiert intuitiv einen Teilgraph des ursprünglichen Graphen, da lediglich Kanten des Ursprungsstreams *weggelassen* werden. Bei der Anwendung des Subgraph-Operators werden typischerweise *Prädikatsfunktionen* (auch *Filterprädikat* oder *Filterfunktion* genannt) spezifiziert, die Knoten und Kanten aus dem Eingabegraphen selektieren.

Definition 19. *Sei $PGS = (ES, L, T, \tau, K, A, \kappa)$ ein Property-Graphstream, PGS' ein Substream von PGS. Sei $\theta_\nu : V \to \{true, false\}$ eine **Prädikatsfunktion** auf Knoten, die auf dem Label und Eigenschaften der Graphelemente in PGS definiert sind. Dann erfüllt PGS' die Prädikatsfunktion, wenn die Knoten in ES die Funktion θ_ν erfüllen, d. h.*

- $\forall (u, v) \in ES$ *mit* $\theta_\nu(u) = true \wedge \theta_\nu(v) = true \implies (u, v) \in ES'$ *und*
- $\forall (u, v) \in ES' : (u, v) \in ES \wedge \theta_\nu(u) = true \wedge \theta_\nu(v) = true$

*Ein solcher Teilgraph heißt auch **knoteninduzierter Teilgraph**.*



(a) Ausgangsgraph G für den Subgraph-Operator.[5] (b) Teilgraph von G

Abbildung 4.3.: Beispiel für einen Teilgraph.

Definition 20. *Sei $PGS = (ES, L, T, \tau, K, A, \kappa)$ ein Property-Graphstream, PGS' ein Substream von PGS. Sei $\theta_\epsilon : E \to \{true, false\}$ eine **Prädikatsfunktion** auf Kanten. Dann erfüllt PGS' die Prädikatsfunktion, wenn die Kanten in ES die Funktion θ_ϵ erfüllen, d. h.*

- $\forall i \in \mathbb{N} : e = ES[i] : \theta_\epsilon(e) = true : \exists j \in \mathbb{N} : e = ES'[j]$ *und*
- $\forall i \in \mathbb{N} : e = ES'[i] : \exists j \in \mathbb{N} : e = ES[j] : \theta_\epsilon(e) = true$

*Ein solcher Teilgraph heißt auch **kanteninduzierter Teilgraph**.*

Auf einem Stream können auch beide Prädikatsfunktionen gleichzeitig angewendet werden. Damit ensteht eine Teilfolge, deren Graphelemente beide Prädikatsfunktionen erfüllen.

Definition 21. *Aus einem gegebenen Property-Graphstream PGS und der Prädikatsfunktion θ_ν und/oder der Prädikatsfunktion θ_ϵ bildet der **Subgraph-Operator** eine (knoten-/kanteninduzierten) Teilgraph PGS' von PGS, der die gegebenen Prädikatsfunkionen erfüllt.*

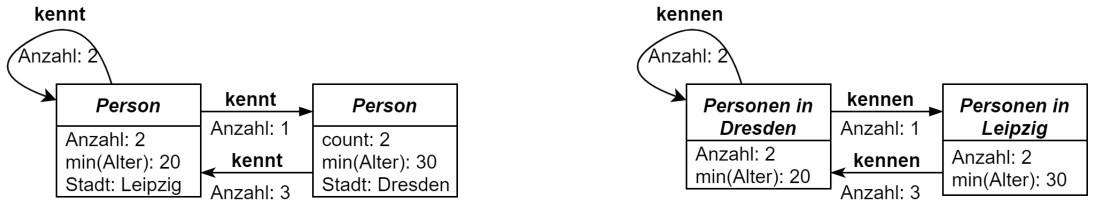
Abbildung 4.3 zeigt ein Beispiel der Anwendung eines solchen Subgraph-Operators. Zur Einfachheit wird statt einzelner Elemente eines Streams der Graph als Ganzes visualisiert. 4.3b ist ein knoteninduzierter Teilgraph, der nur Knoten enthält, die das Label *Person* enthalten ($\theta_\nu(v) = true \Leftrightarrow \tau(v) = Person$).

4.3.1.3. Transformation

Mit dem Transformationsoperator können die Properties und Label von Elementen im Graphstream verändert, hinzugefügt oder gelöscht werden, ohne dass dabei deren Identität oder die Struktur des Graphen selbst verändert wird (vgl. [53]).

Definition 22. *Sei $PGS = (ES, L, T, \tau, K, A, \kappa)$ ein Property-Graphstream. Sei $\mathcal{G}$ die Menge aller Property-Graphstreams. Dann ist $t : \mathcal{PGS} \to \mathcal{PGS}$ ein **Transformationsoperator**, wenn für*

[5] Basierend auf [52, Abb. 1a]



(a) Ausgangsgraph. (b) Transformierter Graph.

Abbildung 4.4.: Beispiel für eine Transformation.[6]

den resultierenden Stream $PGS' = t(PGS)$ mit $PGS = (ES', L', T', \tau', K', A', \kappa')$ gilt, dass die logischen Graphen gleich bleiben, d. h. $L' = L$ und die Graphelemente des Kantenstreams gleich bleiben, d. h. $\forall i \in \mathbb{N} : ES'[i] = ES[i]$.

Transformationen verändern dementsprechend nur die Properties und Label eines Graphen. Abbildung 4.4 zeigt ein Beispiel für die Anwendung einer Transformation. Wie zuvor wird zur Einfachheit der Graph als Ganzes visualisiert. Der Ausgangsgraph ist die Gruppierung aus Abbildung 4.2c. Der transformierte Graph verbessert die Lesbarkeit durch die Umbenennung der Label in den Graphelementen. Die (nun überflüssige) Eigenschaft *Stadt* wurde zudem entfernt.

4.3.2. Graph-zu-Graph-Collection

Graph-zu-Graph-Collection-Operatoren nehmen einen Graphstream als Eingabe und geben einen Graph-Collection-Stream aus. Im Folgendem wird der *Windowed Pattern Matching*-Operator (im Folgenden kurz *Pattern Matching-Operator*) vorgestellt.

Pattern Matching Der Pattern Matching-Operator findet alle Teilgraphen innerhalb eines Windows, die dieselben Strukturen wie der angegebene *Abfragegraph* haben. Die folgenden Definitionen basieren, soweit nicht anders angegeben, auf den Definitionen aus „Cypher-based graph pattern matching in Gradoop“ [29] und wurden auf das PGSM angepasst.

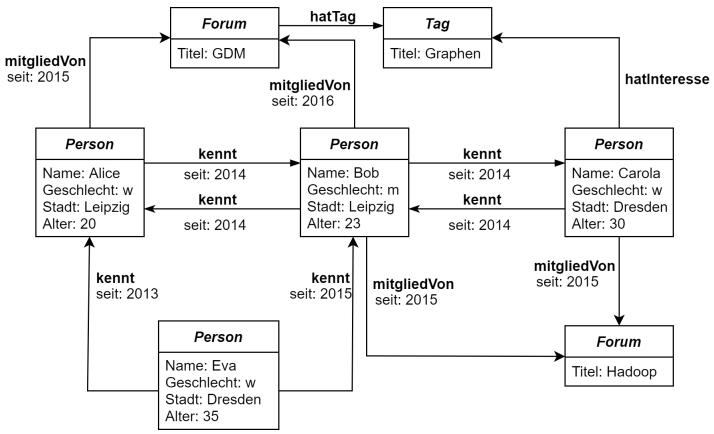
Definition 23. *Sei ES ein Edge Stream und WG ein Windowed (Property-)Graph G eines Windows auf ES. Ein Tupel $Q = (V_q, E_q, \theta_\nu, \theta_\epsilon)$ ist ein **Abfragegraph** mit Abfrageknoten V_q, Abfragekanten E_q. $\theta_\nu : V(G) \to \{true, false\}$ und $\theta_\epsilon : E(G) \to \{true, false\}$ sind Prädikatsfunktion, die auf den Label und Properties der Knoten bzw. Kanten definiert sind.*

Eine Möglichkeit, um die Ähnlichkeit zwischen zwei Graphen zu beschreiben, ist die *Dual Simulation*.

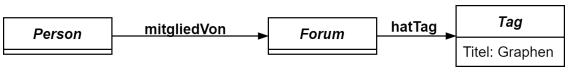
Definition 24 ([54])**.** *Sei G ein Windowed (Property-)Graph, $G' = (V', E')$ ein Teilgraph von G und $Q = (V_q, E_q)$ ein Abfragegraph. Dann besteht G' eine **Dual Simulation-Übereinstimmung** zu Q ($Q \prec_D G$), wenn eine Relation $S \subseteq V_q \times V'$ existiert für deren Elemente $(v_1, v_2) \in S$ gilt:*

- $\theta_\nu(v_2) = true$

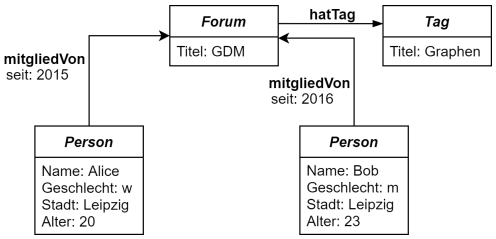
[6] Bilder basierend auf [52, Abb. 1c]



(a) Ausgangsgraph.[7]



(b) Abfragegraph: Personen, die sich gegenseitig kennen.



(c) Ein aus dem Pattern Matching resultierender Teilgraph.

Abbildung 4.5.: Beispiel für Pattern Matching mit Dual Simulation.

- $(v_1, w_1) \in E_q \implies \exists w_2 \in V' : (v_2, w_2) \in E' \wedge \theta_\epsilon((v_2, w_2)) = true \wedge (w_1, w_2) \in S$
- $(u_1, v_1) \in E_q \implies \exists u_2 \in V' : (u_2, v_2) \in E' \wedge \theta_\epsilon((u_2, v_2)) = true \wedge (u_1, u_2) \in S$

Die leere Menge ist eine triviale Dual Simulation-Übereinstimmung. Es gibt eine maximale Dual Simulation-Übereinstimmung für jeden Abfragegraphen und Graphen, da die Vereinigung zweier Übereinstimmungen wieder eine Übereinstimmung ergibt.

Aus den vorherigen Definitionen kann nun ein Operator abgeleitet werden, der Pattern Matching ermöglicht.

Definition 25. *Aus einem gegebenen Windowed Graphen G und einem Abfragegraphen Q gibt der* ***Pattern Matching-Operator*** *eine Menge von neuen Graphen $\mathcal{G}$ aus, sodass $G' \in \mathcal{G} \Leftrightarrow G' \sqsubseteq G \wedge Q \prec_D G'$.*

Abbildung 4.5 zeigt eine beispielhafte Anwendung des Pattern Matching-Operators auf einem Window-Graph mit Dual Simulation. In 4.5b sieht man den Abfragegraphen, der auf den Ausgangsgraphen angewendet wird. Die Prädikatsfunktionen können dabei aus der Abbildung abgeleitet werden, bspw. gilt $\theta_\epsilon(e) = true \Leftrightarrow \tau(e) = mitgliedVon \vee tau(e) = hatTag$. Daraus resultiert in diesem Fall genau ein Teilgraph (vgl. Abbildung 4.5c).

4.3.3. Hilfsoperator - Reduce-Combination

Der *Reduce-Combination*-Operator reduziert einen Graph-Collection-Stream auf einen einzigen Graphstream. Dieser Operator ist notwendig, da keiner der implementierten Operatoren Graph Collection-Streams als Eingabe nutzen kann. Damit ist er eine Implementierung der Graph Collection-zu-Graph-Operatoren.

Definition 26. *Aus einem gegebenen Property-Graphstream $PGS = (ES, L, T, \tau, K, A, \kappa)$ gibt der* ***Reduce-Combination-Operator*** *einen reduzierten Property-Graphstream $PGS' = (ES, L', T, \tau, K, A, \kappa)$ mit $|L'| = 1$ zurück.*

[7] Basierend auf [52, Abb. 1a]

Da der Reduce-Combination-Operator die einzige Reduce-Variante in dieser Arbeit ist (im Gegensatz zu Gradoop, welches mehrere Varianten bietet[8]), wird er im Folgenden auch verkürzt nur Reduce-Operator genannt.

[8] vgl. `https://github.com/dbs-leipzig/gradoop/wiki/Auxiliary-Operators#Reduce`

5. Implementierung

Im folgendem Kapitel wird das im Umfang der vorliegenden Arbeit implementierte End-zu-End-Graphstream-Analyse-Frameworks *GRAFS* (**GR**aph **A**nalysis on **F**link's Data**S**treams) vorgestellt. GRAFS wurde in Java geschrieben und baut dabei auf der `DataStream`-API von Apache Flink auf. Ziel war es, ein System zu erstellen, das das im Entwurf vorgestellte PGS-Modell und dessen Operatoren abbilden kann und Erweiterbarkeit bietet. Zudem sollte sich die Syntax an GrALa orientieren und somit das in Gradoop als auch Flink mögliche Hintereinanderausführung von mehreren Operatoren/Transformationen (hier *Operator-Chaining* genannt) ebenfalls realisiert werden. Zusätzlich soll dabei die Nutzung der `DataStream`-API möglichst abstrahiert werden, sodass die genaue Art des unterliegenden Datenstreams (vgl. Abbildung 2.9) für den Nutzer verborgen bleibt.

Hierfür wird das Modell in drei Teile unterteilt: Die Elemente des Streams, der Stream selbst und die Operatoren, die auf den Stream angewendet werden können. Code 1 zeigt einen vereinfachten Ausschnitt dieser drei Modelle. Das `Triplet` stellt dabei die Elemente des PGS dar. Ein Flink `DataStream` der Triplets wird vom `GraphStream` gehalten. Die `callFor`-Methoden erlauben die Anwendung von Operatoren auf dem Graphstream, die das jeweilige Interface implementieren. Den Operatoren werden bei Erstellung durch den Aufruf des Konstruktors eventuelle Parameter übergeben, bei dem Subgraph-Operator bspw. die jeweiligen Filter. Wird ein so erzeugter Operator an die `callFor`-Methode (im Beispiel `callForGraph`) übergeben, erhält dieser den `DataStream` und wird mit ein oder mehreren Flink-Transformationen transformiert. Der transformierte `DataStream` wird danach an das `GraphStream`-Objekt zurückgegeben. Dort wird der `DataStream` schließlich innerhalb der `callFor`-Methode in einen neuen `GraphStream` (oder in einen `GCStream` für Graph-Collection-Streams) gekapselt und zurückgegeben. In Code 2 sieht man ein Beispiel des dadurch möglichen Operator-Chainings.

Für die Umsetzung müssen diverse Anforderungen erfüllt werden. Wie diese Anforderungen aussehen und wie sie umgesetzt wurden, wird in den folgenden Abschnitten erläutert. Der nächste Abschnitt beschreibt die Umsetzung des PGSMs. Anschließend wird auf die Implementierung der Operatoren eingegangen.

5.1. Implementierung des PGSM

Die Umsetzung des PGSMs wird hier in zwei Abschnitte unterteilt. Zunächst wird auf die Umsetzung der im Stream befindlichen Elemente eingangen. Anschließend wird das Stream-Modell selbst vorgestellt.

5.1.1. Implementierung der Elemente des PGSM

Bei der Umsetzung des PGSM war zu beachten, dass ein Element aus dem PGS alle nötigen Information (i. e. Label, Properties und Graphzugehörigkeit) für die Kante und seine Knoten mitführen muss, damit aus dem Stream allein der Graph vollständig rekonstruiert werden kann. Zusätzlich

```
1  class Triplet {
2    Edge edge;
3    Vertex sourceVertex;
4    Vertex targetVertex;
5  }
6
7  class GraphStream {
8    DataStream<Triplet> stream;
9
10   GraphStream(DataStream<Triplet> stream){
11     this.stream = stream;
12   }
13
14   GraphStream callForGraph(GraphToGraphOperatorI operator){
15     stream = operator.execute(stream);
16     return new GraphStream(stream);
17   }
18 }
19
20 class SomeOperator implements GraphToGraphOperatorI{
21
22   SomeOperator(/*Parameters for Operator*/){
23     // ...
24   }
25
26   DataStream<Triplet> execute(DataStream<Triplet> stream){
27     DataStream<Triplet> manipulatedStream = //... manipulate stream with given
           parameters using Flink's transformations
28     return manipulatedStream;
29   }
30 }
```

Code 1: Vereinfachter Modellierung für die GRAFS-Klassen.

musste sichergestellt sein, dass der Nutzer bei Anwendung eines Operators nur auf genau die Informationen Zugriff hat, die er benötigt, um unvorgesehene Änderungen zu vermeiden.

Für die Modellierungen der Knoten und Kanten wurde sich an Gradoop orientiert, sodass diese von `GraphElement` erben und dessen Elternklasse wiederum `Element` ist. Eine Übersicht der Klassenhierachie ist in Abbildung 5.1 zu sehen (auf Getter-, Setter- und Hilfsmethoden wie z. B. toString wird aus Übersichtsgründen verzichtet).
Ein `Element` enthält seine ID, seine Properties und sein Label. `GraphElement` wiederum erweitert `Element` lediglich um die Zuweisung von Graphen für die Graphelemente durch Graph-IDs. Die Graph-IDs enthalten die IDs aller Graphen, zu denen das Grapelement gehört. Die Klassen für die ID (*GradoopId*), die Graphzugehörigkeit (*GradoopIdSet*) und die Properties wurden von Gradoop importiert. Die Unterteilung von `GraphElement` und `Element` ermöglicht es, dass eine spätere Nachimplementierung der Graphen nach dem EPGM (i. e. mit Properties und Label) theoretisch möglich ist. Schließlich erweitern die `Knoten`- und `Kanten`-Klasse die `GraphElement`-Klasse. Zusätzlich werden die IDs der zu einer `Kante` zugehörigen `Knoten` in der `Kanten`-Klasse abgespeichert.

Schließlich werden die Knoten und Kanten in einem Objekt zusammengefasst. Inspiriert durch den RDF-Stream wurde hier der Name *Triplet* gewählt. Ein Triplet hält ein Kanten-Objekt und die

```
1  GCStream transformedGC = graphStream
2    .callForGraph(new VertexInducedSubgraph(/*...*/))
3    .callForGraph(new EdgeTransformation(/*...*/))
4    .callForGC(new NewlyDefinedG2GCOperator(/* parameters for operator*/));
```

Code 2: Beispiel von Operator Chaining in GRAFS mit bereits definierten Operatoren und einem vom User definierten Operator (letzte Zeile).

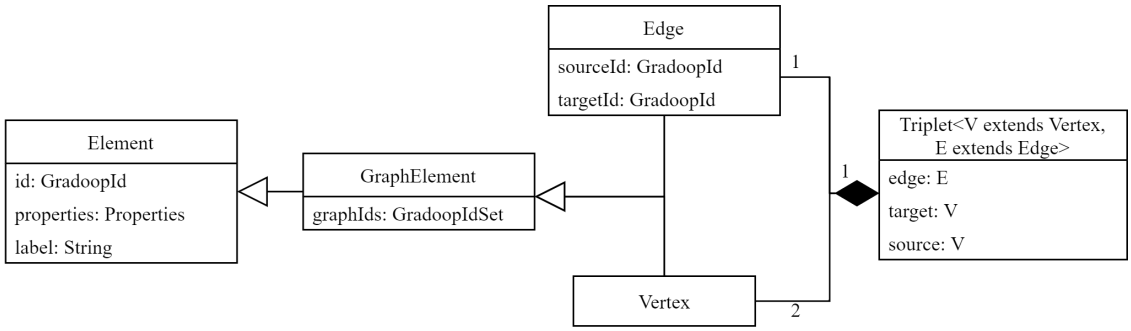
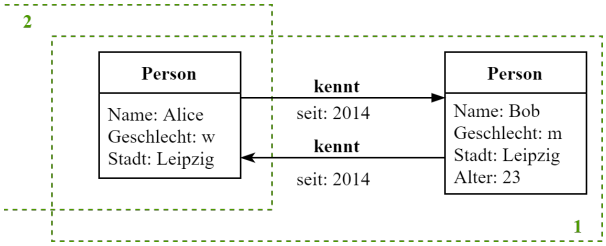


Abbildung 5.1.: Kapselung der Graphelemente durch Triplet.

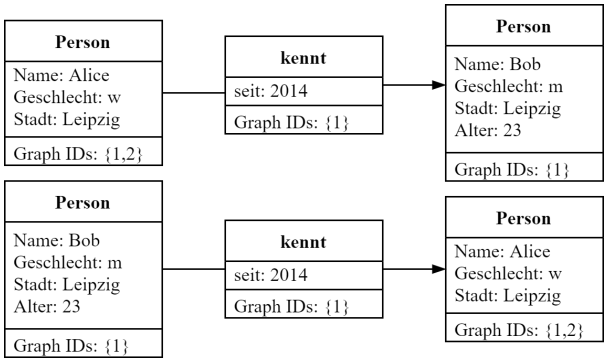
zwei zur Kante zugehörigen Knoten-Objekte.
Die im Stream befindlichen Elemente können sich wie in Abbildung 5.2 vorgestellt werden. In 5.2a sieht man den Ausschnitt einer Graph Collection. Der Knoten *Alice* ist dabei Teil von zwei Graphen. In 5.2b sind zwei Triplets, die den Teilgraphen bilden und einzeln als Stream fließen. In den weiteren Abbildungen werden die Graph-IDs meistens weggelassen, da sie für die implementierten Operatoren, die auf einzelnen logischen Graphen arbeiten, keine Rolle spielen.

5.1.2. Umsetzung der Streamrepräsentation

Im PGSM wird, wie im EPGM, zwischen logischen Graphen und Graph Collections unterschieden. Dementsprechend wurde die Stream-Klasse um die zwei Klassen `GraphStream` und `GCStream` (von *GC* kurz für *Graph Collection*) erweitert. Abbildung 5.3a zeigt, welche Operatoren auf die jeweiligen Stream-Arten angewendet werden können und die jeweiligen Übergänge. Beispielsweise

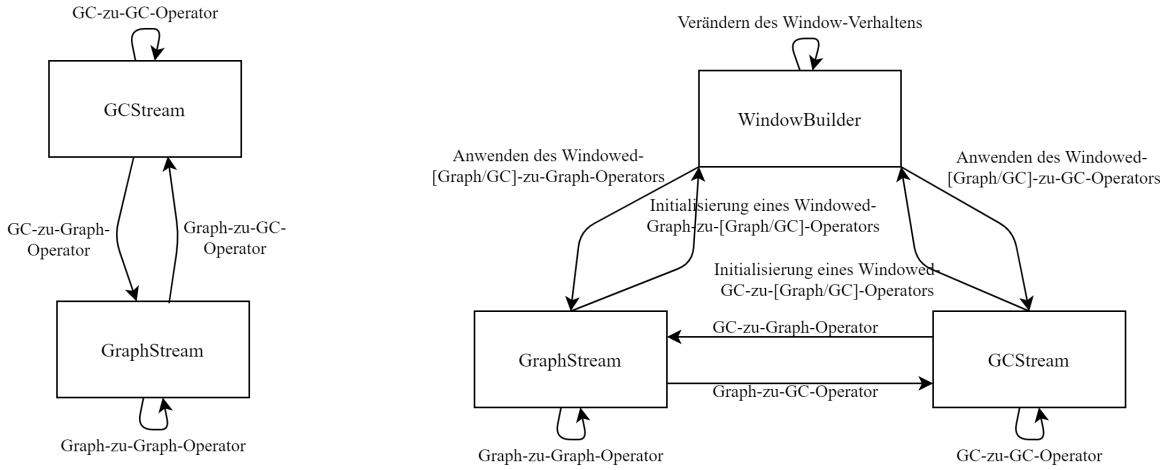


(a) Ausschnitt einer Graph Collection, die aus einem PGS ensteht.



(b) Elemente des Graphen im Stream.

Abbildung 5.2.: Darstellung der im Stream befindlichen Elemente.



(a) Übergänge bei Graphstreams und Graph-Collection-Streams.

(b) Übergange mit Windowing.

Abbildung 5.3.: Übergänge in andere Objekte innerhalb des PGSM.

kann Subgraph, ein Graph-zu-Graph-Operator, auf einen Graphstream angewendet werden und gibt danach wieder einen Graphstream aus.

Zusätzlich muss darauf geachtet werden, dass bestimmte Operatoren Windowing nutzen. Dementsprechend muss vor der Ausführung des Operators noch ein Window Assigner gebaut werden. Bei der Umsetzung wurde darauf geachtet, dass die darunterliegenden Flink-Stream-Arten (wie bspw. `WindowedStream`, `DataStream`, vgl. auch Abbildung 2.9) möglichst abstrahiert sind, sodass ein Nutzer der Stream-Klassen keine Kenntnisse über die Streamarten braucht. Zusätzlich soll das Windowing so gestaltet sein, dass eine einfache Erweiterung der Operatoren ermöglicht wird.

Abbildung 5.3b erweitert das in 5.3a gezeigte Modell um die Übergänge für Windowed Operatoren. Zunächst wird die Erstellung der Windowing-Informationen eingeleitet, indem ein `WindowBuilder` erzeugt wird. Dieser ist an Flinks `WindowOperatorBuilder` angelehnt. Die Windowing-Informationen halten den initialen Window Assigner und ggf. weitere Informationen, wie bspw. Trigger oder Evictor. Code 3 zeigt ein Beispiel für einen solchen Vorgang. Mit der `window`-Methode wird der `WindowBuilder` erzeugt und bekommt einen initialen Window Assigner basierend auf dem `GlobalWindows.create()`-Aufruf. Anschließend können die Windowing-Informationen mit weiteren Aufrufen des `WindowBuilder`-Methoden verändert werden (im Beispiel mit der `trigger`-Methode). Statt die so erstellten Windowing-Informationen direkt auf dem Flink-Stream anzuwenden und den entstehenden Stream (`WindowedStream`/`AllWindowedStream`) an den Operator weiterzugeben, werden beim Aufruf der entsprechenden `callFor`-Methode des `WindowBuilder`s (im Beispiel `callForGraph`) stattdessen die Windowing-Informationen an den Operator übergeben, der diesen für die Aufrufe der Flink-Funktionen nutzt. Dieses Vorgehen hat den Vorteil, dass jeder Operator selbst die Art des Windowed Streams bestimmen kann und die Windowing-Informationen gegebenenfalls auch mehrfach innerhalb eines Operators genutzt werden können (vgl. Unterunterabschnitt 5.2.1.3). Nach der Anwendung des Windowed Operators auf den Graphstream wird schließlich, wie bei herkömmlichen Operatoren, die jeweilige Graphstream-Art zurückgegeben.

```
1  GraphStream stream = graphStream
2  .window(GlobalWindows.create()) // Start building Windowing Information with initial
       Window Assigners
3  .trigger(/*Some trigger which should be executed on the window*/)
4  .callForGraph(new SomeWindowedG2GStreamOperator(/*...*/))
```

Code 3: Beispiel für die Nutzung des Window-Builders.

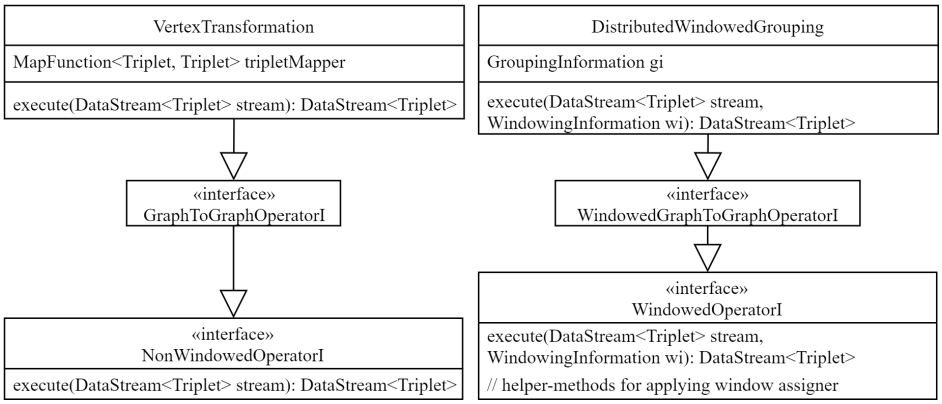


Abbildung 5.4.: Auszug aus den Klassendiagramm für Operatoren.

Erstellung der Streams und Ausgabe Neben den im nächsten Abschnitt beschriebenen Graph-Operatoren wurden zusätzlich Operatoren eingebaut, die für Ein- und Ausgabe des Streams verantwortlich sind. `GraphStream` und `GCStream` können durch die Nutzung des Konstrukturs instanziiert werden, in dem die Konfiguration und ein `DataStream` von Triplets übergeben wird. Alternativ können die `fromSource`-Methoden genutzt werden, die aus der gegebenen (Flink) `SourceFunction` den Graph(mengen)-Stream bauen. Für die Ausgabe wurden die Methoden `addSink` und `print` implementiert. Die in `addSink` mitgegebene (Flink) `SinkFunction` wird direkt an den im GraphStream befindlichen `DataStream` mit derselben Methode weitergeben. Flink selbst bietet einige Quell- und Zielfunktionen an, die bspw. das Lesen und Schreiben von einem Kafka-Broker erlauben, die sich hier anwenden lassen.

5.2. Operatoren

Wie bereits erwähnt, musste bei der Umsetzung des Operatoren-Modells darauf geachtet werden, dass dieses erweiterbar ist, sodass von einem Nutzer neue Operatoren definiert werden können, die auf den Stream angewendet werden können.

Um dies zu erreichen, werden Operatoren in Klassen definiert. Dabei unterscheidet man zwischen Windowed Operatoren und Nicht-Windowed Operatoren, sowie die erwartete Graphstream-Art bei Eingabe und Ausgabe, i. e. ob der Operator auf Graphstreams oder Graph-Collection-Streams angewendet wird und welche Art von Graphstream ausgegeben wird. Abbildung 5.4 zeigt einen Auschnitt der Operator-Klassen und -Interfaces. Insgesamt gibt es zur Zeit jeweils 4 Interfaces, die `NonWindowedOperatorI` bzw. `WindowedOperatorI` erweitern.

```
1  subgraphStream = v.callForGraph(new Subgraph(
2    v -> v.getProperty("Name").startsWith("A"),
3    null, /*not needed for vertex induced subgraph*/,
4    Subgraph.Strategy.VERTEX_INDUCED));
5
6  // OR
7  subgraphStream = graphStream
8    .vertexInducedSubgraph(v -> v.getProperty("Name").startsWith("A"));
```

Code 4: Anwendungsbeispiel für Subgraph auf einen Graphstream.

Parameter der Operatoren Wie im Entwurf deutlich wird, nutzen die Operatoren Parameter. Es wurde bereits erwähnt, dass diese innerhalb des Konstruktors der jeweiligen Operatoren übergeben werden. Dabei wurde darauf geachtet, dass die Eingabe der Parameter möglichst nutzerfreundlich ist.
Dementsprechend wurde, soweit möglich, die Nutzung von UDFs ermöglicht, bspw. Prädikatsfunktionen oder Aggregationsfunktionen. Diese werden über die Flink-Interfaces realisiert, z. B. die Prädikatsfunktionen über Flinks `FilterFunction`. Damit ist die Definition der UDFs über Javas Lambda-Ausdrücke möglich.

5.2.1. Graph-zu-Graph

Zunächst wird hier auf die Subgraph- und Transformation-Operatoren eingegangen, die durch die zur Verfügung stehenden Transformationen in Flink einfach realisierbar waren und damit leichter verständlich sind. Anschließend wird auf den Grouping-Operator eingegangen.

5.2.1.1. Subgraph

Für die Umsetzung des Subgraph-Operators wird an den Konstruktor die jeweilige Prädikatsfunktion übergeben, sowie eine *Strategie*, welcher die Art des Subgraphs bestimmt (`VERTEX_INDUCED` für knoteninduzierten, `EDGE_INDUCED` für kanteninduzierten oder `BOTH` für den Subgraph-Operator, bei dem beide Prädikatsfunktionen angewendet werden). Code 4 zeigt ein Beispiel einer solchen Anwendung mit der `callForGraph`-Methode. Darüber hinaus wurden Hilfsmethoden implementiert, die die Nutzung der Operatoren erleichtern (Zeile 7f.).

Die Prädikatsfunktionen können dabei `null` sein, wenn sie für die gewählte Strategie nicht gebraucht werden. Bspw. ist für knoteninduzierter Subgraph kein Kantenprädikat nötig. Die Implementierung des Subgraph-Operators ist durch die Nutzung der Filter-Transformation von Flink unkompliziert. Code 5 zeigt einen Ausschnitt der `Subgraph`-Klasse. Aus den gegebenen Informationen wird eine Prädikatsfunktion für Triplets gebaut. Die Implementation ergibt sich direkt aus der Definition des Operators: Für einen knoteninduzierten Subgraph müssen beide Knoten das Knoten-Prädikat erfüllen um Teil des ausgegebenen Streams zu sein, für einen kanteninduzierten Subgraph muss die Kante das Kanten-Prädikat erfüllen und für einen herkömmliche Subgraph müssen sowohl die Knoten als auch die Kante ihr jeweiliges Prädikat erfüllen.

```
1  public class Subgraph implements GraphToGraphOperator {
2    private final FilterFunction<TripletFilter> tripletFilter;
3
4    public Subgraph(FilterFunction<Vertex> vertexFilter, FilterFunction<Edge>
         edgeFilter, Strategy strategy){
5      switch(strategy) {
6        case VERTEX_INDUCED: tripletFilter = createVertexInducedSubgraphFilter(
             vertexFilter);
7        /* Other strategies are implemented in a similar way*/
8      }
9    }
10
11   private FilterFunction<Triplet> createVertexInducedSubgraphFilter(FilterFunction<
         Vertex> vertexFilter) {
12     return triplet ->
13         vertexFilter.filter(triplet.getSourceVertex()) && vertexFilter
14             .filter(triplet.getTargetVertex());
15   }
16
17   public DataStream<Triplet> execute(DataStream<Triplet> stream) {
18     return stream.filter(tripletFilter);
19   }
20 }
```

Code 5: Auszug aus der Subgraph-Klasse.

```
1  subgraphStream = graphStream.callForGraph(new VertexTransformation(
2    v -> {
3      var age = v.getProperty("age").getInt();
4      if(age >= 18){
5        v.setProperty("adult", true);
6      } else{
7        v.setProperty("adult", false);
8      }
9      return v;
10   }));
```

Code 6: Anwendungsbeispiel für Knoten-Transformation auf einen Graphstream.

Das so gebaute Triplet-Prädikat wird zwischengespeichert und innerhalb der `execute`-Methode auf den Stream angewendet, indem die Filter-Transformation des *DataStreams* genutzt wird.

5.2.1.2. Transformation

Es wurden `EdgeTransformation` für Kantentransformationen und `Vertextransformation` für Knotentransformationen implementiert. Beide nehmen eine nutzerdefinierte Transformationsfunktion (Flinks `MapFunction`-Interface) auf der `Element`-Klasse an. Die Nutzung von Element als Typ des Eingabeparameters garantiert, dass die Graphzugehörigkeit der Graphelemente nicht verändert werden kann und in der Kanten-Transformation im Speziellen die Knoten-IDs nicht verändert werden können. Genau wie beim Subgraph-Operator wird das jeweilige Transformationsobjekt an die `callForGraph`-Methode übergeben oder es wird die implementierte Hilfsmethode genutzt (im Beispiel nicht enthalten).

```
1  public class VertexTransformation implements GraphToGraphOperator {
2
3    final MapFunction<Triplet<Vertex, Edge>, Triplet<Vertex, Edge>> tripletMapper;
4
5    public VertexTransformation(final MapFunction<Element, Element> transformFunc) {
6      this.tripletMapper = new MapFunction<Triplet<Vertex, Edge>, Triplet<Vertex, Edge
           >>() {
7        @Override
8        public Triplet<Vertex, Edge> map(Triplet<Vertex, Edge> triplet) throws
             Exception {
9          Vertex source = (Vertex) transformFunc.map(triplet.getSourceVertex());
10         Vertex target = (Vertex) transformFunc.map(triplet.getTargetVertex());
11         return new Triplet<>(triplet.getEdge(), source, target);
12       }
13     };
14   }
15   @Override
16   public DataStream<Triplet<Vertex, Edge>> execute(DataStream<Triplet<Vertex, Edge>>
         stream) {
17     return stream.map(tripletMapper);
18   }
19 }
```

Code 7: Auszug aus der Knoten-Transformationsklasse.

Code 7 zeigt einen Ausschnitt aus der `VertexTransformation`-Klasse. Ähnlich wie beim knoteninduziertem Subgraph muss die Funktion auf beide Knoten-Elemente angewendet werden.

5.2.1.3. Windowed-Grouping

Die Implementierung von Windowed-Grouping wurde hier auf zwei Arten realisiert, zum einen in einer verteilten und zum anderen in einer zentralen Variante. Code 8 zeigt ein Beispiel der Anwendung des verteilten Windowed-Groupings (die zentrale Variante kann analog mit entsprechendem Klassennamen genutzt werden). Beide Arten bekommen im Konstruktor für die Knoten und Kanten jeweils eine Menge von Grouping Keys und eine Menge von Aggregationsfunktionen. Alternativ kann das Grouping auch mit einem `GroupingBuilder` erstellt werden (Zeile 15ff.)

Für das Design der Aggregationsfunktionen wurde sich an Gradoop orientiert. Dementsprechend gibt es neben `Count` weitere bereits implementierte Aggregationsfunktionen, die ebenfalls in Gradoop vorhanden und in Tabelle 5.1 zu sehen sind. Darüber hinaus könnten weitere Aggregationsfunktionen implementiert werden, indem eines der dazugehörigen Interfaces (bspw. `AggregateFunction` oder `Sum`) genutzt wird.

Die nachfolgenden Paragraphen erläutern, wie die beiden Grouping-Verfahren im Detail funktionieren.

Verteiltes Windowed-Grouping Bei dem verteilten Grouping wird die Aggregation in drei Phasen unterteilt: zwei Knoten-Aggregationsphasen (für Quell- und Zielknoten) und eine Kanten-Aggregationsphase. Um diese Phasen besser zu erläutern wird ein laufendes Beispiel mit Bildern

```
class DistributedWindowedGrouping{
  public DistributedWindowedGrouping(Set<String> vertexKeys, Set<AggregateFunction>
      vertexAggFunctions,
    Set<String> edgeKeys, Set<AggregateFunction> edgeAggFunctions)
  {/*...*/}
  //...
}

windowStreamBuilder = graphStream.callForGraph(new DistributedWindowedGrouping(
  Set.of(":label","Stadt"), // Vertex Grouping Keys
  Set.of(new Count("count"), // Vertex Aggregation Functions
  Set.of("seit"),          // Edge Grouping Keys
  new Count("countEdge")) // Edge Aggregation Functions
));

// Or with GroupingBuilder
windowStreamBuilder = graphStream.callForGraph(
  DistributedWindowedGrouping.createGrouping()
  .useVertexLabel(true)
  .addVertexGroupingKey("Stadt")
  .addEdgeGroupingKey("seit")
  .addVertexAggregateFunction(new Count("count"))
  .addEdgeAggregateFunction(new Count("countEdge"))
  .build()
);

groupedGraphStream = windowStreamBuilder
  .withWindow(TumblingEventTimeWindows.of(Time.minutes(10)))
  .apply();
```

Code 8: Anwendungsbeispiel für verteiltes Grouping auf einem Graphstream.

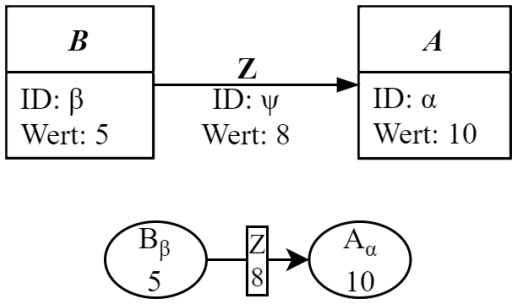
geführt. Ein Bild mit dem vollständigen Ablauf kann im Anhang in Abbildung A.1 gefunden werden.

Abbildung 5.5a zeigt zunächst eine Vereinfachung der grafischen Repräsentation, die für das Beispiel genutzt wird. Für das Beispiel sollen die Knoten anhand ihres Labels gruppiert und der Wert in den Knoten summiert werden. Die Kanten sollen ebenfalls anhand ihres Labels gruppiert und lediglich gezählt werden.
Auf die Repräsentation der IDs einer Kante wird für das Beispiel verzichtet und angenommen, dass alle Kanten unterschiedliche IDs besitzen. Die IDs der Knoten werden nach unten gestellt und an das Label angehangen. Der Property Value des *Wert*-Properties wird unter das Label geschrieben. Abbildung 5.5b zeigt den hier genutzten Window-Graphen und den aus dem Grouping resultierenden Graphen.

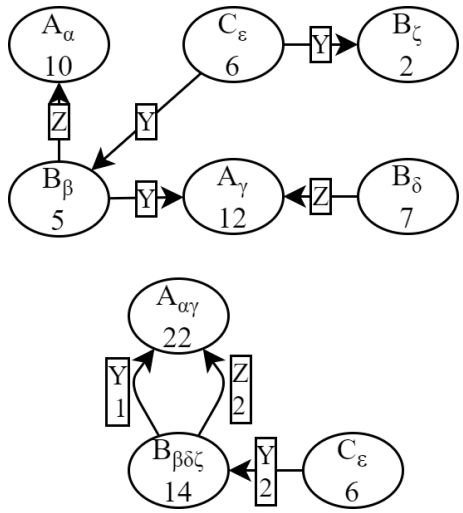
In jeder Phase werden die Triplets im Stream mithilfe der `keyBy`-Methode so auf Worker aufgeteilt, dass sich innerhalb eines Substreams nur Triplets befinden, dessen Property-Werte der gerade betrachteten Graphelemente für alle Grouping Keys gleich sind. Anschließend wird ein Window innerhalb dieser aufgeteilten Streams erzeugt. Auf diesen wird dann die jeweilige Aggregation vorgenommen.

Name	Beschreibung
Count	Zählt die gruppierten Elemente
MaxProperty	Berechnet das Maximum eines numerischen Property Values der gruppierten Elemente
MinProperty	Berechnet das Minimum eines numerischen Property Values der gruppierten Elemente
SumProperty	Berechnet die Summe eines numerischen Property Values über alle gruppierten Elemente

Tabelle 5.1.: Übersicht über die umgesetzten Aggregationsfunktionen.



(a) Veranschaulichung der vereinfachten Repräsentation.



(b) Grouping-Beispiel, oben der Windowed Graph, unten der auf den Labeln gruppierte Graph.

Abbildung 5.5.: Vorlage für das laufende Beispiel.

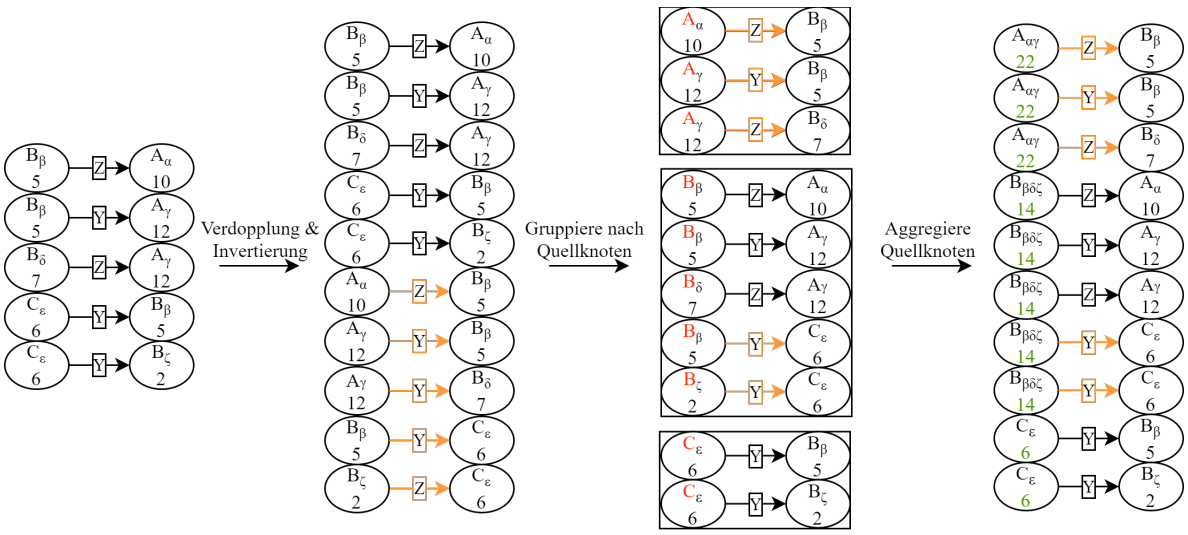


Abbildung 5.6.: Die ersten zwei Phasen des Groupings: Verdopplung der Triplets, Gruppierung und Aggregationsprozess.[9]

Knoten-Aggregation Innerhalb einer Knoten-Aggregationsphase wird immer nur ein Knoten der Triplets (Quell- oder Zielknoten) betrachtet. In der ersten Phase sind die Triplets anhand des Quellknoten aufgeteilt, d. h. alle Quellknoten in einem Stream haben dieselben Property-Werte untereinander für alle Grouping Keys der Knoten. Innerhalb des Workers wird dann über die Quellknoten das Window erzeugt und in diesem aggregiert.
Zunächst gibt es die Problematik zu lösen, dass Knoten eventuell nur als Zielknoten, aber nie als Quellknoten vorhanden sind (oder umgekehrt). Ein solcher Knoten würde nicht korrekt mitaggregiert werden können, da er aufgrund der Aufteilungsstrategie nie mit dem ihm zugehörigen Knoten in einem Worker ist. Dementsprechend muss vor der ersten Aggregationsphase der Stream so verändert werden, dass für jedes Triplet ein inverses Triplet zum Stream hinzugefügt wird, in welchem Quell- und Zielknoten vertauscht sind. Diese Kanten werden gesondert markiert und später aus dem Stream herausgefiltert.

Abbildung 5.6 zeigt den Invertierungsprozess sowie die Gruppierung und Aggregierung auf den Quellknoten. Die invertierten Kanten werden hier in orange angezeigt Nach der Gruppierung befinden sich in dem Substream bzw. im daraus entstehenden Window (in der Abbildung eingerahmt) nur noch Triplets, deren Quellknoten-Label gleich sind. Die Knotenwerte innerhalb eines Windows werden entsprechend der Aggregationsfunktion aggregiert (i. e. summiert). Daraus entstehen die Superknoten. Bei der Aggregierung muss darauf geachtet werden, dass Knoten nicht doppelt aggregiert werden. Am Ende der Phase wird für jedes eingegangene normale Triplet (i. e. nicht invertiert) ein neues Triplet erzeugt, dass die alte Kante und den Zielknoten übernimmt und für den Quellknoten den entstandenen Superknoten einsetzt. Die ID für den Quellknoten innerhalb der Kante wird entsprechend angepasst. Dieses Triplet wird anschließend an den Stream übergeben, der die nächste Phase einleitet.

Die Zielknoten-Aggregation funktioniert analog zur Quellknoten-Aggregation: man gruppiert in der ersten Phase die Triplets nach den Zielknoten und aggregiert diese mit den Knoten-Aggregatfunktionen

[9] Eine Aktualisierung der invertieren Triplets ist theoretisch nicht notwendig, wird im Beispiel aber aus Einfachheit durchgeführt

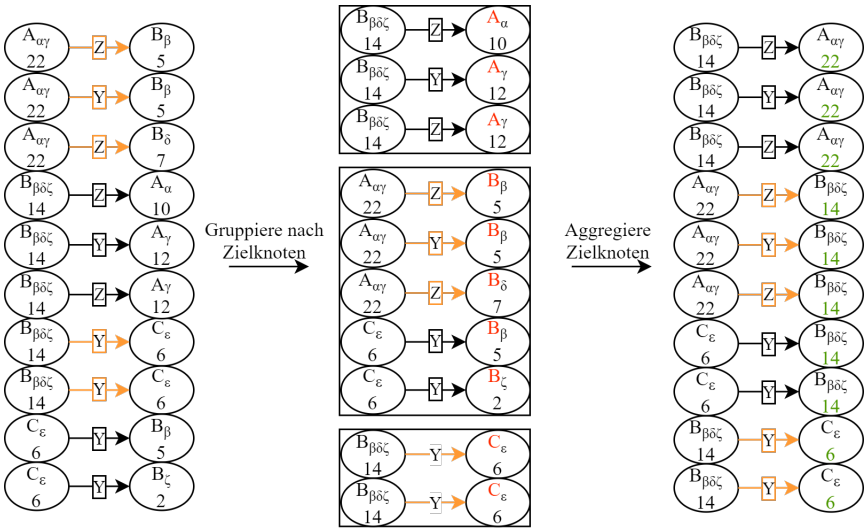


Abbildung 5.7.: Aggregationsprozess für die Zielknoten.

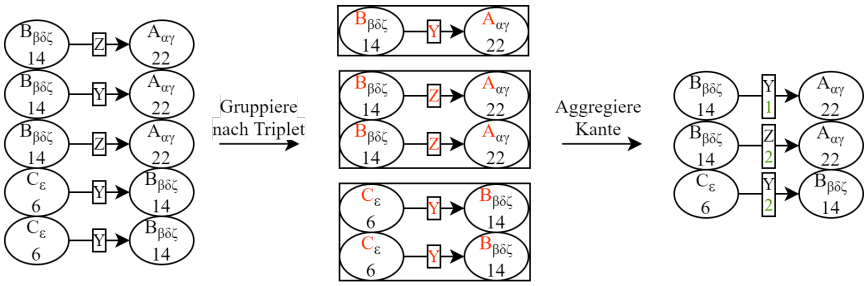


Abbildung 5.8.: Gruppierung und Aggregation der Kanten (invertierte Kanten sind bereits rausgefiltert).

(vgl. Abbildung 5.7). Der so entstandene Stream wird anschließend gefiltert, sodass keine invertierten Kanten im Stream vorhanden sind.

Kanten-Aggregation Für die Kantenaggregation wird der Stream so aufgeteilt, dass die Triplets nicht nur nach den Grouping Keys der Kanten aufgeteilt werden, sondern auch nach den Knoten. In einem Window sind dementsprechend alle Triplets, deren Kantenwerte für die Grouping Keys jeweils gleich sind und die Kanten denselben Startknoten und denselben Zielknoten haben. Abbildung 5.8 zeigt auf, wie diese für das Beispiel aufgeteilt sind.

Einschränkungen des verteilten Verfahrens Wie bereits erwähnt, benutzt diese verteilte Variante drei hintereinanderfolgende Windows, um das Grouping auszuführen. Dadurch ergeben sich mehrere Probleme mit verschiedenen Window-Assigner-Varianten.
Bei der Nutzung der Processing Time ändert sich diese nach jeder Transformationsanwendung. Da die Verarbeitung parallel stattfindet, kann es beisspielsweise passieren, dass ein Worker mehr Elemente hat und deutlich länger braucht als andere Worker, um seine Elemente zu verarbeiten und ihnen einen neuen Zeitstempel zuzuweisen. Dadurch könnten die Elemente, die in Phase 1 (oder 2) im selben Window waren, aber in zwei unterschiedlichen Workern aggregiert wurden, in der nächsten Aggregationsphase in zwei unterschiedliche Time Windows sind. Dadurch ist die Korrektheit des Operators nicht mehr gegeben.

Auch von der Nutzung von Sliding Windows ist abzuraten, da durch die Überlappung der Windows Elemente des (zeitlich gesehen) nächsten Windows Teil der zweiten Phase des vorherigen Windows werden können.

Zur Zeit ist im praktischen Rahmen nur die Nutzung der Event Time mit Tumbling Windows sinnvoll. Diese benötigt widerum eine vom Nutzer definierte Watermark-Strategy. Dementsprechend muss darauf geachtet werden, dass diese während des Prozesses immer noch anwendbar ist, i. e. die genutzten Felder im Triplet nach einer Knoten-Aggregation vorhanden sind und nicht so manipuliert werden, dass sie außerhalb des Windows für den nächsten Schritt fallen.
Aufgrund der Beschränkungen wurde ein weiteres Grouping erarbeitet, welches im nächsten Paragraph erläutert wird.

Windowed-Grouping auf einem Worker In der `AllWindowedGrouping`-Variante wird der Stream mit der `windowAll`-Methode des Flink `DataStream`s in Windows unterteilt und alle Elemente im Window werden an einen Worker im Cluster übergeben. Dieser führt anschließend die gesamte Berechnung für das Grouping aus. Für die Berechnung selbst werden zunächst die Triplets in ihre Knoten und Kanten aufgespalten und diese anhand der Grouping Keys aufgeteilt und gespeichert. Anschließend findet zunächst die Aggregation auf den Knoten, anschließend auf den Kanten statt.

Diese Variante sorgt dafür, dass innerhalb eines Windows das Grouping nach Definition ausgeführt wird. Allerdings wird dadurch die Parallelität des Streams auf eins gesetzt (da nur ein Worker für jeweils ein Window zuständig ist) und die Anwendung des Operators könnte somit einen starken Datenfluss-Engpass für einen Analyse-Prozess bilden.

5.2.2. Graph-zu-Graph-Collection

Im Folgendem wird der im Entwurf vorgestellte Pattern Matching Operator auf Basis von Dual Simulation erläutert. Eine Referenzimplementierung besteht mit [48] bereits. Auf dieser wurde hier aufgebaut, indem für die Verarbeitungsschritte das Design von *SGraPMa* [48] übernommen wurde. Der Algorithmus basiert auf [55] und wurde um die Möglichkeit Prädikatsfunktionen anzuwenden erweitert.

Als Eingabe erwartet der Operator eine *GDL*-Abfrage. GDL (kurz für *Graph Definition Language*, [56]) ist eine Graph-Abfragesprache, welche von Neo4j's Cypher [57] inspiriert wurde. Sie ermöglicht die Erstellung von EPGs und Abfragen auf diesen. Tabelle 5.2 zeigt einige Beispiele. Die Umsetzung kann mit Prädikatdefinitionen für Knoten im `WHERE`-Teil umgehen. Einschränkungen für die Kanten sind nur bei der Strukturdefinition des Graphen (i. e. `MATCH`-Teil) möglich (vgl. zweites Abfrage-Beispiel in Tab. 5.2). Code 9 zeigt, wie Dual Simulation in GRAFS angewendet wird.

Innerhalb der Implementierung wird aus der GDL-Abfrage zunächst ein Abfragegraph gebaut. Dieser beruht auf einer Erweiterung der Graphelemente, um Prädikatsspeicherung und Variablenzuordnung zu ermöglichen (vgl. Abbildung 5.9).

Beschreibung	Beispiel
Definition eines Knotens mit Personen-Label und Name als Eigenschaft	(alice:Person {Name: "Alice"})
Definition einer Kante mit Label	(alice)-[:kennt]->(bob)
Abfrage-Ausdruck: Alle die Bob kennen und älter als 25 sind	**MATCH** (a)-[:kennt]->(b) **WHERE** b.Name = "Bob" **AND** a.Alter > 25
Abfrage-Ausdruck: Alle die Bob seit 2014 kennen	**MATCH** (a)-[:kennt {seit: 2014}]->(b) **WHERE** b.Name = "Bob"

Tabelle 5.2.: Beispiele für Ausdrücke mit GDL.

```
1  windowStreamBuilder = graphStream.callForGC(
2    new DualSimulation(
3      "MATCH(a)-[:knows {since: 2014}]->(b)" +
4      "WHERE b.Name = \"Bob\"")
5  );
6
7  graphCollectionStream = windowStreamBuilder
8    .withWindow(TumblingEventTimeWindows.of(Time.minutes(10)))
9    .apply();
```

Code 9: Anwendungsbeispiel für Dual Simulation auf einem Graphstream.

Vorverarbeitung Die Verarbeitung ist in drei Phasen unterteilt (vgl. Abbildung 5.10). Die Triplets können in den ersten beiden Phasen parallel verarbeitet werden. Zunächst werden die eingehenden Triplets umgewandelt. Dies geschieht unter Anwendung von Flinks `DataStream.transform`-Methode. Die in den Triplets enthaltenen Graphelemente werden zu dem oben beschriebenen Modell portiert. Dadurch ist die Zuordnung von Variablen möglich.

Diese Eigenschaft wird in der zweiten Phase ausgenutzt, in der die entstandenen *Abfrage-Triplets* vorgefiltert werden. Ziel ist es, die Anzahl der potenziellen Kandidaten für die letzte Phase zu reduzieren.

Der mit der `DataStream.filter`-Methode angewendete Filter wendet mehrere Kriterien an: Für die Kante des betrachteten Triplets muss eine Übereinstimmung mit mindestens einer Kante aus dem Abfragegraphen bestehen, sodass alle in der Abfragekante definierten Properties, sowie ggf. das Label mit der betrachteten Kante übereinstimmen. Falls eine Übereinstimmung mit einer Ab-

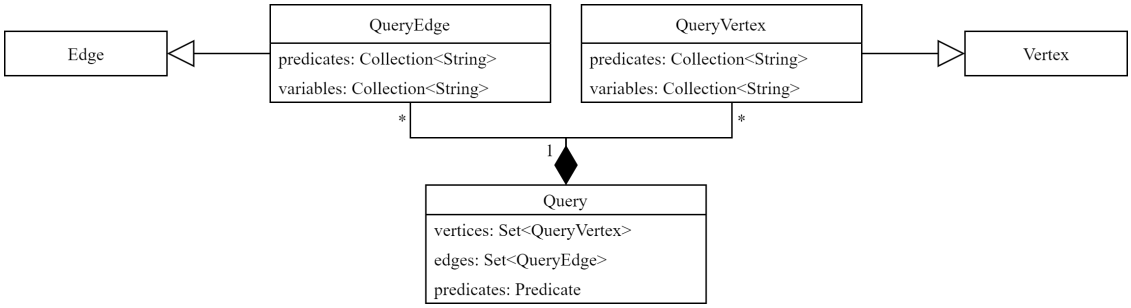


Abbildung 5.9.: Erweiterung der Graphelemente zur Speicherung von Variablennamen und Prädikaten.

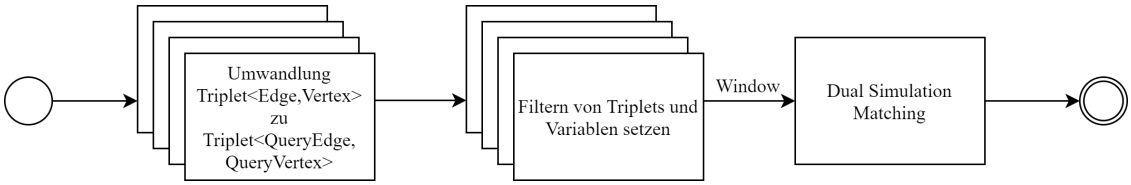


Abbildung 5.10.: Übersicht über den Datenfluss beim Pattern Matching.

```
1   Input: Triplets
2   Output: Filtered Triplets
3   Data: Query Graph Q
4   Function filterTriplet (t: Triplet):
5     matched ← false
6     for Triplet q ∈ Q:
7       if match(q,t):
8         if checkPredicate(q,t):
9           matched ← true
10          store variables of q in t
11        end
12      end
13    end
14    if matched:
15      emit t
16    end
17  end
```

Code 10: Pseudocode für den angewendeten Filter.

fragekante gefunden wurde, müssen die dazugehörigen Quell- bzw. Zielknoten ebenfalls mit dem jeweiligen Knoten im Triplet übereinstimmen. Schließlich wird für das Triplet die Prädikatsfunktionen evaluiert (vgl. auch Code 10), falls auf den generierten Prädikatenbaum folgendes zutrifft:

- Wenn die boolschen Verzweigungen des Baums nur aus `AND`-Operatoren bestehen
- Wenn ein Prädikat selbstbeschreibend ist, bspw. `a.Alter > 30`
- Wenn Prädikat die Beziehung zwischen Quell- und Zielknoten beschreibt, bspw. `q.Gehalt > z.Gehalt`

Ansonsten findet die Auswertung der Prädikatsfunktionen erst in der letzten Phase statt. Code 10 zeigt den Pseudocode für diesen Filter. Sollten alle Kriterien erfüllt sein, dann werden in den Graphelementen des Triplets die Variablen der übereinstimmenden Elemente des Abfragegraphen gespeichert, um eine Zuordnung während der letzten Phase zu vereinfachen. Sollte bspw. ein Knoten mit zwei Abfrageknoten übereinstimmen, denen die Variable `a` bzw. `b` zugeordnet ist, so werden beide Variablennamen in diesem Knoten gespeichert.
Der so entstandene Stream wird als initiale Kandidatenmenge für den Abfragegraphen behandelt. Alle weiteren Überprüfungen (bspw. die Überprüfung der in der Anfrage gegebenen Graphstrukturen) müssen innerhalb eines Windows stattfinden. Dementsprechend werden in der letzten Phase die übrigen Triplet-Elemente innerhalb eines Windows auf einem Worker gesammelt.

Anwendung von Dual Simulation im Window Der vorherige Prozess hat die Triplets vorgefiltert und die Variablen zugeordnet. Im letzten Schritt wird die Dual Simulation angewendet. Das Sammeln von allen Elementen in einen Worker geschieht, wie in `AllWindowedGrouping`, mithilfe von einem `AllWindowedStream` und der `process`-Methode. Dafür werden die im Window enthaltenen Triplets zunächst in einen Graphen umgewandelt und eine Abbildung erstellt, die

unter Nutzung der Variablen die Abfrageelemente den Graphelementen zuordnet. Anschließend werden diese Informationen an den eigentlichen Dual Simulation Algorithmus weitergegeben. Code 11 zeigt den Pseudocode dieses Prozesses. `G.child(`v`)` sind die Kinder eines Knotens v, i. e. `G.child(`v`)`$= \{v' | (v, v') \in E(G)\}$. Der Algorithmus aus [55] wurde um die Zeilen 26-29 und 31-35 erweitert. Die Implementierung der Funktion `checkPredicate` stammt aus [48] und prüft, ob das Element die aus der Anfrage extrahierten Prädikatsfunktionen erfüllt.

5.2.3. Hilfsoperator - Reduce

Die Implementierung des Hilfsoperators `Reduce` ist mit der Implementierung des Transformation-Operators vergleichbar. Auch diese nutzt Flinks `MapFunction` um die Triplets zu verändern. Es wird eine neue ID erstellt und die Graph-ID-Menge in den Triplet-Elementen auf diese ID gesetzt.

```
1   Data: Query Graph Q, Predicates
2
3   Input: Triplets with applied Variables T, Candidate Mapping Φ
4   Output: Stream of Triplets which have matches in Q via Dual Simulation
5   Function ProcessWindow(T : Triplets)
6     G ← make Graph from T
7     Φ ← make candidate map Φ : V(Q) → P(V(G)) by using the applied variables in T
8     Φ_final ← DualSim(G,Φ)
9     T ← extract Triplets using Φ_final and G
10    for t ∈ T:
11      add new graph ID to elements in t
12      emit t
13    end
14  end
15
16  Input: Window Graph G, Candidate Mapping Φ
17  Output: Candidate Mapping Φ updated to fit Dual Simulation
18  Function DualSim(G : Graph, Φ: Candidate Mapping):
19    changed ← true
20    while changed:
21      changed ← false
22      for u ∈ V(Q):
23        for u' ∈ Q.child(u):
24          Φ'(u') ← ∅
25          for v ∈ Φ(u):
26            if not checkPredicate(v):
27              remove v' from Φ(u)
28              continue
29            end
30            Φ_v(u') ← G.child(v) ∩ Φ(u')
31            for v' ∈ Φ_v(u'):
32              if not checkPredicate(v'):
33                remove v' from Φ_v(u')
34              end
35            end
36            if Φ_v(u') = ∅:
37              remove v from Φ(u)
38              if Φ(u) = ∅:
39                return empty Φ
40              end
41              changed ← true
42            end
43            Φ'(u') ← Φ'(u') ∪ Φ_v(u')
44          end
45          if Φ'(u') = ∅:
46            return empty Φ
47          end
48          if |Φ'(u')| < |Φ(u')|:
49            changed ← true
50          end
51          Φ(u') = Φ(u') ∩ Φ'(u')
52        end
53      end
54    end
55    return Φ
56  end
```

Code 11: Pseudocode für das Verarbeiten des Windows.

6. Evaluation

Im folgendem Kapitel wird die Performance der Implementierung *GRAFS* evaluiert. Da es, soweit bekannt, keine vergleichbaren Systeme für eine Gegenüberstellung gibt, werden die Operatoren in GRAFS stattdesssen alleinstehend getestet und, soweit möglich, miteinander verglichen. Auf die Evaluation des Hilfsoperators *Reduce* wird dabei aufgrund seiner Implementierungsähnlichkeit zu den Tranformationsoperatoren verzichtet. Da sich die Arbeitsweise von Windowed Operatoren stark von der Nicht-Windowed Operatoren unterscheidet (ersteres speichert/sammelt Elemente), werden die Ergebnisse nach Windowed und Nicht-Windowed Operatoren aufgeschlüsselt. Abschließend wird eine mögliche Analyse-Pipeline für den Datensatz mit mehreren Operatoren vorgestellt und deren Performance ebenfalls evaluiert. Diese Pipeline wird im folgendem *Citi Bike-Pipeline* genannt. Hier wurden zunächst die sich in den Knoten befindlichen Höhen- und Breitengrade genutzt, um wie in [58] Gitterzellen-IDs zu generieren und in den Knoten zu speichern (Knoten-Transformation). Auf dem so enstandenen Stream wird das Verteilte Grouping angewendet, in dem die Knoten nach ihren Gitterzellen gruppiert werden und pro Zelle gezählt werden. Schließlich werden nur die Kanten ausgegeben, deren Gitterzellen-ID einen bestimmten Wert überschreiten (Knoten-induzierter Subgraph). Die entsprechende Pseudocode-Implementierung befindet sich im Anhang, ebenso wie alle für die Operatoren-Messungen genutzten UDFs.

Umgebung und Datensatz der Evaluation Die Evaluation findet auf einem Big Data Cluster statt, welches durch Prof. Rahm von der Datenbank-Abteilung der Universität Leipzig zur Verfügung gestellt wurde. Hierfür stehen insgesamt 18 Maschinen zur Verfügung, zwei davon dienen als Master zur Verwaltung der Flink-Jobs, die restlichen Maschinen dienen als Task-Manager, auf denen die Worksflows ausgeführt werden. Jede Maschine ist mit folgender Hardware ausgestattet:

- Intel Xeon E5-2430 v2 (6 Kerne mit 2,5GHz)
- 48 GByte Arbeitsspeicher (davon werden 40GB Flink zur Verfügung gestellt)
- 1 GBit/s Ethernet-Anschluss

Alle Maschinen nutzen openSUSE-Betriebssystem. Wie bereits erwähnt wurde für Apache Flink die Version 1.12.1 verwendet.

Da jede Maschine 6 Kerne nutzt, wurden jedem Task-Manager 6 Task Slots zugeteilt, sodass die maximale Parallelität insgesamt 96 beträgt. Im Folgenden ist dementsprechend bei der Nutzung des Wortes *Parallelität* die maximale belegte Anzahl an Task Slots gemeint.

Der für die Evaluation verwendete Datensatz basiert auf Samplings von Citi Bike [51], einer Bike-Sharing-Plattform aus New York City. Die Knoten des Graphen repräsentieren hier Fahrrad-Stationen. Die Fahrten zwischen den Stationen sind die Kanten des Graphen. Datensätze wie Citi Bike eignen sich für Analysen mit GRAFS, da in der Praxis ständig neue Fahrten enstehen und diese mit GRAFS in Near-Realtime analysiert werden können. Der Datensatz liegt in verschiedenen Skalierungsgrößen als CSV vor. Für die Evaluation wurde der 100er Datensatz genutzt, welcher

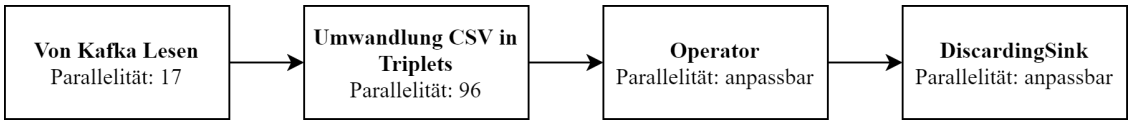


Abbildung 6.1.: Übersicht des generellen Evaluationsdesigns für Durchsatz-Messungen.

rund 10 Millionen Kanten aufweist. Hierfür wurden auf einem der Master sowie allen Workern jeweils ein Kafka-Broker aufgesetzt, um möglichst hohe Leseraten zu gewährleisten. Der Datensatz wurde anschließend in einem CSV-Format auf diese 17 Kafka-Broker verteilt (17 Partitionen, keine Replikation), wobei jede Zeile des Datensatzes genau eine Kante (und die Informationen) und die dazugehörigen Knoten(-informationen) hält. Damit diese Daten während des Benchmarkings im Kafka-Broker erhalten bleiben, wurde die Speicherperiode entsprechend hoch gewählt.

Evaluationsdesign für Durchsatz Das folgende Design wurde für alle Durchsatz-Messungen genutzt. Abbildung 6.1 zeigt eine Übersicht über diesen Vorgang (eventuelle Optimierungen von Flink sind dort nicht berücksichtigt). Zunächst werden die Daten unter Nutzung von Flinks `KafkaConsumer` von den Kafka-Brokern mit einer (Eingabe-)Parallelität von 17 gelesen, um die höchste Leserate zu gewährleisten. Anschließend werden die so gelesenen CSV-Daten mit maximaler Parallelität in Triplets umgewandelt. Danach werden die Triplets an den Operator weitergeleitet und dort verarbeitet, wobei die Parallelität des Operators angepasst werden kann. Schließlich werden die so verarbeiteten Triplets an eine sogenannte `DiscardingSink` weitergeleitet. Hier werden die Triplets verworfen, was im Gegensatz zu anderen Arten von Datensenken den Overhead durch eventuelle Schreibvorgänge (bspw. auf eine Festplatte oder an einen Kafka Broker) minimiert. Auch die Parallelität der Senke kann angepasst werden.

Sollte bei der Evaluation das System so sehr ausgelastet sein, dass Backpressure entsteht, so wird diese bis zur Quelle propagiert, sodass die `KafkaConsumer` ihre Lesegeschwindigkeit reduzieren.

Metriken Für die Evaluation von Datenstromverarbeitungssystemen werden mehrere Metriken benutzt. Für die Nicht-Windowed Operatoren sind *Durchsatz* und *Latenz*, sowie der *Speedup* die gemessenen Metriken. Da der Durchsatz nicht direkt messbar ist, wird hierfür die Laufzeit erfasst und mit dieser und der Anzahl der durch den Operator verarbeiteten Elemente der Durchsatz berechnet.

Definition 27 (vgl. [59])**.** *Der **Durchsatz** eines Operators ist das Verhältnis der Anzahl der durch den Operator verarbeiteten Elemente zur Laufzeit der Operatorverarbeitung:*

$$Durchsatz = \frac{Anzahl\ der\ durch\ den\ Operator\ verarbeiteten\ Elemente}{Gesamtlaufzeit\ der\ Operatorverarbeitung}$$

Im Gegensatz zum Durchsatz wird die Latenz direkt gemessen.

Definition 28 (vgl. [59])**.** *Die **Latenz** eines Operators beschreibt die durchschnittliche Zeit, die der Operator für die Verarbeitung eines Elements benötigt.*

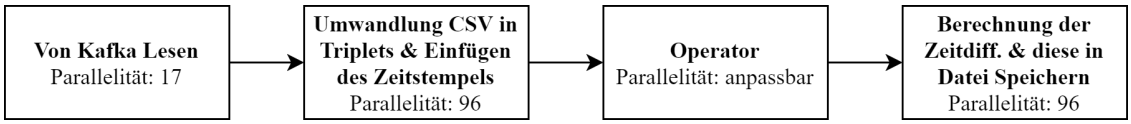


Abbildung 6.2.: Übersicht des Evaluationsdesign für Latenz-Messungen.

Der Speedup ist ein Maß zur Evaluierung der Performance-Steigerung bei Erhöhung der Prozessoren innerhalb der Parallelverarbeitung

Definition 29 (vgl. [60, 61])**.** *Der* ***Speedup*** S_p *ist das Verhältnis von Antwortzeiten bei Erhöhung der Parallelität um den Faktor n:*

$$S_p = \frac{\textit{Gesamtlaufzeit der Operatorverarbeitung unter Nutzung von } p * b \textit{ Prozessoren}}{\textit{Gesamtlaufzeit der Operatorverarbeitung unter Nutzung von } b \textit{ Prozessoren}}$$

Mit Prozessoren sind hier im speziellen Prozessor-Kerne gemeint. Die Basis b der Tests ist dabei ein Prozessor, also 6 Kerne.

Darüber hinaus wird für die Windowed Operatoren auch die Laufzeit der Operatoren über ein Window gemessen und als Metrik verwendet.

6.1. Nicht-Windowed Operatoren

Für alle Nicht-Windowed Operatoren wurden einfache UDFs definiert, deren Rechenkomplexität für ein Element konstant ist. Die genauen Definitionen der UDFs befinden sich als Pseudocode in Code 12 im Anhang.

Latenzmessungen Für die Latenz wurde zunächst das oben erwähnte Evaluationsdesign des Durchsatzes als Vorlage genommen. Zusätzlich wurde direkt vor der Ausführung des Operators in den Properties der Kante ein Zeitstempel für die aktuelle Zeit gesetzt. Dieser wird direkt nach der Ausführung des Operators ausgelesen und eine Zeitdifferenz in Millisekunden gebildet, die für die Auswertung gespeichert wird (vgl. Abbildung 6.2).

Tabelle 6.1 zeigt eine Übersicht der Ergebnisse. Alle Operatoren weisen vergleichbare Ergebnisse auf. Bei allen Tests unter der maximalen Parallelität wurde von Flink eine Backpressure gemessen. Dies wird auch aus den Latenzen deutlich: Während die Latenz des Operators bei maximaler Parallelität häufig unter einer Millisekunde liegt (durchschnittlich in 88% der Fälle), liegt die durchschnittliche Latenz für andere Parallelitäten zwischen 982-1224 Millisekunden. Dabei scheint es in der Statistik keinen deutlichen Unterschiede zwischen diesen Parallelitäten zu geben. Lediglich eine genauere Untersuchung der Latenz-Verteilung zeigt diese auf.
Abbildung 6.3 zeigt die relative Verteilung der Latenz innerhalb der Messung auf. Hierfür wurde repräsentativ Kanten-Transformation gewählt, alle anderen Operatoren zeigen vergleichbare Ergebnisse. In der Abbildung wird deutlich, dass kleinere Parallelitäten einen höheren Anteil an kleineren Latenzwerten haben, aber auch größeren Schwankungen unterliegen, wie man z. B. im Bereich von 3.600-3800 Millisekunden sieht, da bei der Parallelität von 6 ca. 8% der Triplets eine solche Latenz

		kein Op.	Kanten-induzierter Subgraph	Knoten-induzierter Subgraph	Kanten-Transformation	Knoten-Transformation
P=6	Ø		982	1064	1168	998
	Max		8360	6140	9325	6669
	Quantile (90;95;99)		(3386;3630;4395)	(3459;3650;3869)	(3758;4504;5213)	(3507;3622;3907)
P=12	Ø		1022	1058	916	922
	Max		8032	8388	10702	9943
	Quantile (90;95;99)		(2761;3128;4610)	(2918;3623;4765)	(2444;3714;4865)	(2720;3640;4560)
P=24	Ø		942	943	1064	1036
	Max		11251	9601	7996	8772
	Quantile (90;95;99)		(2463;3243;4650)	(2734;3535;4678)	(2617;3155;4515)	(2578;3395;4842)
P=48	Ø		1072	1179	1224	1103
	Max		9131	11396	10147	9815
	Quantile (90;95;99)		(3153;3980;4888)	(3705;4701;6068)	(3128;3804;4897)	(3166;4597;6003)
P=96	Ø	0	0	0	0	0
	Max	159	50	129	20	40
	Quantile (88;95;99)	(0;1;1)	(0;1;1)	(0;1;1)	(0;1;1)	(0;1;1)

Tabelle 6.1.: Latenz-Statistik mit Durchschnitt, Maximum und Quantile (90%,95%,99%) in Millisekunden für Nicht-Window-Operatoren. Das Minimum betrug bei allen Messungen zwischen 0-2ms.

aufweisen. An den Ergebnissen lässt sich vermutlich nur das Verhalten von Flink bei auftretender Backpressure ablesen.

Die Task-Optimierungen von Flink bei der Parallelität von 96 spielen in den Ergebnissen vermutlich auch eine Rolle, da Flink hier die Umwandlung von CSV in Triplets und den Operator-Subtask zu einem Task zusammenfasst, sodass nach der Umwandlung keine Weitersendung an den Operator-Task nötig ist, was die Netzwerklatenz verringert. Dies lässt sich jedoch nicht klar aus den Ergebnissen ablesen.

Durchsatzmessungen Es wurden zwei verschiedene Durchsatz-Messungen vorgenommen. Zunächst wurde lediglich die Parallelität der Operatoren angepasst und die Senkenparallelität auf den Maximalwert (96) gesetzt. Durch diesen Test lässt sich die Senke, oder der Weg zur Senke, als Bottleneck ausschließen, allerdings entstehen zusätzliche Kommunikationskosten, da die Triplets nach der Operation neuverteilt werden, solange die Parallelität von Operatoren und Senke nicht gleich ist.
Im zweiten Teil wurde die Optimierung von Flink ausgenutzt und die Senkenparallelität mit der Operatorparallelität gleichgesetzt. Dadurch werden beide zu einem Task zusammengefasst und die Ausführung findet lediglich auf einem Task-Manager statt. Hier entsteht das umgekehrte Problem, dass die zusätzlichen Kommunikationskosten wegfallen, die Senke aber als Bottleneck fungieren kann.
In beiden Messungen wurde zusätzlich eine weitere Messung ohne Operator durchgeführt, d. h. direkt nach der Umwandlung der CSV-Daten in Triplets werden diese zur Senke geführt. Dies erlaubt es, ein ungefähres Durchsatzmaximum des Systems zu berechnen, wenn alle Elemente zur Senke geleitet werden.

Abbildung 6.4 zeigt die Messergebnisse sowie den Speedup der Operatoren (die genauen Werte können in Tabelle A.1 im Anhang gefunden werden). Das berechnete Durchsatzmaximum (ohne

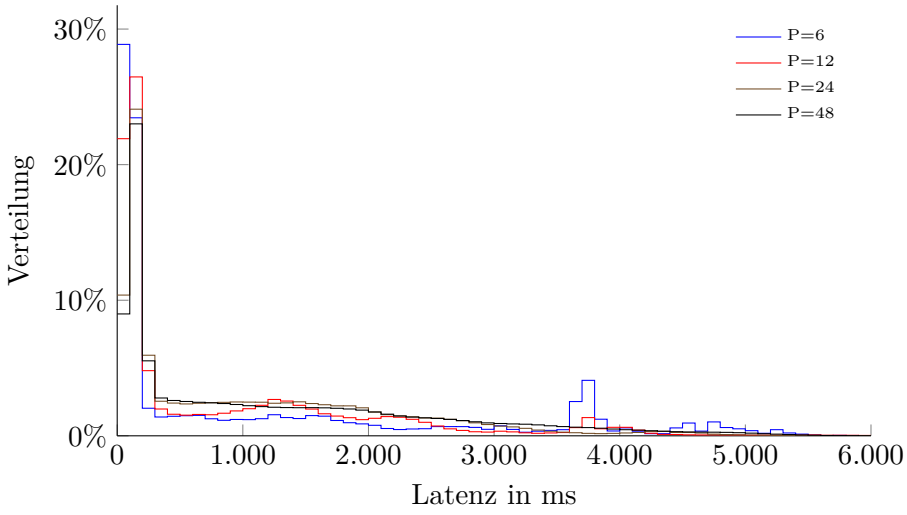
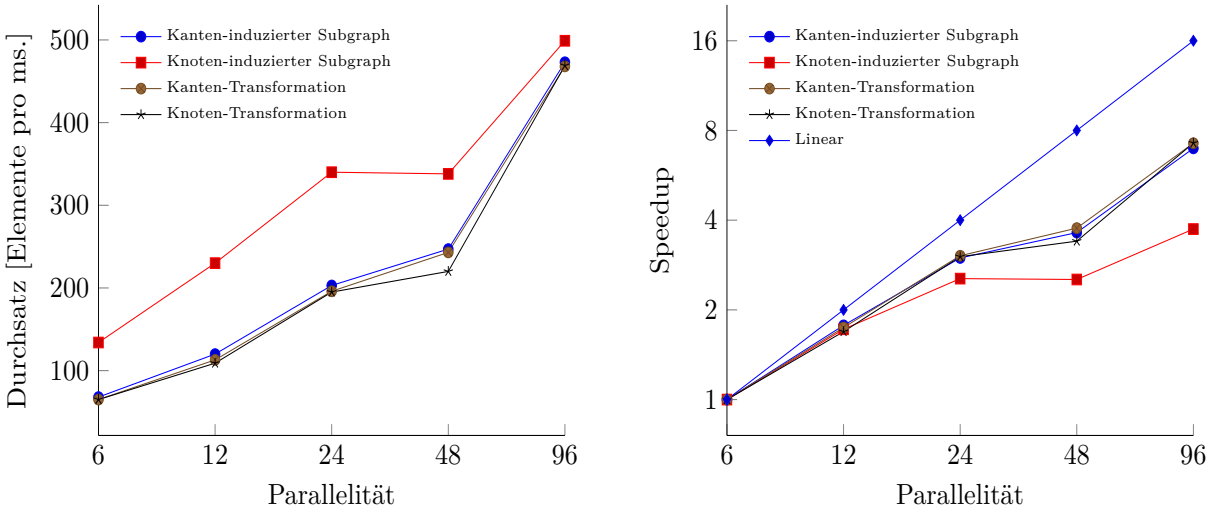
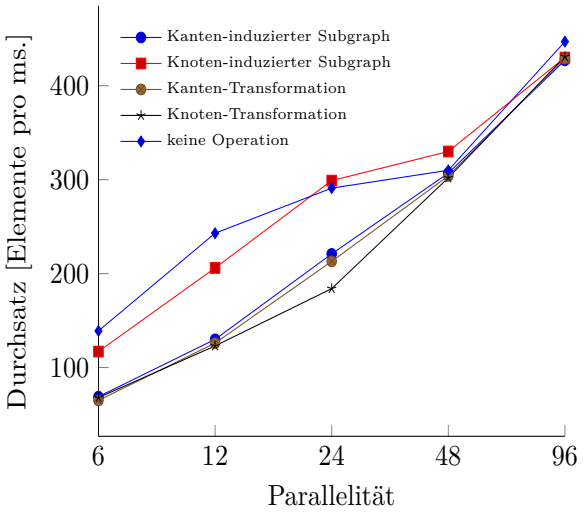


Abbildung 6.3.: Histogramm der Latenz von Kanten-Transformation, unterteilt in 100ms-Schritten.

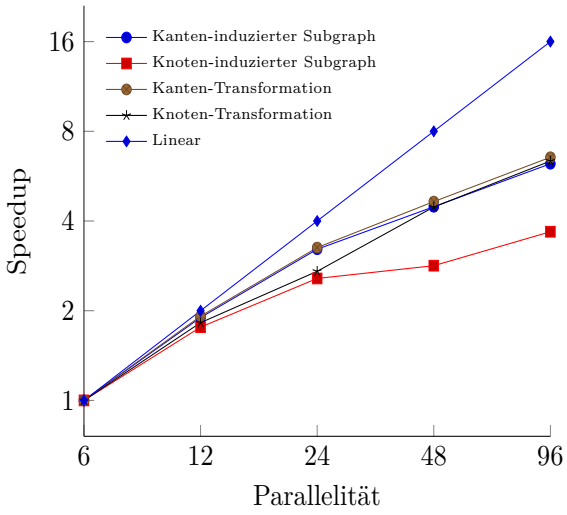


(a) Durchsatz bei maximaler Senkenparallelität.

(b) Speedup bei maximaler Senkenparallelität.



(c) Durchsatz bei skalierter Senkenparallelität.



(d) Speedup bei skalierter Senkenparallelität.

Abbildung 6.4.: Ergebnisse der Nicht-Window-Operatoren bei maximaler Senkenparallelität.

Operator) bei maximaler Senkenparallelität liegt bei ungefähr 447 Triplets pro Millisekunde. Dieses ist vermutlich durch das Kafka-Cluster und die Eingabeparallelität des Clusters begrenzt.
Aufgrund der Latenzmessungen war zu erwarten, dass alle Operatoren zunächst einen ähnlichen Durchsatz aufweisen. Da die Subgraph-Operatoren Triplets filtern und diese dementsprechend nicht mehr zur Senke gesendet und dort verarbeitet werden müssen, weisen diese einen höheren Durchsatz auf. Die gewählten UDFs der Subgraph-Operatoren weisen für den Knoten-induzierten Subgraph 0% Selektivität und für den Kanten-induzierten Subgraph ca. 95,4% Selektivität auf.

Die Transformationsoperatoren und der Kanten-induzierte Subgraph-Operator verhalten sich ähnlich, was nach den Ergebnissen der Latenzmessung zu erwarten ist, da die Verarbeitungskosten relativ gering zu den Kommunikationskosten sind und diese durch die hohe Selektivität beim Kanten-induzierten Subgraph immer noch einen hohen Anteil der Kosten ausmachen. Alle Operatoren erreichen bei den Messungen mit maximaler Senkenparallelität am Ende ein leicht höheres Maximum als das bei der Messung ohne Operator, dies ist aber auf die Varianz der Messungen zurückzuführen.

Beim Vergleich der beiden Messvarianten wird bei P=48 deutlich, dass es effizienter sein kann die Senke mitskalieren zu lassen, da das Zusammenfassen des Operator-Task und des Senken-Tasks hier zu höheren Durchsatzwerten führt. Weiterhin erreichen die Operatoren bei der skalierten Variante das vorgegebene Durchsatz-Maximum der Variante *kein Operator* (vgl. Abb 6.4c) bereits bei P=48. Wie zu erwarten, weist der Knoten-induzierter Subgraph-Operator insgesamt deutlich höhere Durchsätze auf, da die Triplet-Elemente verworfen werden.

Bei Betrachtung der Skalierbarkeit bei maximaler Senkenparallelität (Abbildung 6.4b) wird ebenso ersichtlich, dass die Operatoren vom Zusammenfassen der Tasks profitieren, da der Speedup bei P=96 deutlich ansteigt. Hieraus wird deutlich, dass die Kommunikationskosten einen hohen Anteil ausmachen. Warum auch der Knoten-induzierter Subgraph davon betroffen ist, in welchem keine Triplets zur Senke gesendet werden müssen, ist unklar.

Beide Messansätze (Abb. 6.4b und 6.4d) können den initialen Speedup bei höheren Skalierungen nicht beibehalten. Dass in höheren Skalierungen keine höheren Speedups erreicht werden, ist auch dem allgemeinen Durchsatzmaximum verschuldet, welches, wie bereits erwähnt, alle Operatoren erreichen. Quellen mit höherem Durchsatz und eine höhere Eingabeparallelität könnten durchaus bessere Speedups erreichen.

6.2. Windowed Operatoren und Analyse-Pipeline

Im folgenden Abschnitt werden zunächst die Durchsatzmessungen der einzelnen Windowed Operatoren sowie der bereits vorgestellten Analyse-Pipeline dargestellt. Im Anschluss wird ein Verfahren für die Messung der Processing Time eines Windows fester Größe vorgestellt und die Ergebnisse präsentiert. In beiden Messungen werden zudem die beiden Grouping-Implementierungen untereinander sowie das verteilte Grouping mit verschiedenen Parallelitäten verglichen.

Da bei ersten Messungen deutlich wurde, dass der im Pattern Matching genutzte `windowAll`-Ansatz einen Bottleneck bildet, da in diesem alle Elemente eines Windows in einem Task verarbeitet werden

(P=1), wurde bei den Messungen darauf verzichtet die Vorverarbeitung in Dual Simulation mit anderen Parallelitäten als dem Maximum (P=96) zu messen. Das zentrale Grouping nutzt ebenso den `windowAll`-Ansatz, sodass der Operator insgesamt nur mit P=1 arbeitet.

Durchsatzmessungen Für die Windowed Operatoren wurde das bereits vorgestellte Evaluationsverfahren für den Durchsatz genutzt. Die für die Messung genutzten Parameter befinden sich als Pseudocode im Anhang in Code 13.
Da die Windowed Operatoren Elemente sammeln und die Processing Time eines Windows auch von der Anzahl der Elemente im Window abhängig ist, spielt nicht nur die von der Quelle ausgehende Durchsatzgeschwindigkeit eine Rolle, sondern auch die gewählte (Time-)Window Size. Die Messungen wurden auf einem Tumbling Window mit Processing Time ausgeführt. Letzteres wurde gewählt, da der vorhandene Datensatz nicht sortiert ist und dementsprechend die in den Kanten vorhandene Zeit nicht als Event Time genutzt werden kann, da Flink bei der Nutzung von Event Time annimmt, dass die Elemente in Reihenfolge ihrer Event Time eingehen oder zumindest eine angegebene *Allowed Lateness* nicht überschreiten. Die Ausgabe für das verteilte Grouping sind dadurch zwar unbrauchbar (vgl. die in Unterunterabschnitt 5.2.1.3 angesprochenen Einschränkungen des verteilten Verfahrens), sollten trotzdem einen Einblick in die zu erwartenden Durchsatzgeschwindigkeiten geben.

Abbildung 6.6c zeigt den Durchsatz der Operatoren mit maximaler Parallelität über verschiedene Window Sizes. Keiner der Operatoren erreicht dabei den im vorherigen Abschnitt gemessenen Maximaldurchsatz (447 Triplets pro Millisekunde).
Zudem scheint es, bis auf für das Verteilte Grouping, keine Relation zwischen dem Durchsatz und der Window Size zu geben. Dual Simulation und Zentrales Grouping behalten ihren Durchsatz annähernd über alle Window Sizes. Auch das Verteilte Grouping und die *Citi Bike*-Pipeline bleiben in ihrem Durchsatz nahezu konstant, bis dieser bei einer Größe von 5000ms stärker einbricht. Der nahezu konstante Durchsatz kann zum Teil durch die Rückkopplung der Backpressure erklärt werden: Alle Operatoren erzeugen bei der Ausführung Backpressure, sodass der Eingangsdurchsatz sinkt. Da die Window Size in Millisekunden angegeben ist und diese auf der Processing Time beruht, werden durch die Verlangsamung weniger Triplets pro Window gesammelt. Es lässt sich leider nicht nachvollziehen, wie viele Elemente pro Window sich in den Operatoren (durchschnittlich) befinden.

Trotzdem ist ersichtlich, dass das Verteilte Grouping bei maximaler Parallelität deutlich höhere Durchsatzwerte erreicht als das Zentrale Grouping (je nach Window Size 5-9x so schnell). Die *Citi Bike*-Pipeline ist erwartbar langsamer als das Verteilte Grouping (erreicht 60-65% des Durchsatzes), verhält sich aber bei verschiedenen Window Sizes ähnlich.
Es sei erwähnt, dass Flink bei der Optimierung der *Citi Bike*-Pipeline die Kanten-Aggregationsphase des Verteilten Groupings mit dem Subgraph-Operator zu einem Task zusammenfasst. Dies kann dazu führen, dass eine Last-Inbalance entsteht, da nicht angenommen werden kann, dass das Kanten-Grouping gleichgroße Gruppen erzeugt. Dementsprechend könnte ein Worker nicht nur mehr Verarbeitungszeit benötigen, wenn die ihm zugeordnete Gruppe größer ist als anderen Task-Slots, er müsste anschließend auch den Subgraph-Operator auf all diese Elemente anwenden. Die Effekte ei-

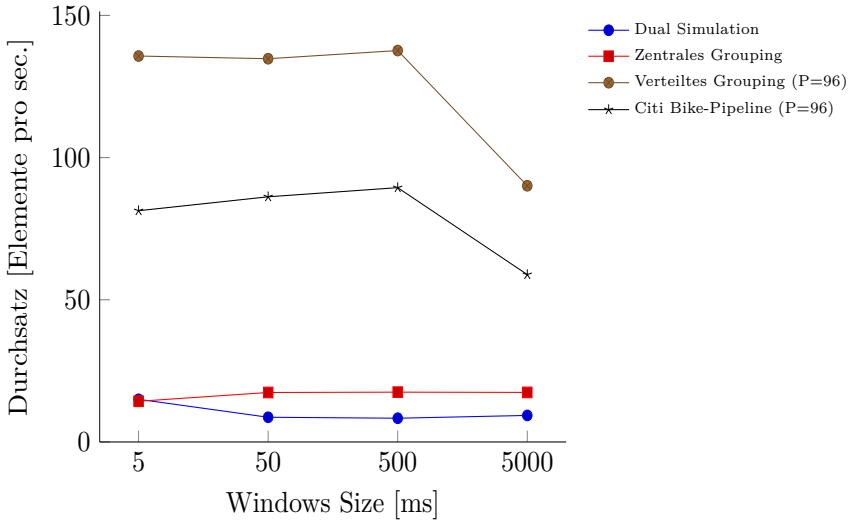
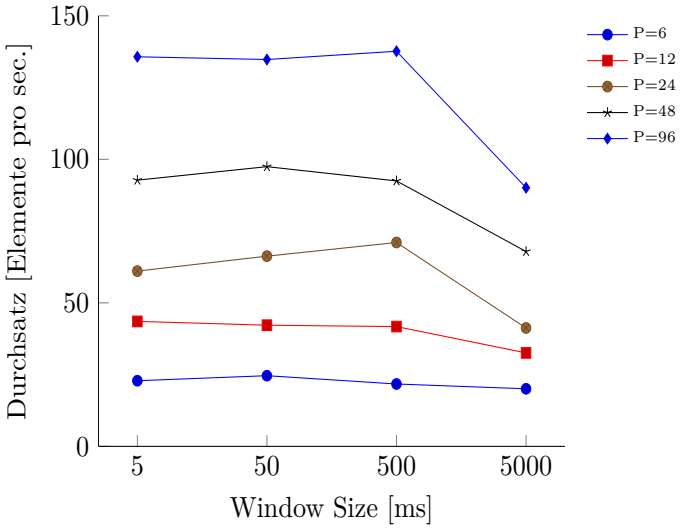


Abbildung 6.5.: Durchsatz der Window-Operatoren.

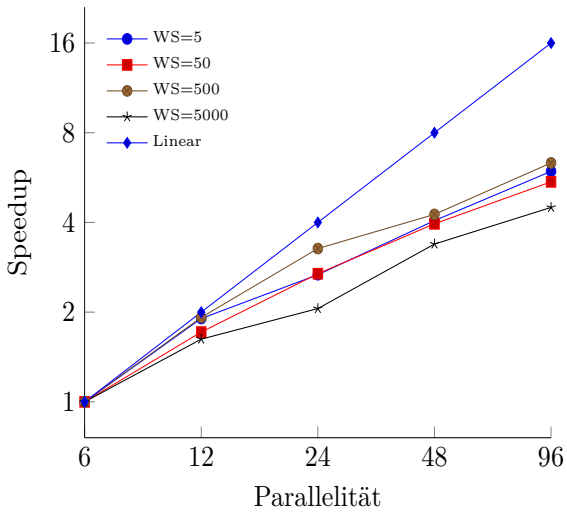
ner forcierten Lastbalancierung, bspw. durch die Nutzung von Flinks `reshuffle`-Methode, bleiben zu untersuchen.

Abbildung 6.6 zeigt den Durchsatz und den Speedup von Verteiltem Grouping und der *Citi Bike*-Pipeline für verschiedene Parallelitäten. Die Durchsatz-Grafiken (Abb. 6.6a und 6.6c) zeigen, dass das Durchsatzverhältnis der Pipeline zu dem Verteiltem Grouping bei niedrigeren Parallelitäten steigt (von ca. 64% auf 80%). Dies kann unter anderem dadurch erklärt werden, dass niedrigere Parallelitäten zu längerer Rechenzeit für die Operatoren führen, sodass der Overhead durch Netzwerkkommunikation sinkt. Zudem findet diese in niedrigen Parallelitäten nicht mehr so oft statt, da dadurch die Wahrscheinlichkeit steigt, dass für einen darauffolgenden Task derselbe Task-Manager verantwortlich ist.
Vergleicht man den Speedup der Windowed Operatoren (Abb. 6.6b und 6.6d), sieht man noch deutlicher als bei den Nicht-Windowed Operatoren, dass der Kommunikationsoverhead bei höheren Parallelitäten zu schlechteren Skalierungen führt: Beide starten mit einer linearen Skalierung bei der ersten Verdopplung der Parallelität, flachen danach stärker ab, sodass die Skalierung des Verteilten Groupings bei der 16fachen Task-Slots-Anzahl nur bei durchschnittlich 5,6 liegt und die der Pipeline bei 4,4. Hierbei sei nochmal erwähnt, dass die Nicht-Windowed Operatoren, mit Ausnahme des Knoten-induzierten Subgraph-Operators, trotz des Erreichens des Maximaldurchsatzes im Schnitt eine Skalierung von 7,2 erreichen.
Es bleibt zu untersuchen, wie die Skalierung bei einer Verarbeitung mit Event Time aussehen würde, in der die durchschnittliche Window Size nicht von dem Eingangsdurchsatz abhängig ist.

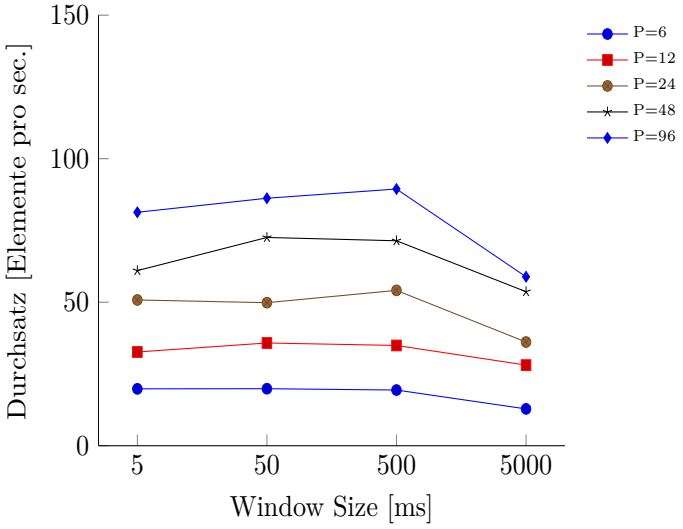
Laufzeit bei fester Window Size Da die vorherigen Ergebnisse keine Erkenntnisse über das Verhalten der Windowed Operatoren in Relation zur Anzahl der sich im Time Window befindlichen Triplet-Elemente bietet, wird ein zweiter Test konzipiert, der einen besseren Aufschluss über dieses Verhalten gibt. Ziel ist es die Laufzeit **einer** Window-Berechnung für alle Windowed Operatoren bei unterschiedlicher Anzahl an Elementen in dem Window zu messen. Im Folgenden ist, im Gegensatz zum vorherigen Abschnitt, mit der *Window Size* die Anzahl der Elemente im Window gemeint. Andere Window-Arten (bspw. *Time Window*) werden entsprechend benannt.



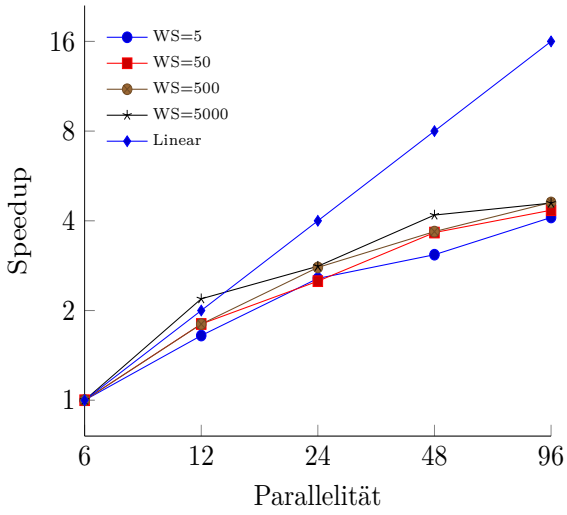
(a) Durchsatz von verteiltem Grouping.



(b) Speedup von verteiltem Grouping.



(c) Durchsatz der *Citi Bike*-Pipeline.



(d) Speedup der *Citi Bike*-Pipeline.

Abbildung 6.6.: Vergleich von Durchsatz und Speedup von Verteiltem Grouping und der *Citi Bike*-Pipeline über verschiedene Parallelitäten (FG=Fenstergröße).

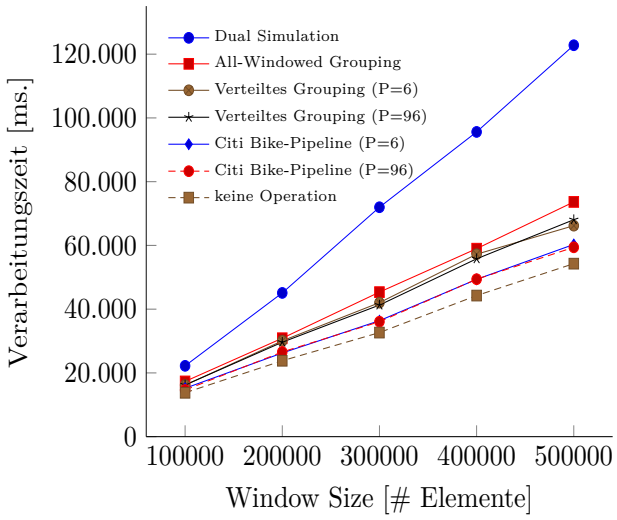


Abbildung 6.7.: Laufzeit der Window-Operatoren bei Verarbeitung eines Windows mit fester Anzahl an Elementen.

Für die Messung wird zunächst von Kafka eine feste Anzahl an Elementen gelesen, in Triplet-Elemente umgewandelt und anschließend in einer Liste zwischengespeichert. Anschließend wird mit Hilfe der `StreamExecutionEnvironment.fromCollection`-Methode aus der Liste ein `DataStream` erzeugt. Dies stellt die Window Size dar. Daraus wird das `GraphStream`-Objekt erzeugt, wobei alle Triplets denselben Event Time-Zeitstempel aufweisen. Anschließend wird die Streamverarbeitung mit dem jeweiligen Operator ausgeführt, wobei ähnlich zum vorherigen Test ein Tumbling Window genutzt wird, diesmal als Zeitstempel jedoch die Event Time genutzt wird. Die dabei festgesetzte Größe des Time Windows ist dabei zu vernachlässigen, da alle Elemente denselben Zeitstempel aufweisen. Dies führt dazu, dass alle Elemente des Streams demselben Window zugewiesen werden. Das Window wird automatisch geschlossen und die Verarbeitung begonnen, wenn das letzte Element des Streams im Window ankommt.
Die für die Streamverarbeitung benötigte Zeit wurde gemessen. Die Messung wurde viermal mit je unterschiedlichen Elementen (aber gleicher Anzahl) durchgeführt und die Ergebnisse wurden gemittelt.

Abbildung 6.7 zeigt das Laufzeit-Verhalten der Operatoren bei verschiedenen Window Sizes. 500.000 Elemente bildet dabei die maximal getestete Window Size, da aufgrund des Messdesigns eine höhere Anzahl an Elementen zu Timeouts bei der Initialisierung führt. Es wurde zusätzlich die minimal benötigte Zeit gemessen, wenn kein Operator angewendet wird. Eine Tabelle mit allen Ergebnissen kann wieder im Anhang (Tabelle A.3) gefunden werden.
Zunächst sieht man, wie erwartet, einen Anstieg der Zeit bei höheren Window Sizes. Dabei verhalten sich alle Operatoren mit Ausnahme von Dual Simulation ähnlich zu der Verarbeitung ohne Operator und verarbeiten bei höheren Window Sizes durchschnittlich 15% mehr Elemente. Der genaue Grund hierfür ist unklar, eine Erklärung wären eventuelle Fixkosten, die bspw. bei dem Aufsetzen der Flink-Umgebung enstehen, da die Verarbeitungszeit für diese Tests lediglich ca. 14 Sekunden (bei 100.000 Elementen) bis 54 Sekunden (bei 500.000 Elementen) beträgt.

Dementsprechend scheint es plausibel, dass die Berechnungszeit für die beiden Grouping-Operatoren sowie die *Citi Bike*-Pipeline annähernd linear zu der Anzahl der Elemente ist.

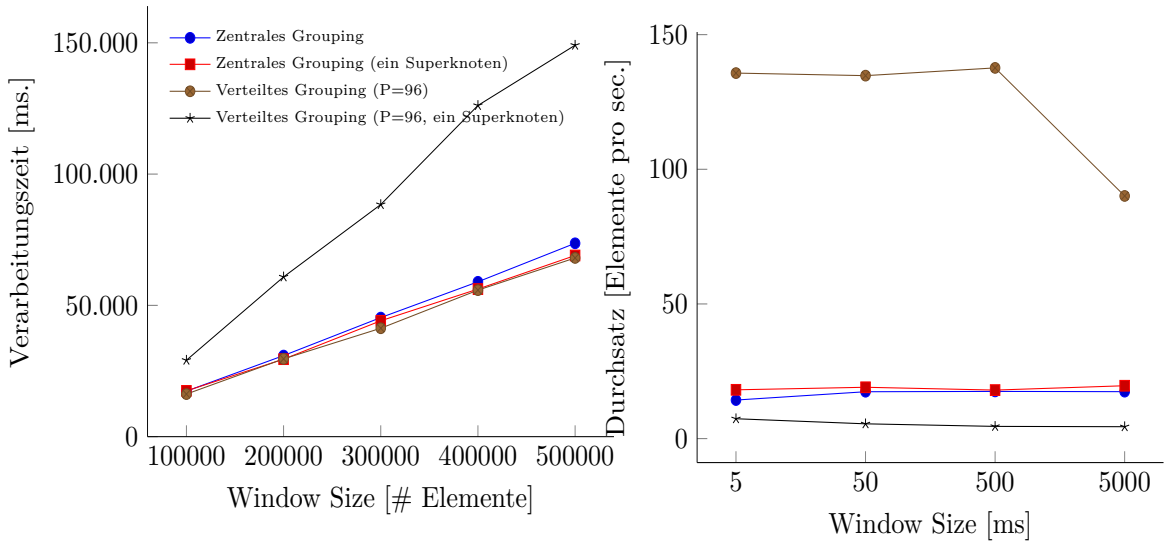
Wie in den vorherigen Tests schneiden das Verteilte Grouping und die *Citi Bike*-Pipeline besser ab. Dabei scheint die Parallelität keine signifikante Auswirkung zu haben (im Diagramm sind nur die Parallelitäten P=6 und P=96 gelistet, alle anderen weisen ähnliche Werte auf). Hier sind die Performancewerte des Zentrale Groupings im Verhältnis zum Verteilten Grouping deutlich besser als bei den vorherigen Messungen. Dies weist insgesamt darauf hin, dass durch die Nutzung von Flinks `windowAll`-Transformation in der Implementierung des Zentralen Grouping-Operators Flink genau ein Task-Slot nutzt, der zunächst wieder frei werden muss, bevor das nächste Window berechnet werden kann. Dementsprechend sind die Performancewerte für das Zentrale Grouping gut, wenn das System (wie bei diesem Test) vor Beginn der nächsten Window-Verarbeitung noch nicht ausgelastet ist. Sie brechen aber stark ein, falls das nächste Window schneller geschlossen wird als der Worker Elemente verarbeiten kann, da die Elemente bereit für die Verarbeitung wären, der Worker aber noch mit der Berechnung des vorherigen Windows beschäftigt ist.
Das Verteilte Grouping profitiert hingegen bei einem Stream, in dem mehrere Windows gebildet werden, anscheinend von den höheren Parallelitäten, da dadurch mehr Task-Slots zur Verfügung stehen und so eine bessere Verteilung der Daten möglich ist.

Der Dual Simulation-Operator weist insgesamt eine steigende Verarbeitungszeit pro Triplet-Element bei höherer Anzahl an Elementen im Window auf. Dies ist erwartbar, da Dual Simulation-Berechnungen eine höhere Komplexität aufweisen als die des Groupings.

Untersuchung vom Extremfall in den Grouping-Operatoren Neben den obigen Untersuchungen wurde auch untersucht, wie sich die Grouping-Operatoren bei schlechter Verteilung der Knoten verhalten. Hierfür wurde für die Knoten kein Grouping Key angegeben, sodass alle Knoten eines Windows zu einem Superknoten zusammengefasst werden. Dies führt dazu, dass beim Verteiltem Grouping alle Knoten innerhalb eines Task-Slots auf einem Worker gesammelt werden und dort verarbeitet werden müssen. Abbildung 6.8a zeigt zunächst, dass sich in diesem Fall die Verarbeitungszeit des Verteiltem Groupings für ein Window fast verdoppelt, während das Zentrale Grouping von der Wahl der Grouping Keys unbeeinflusst zu sein scheint.

Die stärkeren Auswirkungen sieht man, wenn man die Messungen für den gesamten *Citi Bike*-Stream mit mehreren Windows betrachtet (Abbildung 6.8b). Hier sinkt der Durchsatz von Verteiltem Grouping auf 3-5%. Der Grund dafür ist vermutlich, dass die Triplet-Elemente mit demselben `KeySelector`-Wert immer an dieselben Task-Slots zugeteilt werden und die Window-Verarbeitung für diese Elemente dort stattfindet. Da in diesem Fall alle Knoten denselben Wert aufweisen, werden sie somit alle zu demselben Task-Slot auf demselben Worker gesendet, der alle Berechnungen ausführen muss. Dadurch, dass das Verteilte Grouping zwei Knotenaggregationsphasen hat, enstehen für diesen Fall zwei Bottlenecks.

Die Ergebnisse zeigen, dass Verteiltes Grouping nicht immer besser ist als das Zentrale Grouping. Auch das Zusammenfassen aller Kanten in einem Window zu einer Superkante führt damit zu einem theoretischen Bottleneck. Je nach Wahl der Grouping Keys kann dementsprechend die Nutzung des Zentralen Groupings empfohlen werden.



(a) Vergleich bei einem Window mit fester Anzahl an Elementen.

(b) Vergleich über Stream mit verschiedenen Time Window Sizes.

Abbildung 6.8.: Vergleich der beiden Grouping-Operatoren mit versch. Grouping-Keys.

Verzicht auf Latenzmessungen Bei den Messungen für die Windowed Operatoren wurde auf die Latenzmessung verzichtet, da diese auch von der Time Window Size abhängig sind. Ist das Time Window beispielsweise 1 Sekunde groß, so erhalten die ersten Elemente des Windows eine Latenz von mindestens einer Sekunde, da sie zunächst darauf warten müssen, dass das Window geschlossen wird, bevor die eigentlich Berechnung beginnt. Die in [59, Definition 3] vorgeschlagene Vorgehensweise, dass der Zeitstempel eines Windows durch das Maximum der Zeitstempel der im Window befindlichen Elemente bestimmt wird, hätte auf der Operator-Ebene implementiert werden müssen.
Die aus dem zweiten Messvorgang entstandenen Verarbeitungszeiten liefern zudem einen Einblick in die zu erwartenden Latenzen, wenn das System nicht ausgelastet ist.

6.3. Fazit

Die Messungen zeigen, dass im Speziellen die Nicht-Windowed Operatoren für die Analyse von PGS geeignet sind und die Erwartungen übertreffen, sodass in Zukunft weitere Performance-Messungen mit höherer Eingabeparallelität sinnvoll sind.
Wie erwartet weisen die Windowed Operatoren schlechtere Durchsatzergebnisse auf. Dabei schneidet das Verteilte Grouping in den meisten Fällen besser ab als das Zentrale Grouping, auch wenn dies mit Einschränkungen der Window-Arten einhergeht. Die Performance von Dual Simulation war, wie durch die Ergebnisse von [48] erwartet, im Vergleich zu den anderen Operatoren deutlich langsamer. Es bleibt zu untersuchen wie sich die Windowed Operatoren unter Nutzung der Event Time verhalten und ob die im Vergleich zu den Nicht-Windowed Operatoren geringen Durchsatzwerte für die Praxis problematisch sind.
Schließlich sei die hier gemessene Citi Bike-Pipeline erwähnt, deren Ergebnisse zeigen, dass eine Analyse-Pipeline mit dem hier implementierten System parallelisierbar ist.

7. Zusammenfassung und Ausblick

In dieser Masterarbeit wurde erfolgreich ein Modell entwickelt, welches Property-Graphen mit Streaming vereint und zusätzlich eine einfache Form der Graph Collections unterstützt. Darauf aufbauend wurden die Operatoren-Modelle Subgraph, Transformation und der Hilfsoperator Reduce-Combination, sowie Grouping und Pattern Matching umgesetzt. Die letzteren beiden Operatoren nutzen dabei Windowing.

Anschließend wurde das Graphstream-Analyse-System `GRAFS` mit Apache Flinks `DataStream`-API auf Basis von Java entwickelt, welche die genannten Modelle implementiert. Für einfache Graph-Operatoren wie Subgraph und Transformation wurden zu Gradoop äquivalente Operatoren für das Modell umgesetzt. Weiterhin wurden zwei Varianten von Grouping und das Pattern Matching auf Basis eines Dual Simulation-Algorithmus umgesetzt. Beide Operatoren nutzen dabei Windows, sodass beide Operatoren in der Theorie auf einem Graphdatensatz mit Hilfe eines Global Windows dieselben Ergebnisse liefern wie beispielsweise Gradoops Implementierung. Damit ist es, soweit bekannt, eines der ersten Systeme, welches mit Graphstreams umgehen kann, in dem die Graphelemente zusätzliche Eigenschaftsinformationen beinhalten.
Bei der Umsetzung des Graphstream- und Operatoren-Modells wurde zum einen auf die Erweiterbarkeit geachtet, sodass im Speziellen die Implementierung neuer Operatoren durch die Nutzung der vordefinierten Interfaces möglich ist. Zum anderen wurde auf die Nutzerfreundlichkeit geachtet, welche unter anderem durch die Bereitstellung von Javas Lambda-Ausdrücken für in den Operatoren genutzte UDFs und der Abstraktion von Flinks Stream-Klassen erreicht wurde.

Die einzelnen Operatoren sowie eine beispielhafte Analyse-Pipeline wurden auf Basis des *Citi Bike*-Datensatzes auf ihre Performance evaluiert. Die Durchsatz-Peformance der Nicht-Windowed Operatoren bei hoher Skalierung scheinen dabei den Output-Durchsatz des auf Apache Kafka basierenden Clusters zu übertreffen. Es wurden jedoch die Grenzen der Nutzung von Windowed Operatoren aufgezeigt, deren Durchsatz je nach Art des Operators und Größe des Windows nur einen Bruchteil des Durchsatzes der Nicht-Windowed Operatoren bieten können. Im Speziellen wurden dort die Grenzen von Zentralem Grouping und Pattern Matching aufgezeigt, deren Implementierung die `windowAll`-Methode von Flink nutzt. Der Durchsatz dieser beiden Operatoren ist deutlich schlechter als der des Verteilten Groupings und sie weisen keine Skalierbarkeit auf. Das Verteilte Grouping zeigt jedoch, dass skalierbare Windowed Operatoren umsetzbar sind.

Ausblick Mit dem Abschluss dieser Arbeit ist der Grundstein für die Analyse von Property-Graphstreams gelegt. Darauf aufbauend bietet es sich an, weitere Forschung zu betreiben.

Es könnte untersucht werden, ob eine Erweiterung des Property-Graphstream-Modells auf die volle Unterstützung des EPGMs sinnvoll ist. Der Fokus in dieser Masterarbeit bei der Entwicklung des neuen Modells war die Operator-API, sodass der Mengen-Support nur schemenhaft übernommen wurde. Außerdem bringt die Erweiterung des Modells auf das EPGMs zusätzliche Kommunikationskosten für jedes Element mit, da, solange ein Triplet weiterhin alle Informationen halten soll, alle Eigenschaften der Graphen, zu denen das Triplet gehört (i. e. alle Graphen, zu denen die Knoten gehören), gespeichert werden müssen.

Im Hinblick auf die Performance könnten einige weitere Untersuchungen getätigt werden.
Zunächst wäre es sinnvoll, die Durchsatz-Grenzen der Nicht-Windowed Operatoren zu untersuchen, was in dieser Arbeit aufgrund der Durchsatz-Begrenzung des Eingabestreams nicht gelungen ist. Zudem scheint, wie bereits erwähnt, eine Evaluation mit Event Time sinnvoll, um mit der Skalierung der Window Size auch die Anzahl der Elemente in einem Window zu skalieren. Weiterhin wäre die Evaluation von komplexere UDFs und anderen Operatoren-Pipelines sinnvoll.

Darüber hinaus könnte im Speziellen für die Windowed Operatoren untersucht werden, für welchen praktischen Einsatz sich diese eignen. Streamverarbeitung ist, im Gegensatz zu Stapeldatenverarbeitung, hauptsächlich für den Live-Betrieb zur Near-Realtime-Analyse gedacht. Gerade im Bezug auf Nutzungsszenarien wie Citi Bike, in welchem die Kanten Fahrten zwischen Stationen darstellen, ist die Erzeugung von mehreren hundert Triplets pro Sekunde unrealistisch. Auf der anderen Seite generieren Nachrichtendienste, wie z. B. Twitter, teils mehrere 1000 Elemente die Sekunde. Dementsprechend wäre eine Anwendungsfall-Analyse sinnvoll, die aufzeigt für welche Art von Graphdaten sich Windowed Operatoren eignen würden, welche Window Sizes nützlich wären und wie viele Elemente sich in einem Window befinden. Aus den daraus entwickelten Erkenntnissen könnten zudem weitere Datensätze generiert werden, die für Benchmarks genutzt werden können, um ein breiteres Performance-Bild des Systems zu erhalten.

Weiterhin kann die Effizienzsteigerung der vorhandenen Windowed Operatoren angestrebt werden. Beispielsweise sind alle drei Aggregationsphasen des Verteilten Groupings auf der `process`-Methode von Flink aufgebaut, sodass zunächst alle Window-Elemente gesammelt werden, bevor die Verarbeitung beginnt. Die Nutzung der `reduce`-Methode wäre für viele Aggregationsfunktionen effizienter, bei der die Elemente aggregiert werden, sobald sie im Window eintreffen. Zusätzlich dazu könnte eine Erweiterung des Verteilten Groupings angestrebt werden, sodass dieses mit mehr Window- und Zeit-Varianten funktioniert. Hierfür müssten im Speziellen Überlegungen bezüglich der Window-Zuordnung in Phase 2 und 3 angestellt werden.

In Kapitel 3 wurden weitere Streaming Frameworks erwähnt, die ähnliche Funktionalitäten wie Flink bieten, namentlich beispielsweise Apache Kafka und Apache Storm. Dementsprechend könnte eine Umsetzung des PGSM auf diesen Frameworks, im Speziellen auf Hinblick der Performance, untersucht werden.

In der Zukunft ist auch die Erweiterung des Systems um andere Operatoren denkbar, die durch die Verwendung der `OperatorI`-Schnittstellen gegeben ist. Insbesondere bietet GRAFS mit Reduce-Combination bisher nur einen Operator zu Anwendung auf Graph Collections.
Darüber hinaus wäre eventuell auch eine Entwicklung von Operatoren unter Nutzung mit approximativen Ergebnissen sinnvoll, beispielsweise unter Nutzung der `IterativeStream`-Klasse. Dabei könnten beispielsweise Operatoren aus [49] umgesetzt werden.

Literatur

[1] *Don't Get Caught Waiting On Fast Data*. Techn. Ber. Forrester Research, 2018.

[2] Martin Junghanns u. a. „Analyzing extended property graphs with Apache Flink". In: *Proceedings of the 1st ACM SIGMOD Workshop on Network Data Analytics*. 2016, S. 1–8.

[3] Salim Jouili und Valentin Vansteenberghe. „An empirical comparison of graph databases". In: *2013 International Conference on Social Computing*. IEEE. 2013, S. 708–715.

[4] Olaf Hartig. „Reconciliation of RDF* and property graphs". In: *arXiv preprint arXiv:1409.3288* (2014).

[5] Alexander Schätzle u. a. „S2X: graph-parallel querying of RDF with GraphX". In: *Biomedical Data Management and Graph Online Querying*. Springer, 2015, S. 155–168.

[6] Inc. Neo4j. *Neo4j Graph Platform – The Leader in Graph Databases*. [Online; abgerufen 17. Februar, 2021]. URL: https://neo4j.com/.

[7] Shanmugavelayutham Muthukrishnan. *Data streams: Algorithms and applications*. Now Publishers Inc, 2005.

[8] Tyler Akidau u. a. „The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing". In: (2015).

[9] J.P. D'Angelo und D.B. West. *Mathematical Thinking: Problem-solving and Proofs*. Featured Titles for Transition to Advanced Mathematics. Prentice Hall, 2000, S. 277. ISBN: 9780130144126. URL: https://books.google.de/books?id=fL6nQgAACAAJ.

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

[11] Apache Software Foundation. *Tumbling Windows*. [Online; abgerufen 23. November, 2020]. URL: https://ci.apache.org/projects/flink/flink-docs-release-1.11/fig/tumbling-windows.svg.

[12] Apache Software Foundation. *Sliding Windows*. [Online; abgerufen 23. November, 2020]. URL: https://ci.apache.org/projects/flink/flink-docs-release-1.11/fig/sliding-windows.svg.

[13] Maciej Besta u. a. „Practice of streaming and dynamic graphs: Concepts, models, systems, and parallelism". In: *arXiv preprint arXiv:1912.12740* (2019).

[14] Andrew McGregor. „Graph stream algorithms: a survey". In: *ACM SIGMOD Record* 43.1 (2014), S. 9–20.

[15] Apache Software Foundation. *Flink Software Stack*. [Online; abgerufen am 02. Februar, 2021]. URL: https://flink.apache.org/img/flink-stack-frontpage.png.

[16] Apache Software Foundation. *Apache Flink 1.11 Documentation: Operators*. [Online; abgerufen 24. November, 2020]. URL: https://ci.apache.org/projects/flink/flink-docs-release-1.11/dev/stream/operators/.

[17] Apache Software Foundation. *Parallel Dataflow*. [Online; abgerufen 23. November, 2020]. URL: https://ci.apache.org/projects/flink/flink-docs-release-1.11/fig/parallel_dataflow.svg.

[18] Apache Software Foundation. *Apache Flink 1.11 Documentation: Windows*. [Online; abgerufen 24. November, 2020]. URL: https://ci.apache.org/projects/flink/flink-docs-release-1.11/dev/stream/operators/windows.html.

[19] Jay Kreps, Neha Narkhede, Jun Rao u. a. „Kafka: A distributed messaging system for log processing“. In: *Proceedings of the NetDB*. Bd. 11. 2011, S. 1–7.

[20] Apache Software Foundation. *KAFKA STREAMS*. [Online; abgerufen 15. Februar, 2021]. URL: https://kafka.apache.org/documentation/streams/.

[21] Jay Kreps. *Introducing Kafka Streams: Stream Processing Made Simple*. [Online; abgerufen 14. Februar, 2021]. URL: https://www.confluent.de/blog/introducing-kafka-streams-stream-processing-made-simple/.

[22] M Haseeb Javed, Xiaoyi Lu und Dhabaleswar K Panda. „Characterization of big data stream processing pipeline: a case study using Flink and Kafka“. In: *Proceedings of the Fourth IEEE/ACM International Conference on Big Data Computing, Applications and Technologies*. 2017, S. 1–10.

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

[24] Matei Zaharia u. a. „Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing“. In: *Presented as part of the 9th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 12)*. 2012, S. 15–28.

[25] Apache Software Foundation. *Apache Storm*. [Online; abgerufen 15. Februar, 2021]. URL: https://storm.apache.org/.

[26] Diego Garcia-Gil u. a. „A comparison on scalability for batch big data processing on Apache Spark and Apache Flink“. In: *Big Data Analytics* 2.1 (2017), S. 1–11.

[27] Darshankumar Vinubhai Gorasiya. „Comparison of Open-Source Data Stream Processing Engines: Spark Streaming, Flink and Storm“. In: ().

[28] Martin Junghanns u. a. „Gradoop: Scalable graph data management and analytics with hadoop“. In: *arXiv preprint arXiv:1506.00548* (2015).

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

[30] Christopher Rost, Andreas Thor und Erhard Rahm. „Temporal graph analysis using gradoop“. In: *BTW 2019–Workshopband* (2019).

[31] Timo Adameit. „Evaluation des EPGM auf Basis von Apache Spark“. Magisterarb. Institut für Informatik, Leipzig, 2020.

[32] Elias Saalmann. „Relationale Abstraktion des EPGM unter Verwendung der Apache Flink“. Magisterarb. Institut für Informatik, Leipzig, 2019.

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

[34] Apache Software Foundation. *Apache Giraph*. [Online; abgerufen 12. Januar, 2021]. URL: `https://giraph.apache.org`.

[35] Sergey Brin und Lawrence Page. „The anatomy of a large-scale hypertextual web search engine“. In: (1998).

[36] Sherif Sakr u. a. *Large-scale graph processing using Apache Giraph*. Springer, 2016.

[37] Apache Software Foundation. *Introducing Gelly: Graph Processing with Apache Flink*. [Online; abgerufen 14. Dezember, 2020]. URL: `https://flink.apache.org/news/2015/08/24/introducing-flink-gelly.html`.

[38] Apache Software Foundation. *Introducing Gelly: Graph Processing with Apache Flink*. [Online; abgerufen 11. Januar, 2021]. URL: `https://spark.apache.org/docs/latest/graphx-programming-guide.html`.

[39] Apache Software Foundation. *Graph Algorithms*. [Online; abgerufen 11. Januar, 2021]. URL: `https://ci.apache.org/projects/flink/flink-docs-release-1.12/dev/libs/gelly/graph_algorithms.html`.

[40] Zainab Abbas u. a. „Streaming graph partitioning: an experimental study“. In: *Proceedings of the VLDB Endowment* 11.11 (2018), S. 1590–1603.

[41] Wenlei Xie u. a. „Dynamic interaction graphs with probabilistic edge decay“. In: *2015 IEEE 31st International Conference on Data Engineering*. IEEE. 2015, S. 1143–1154.

[42] Joan Feigenbaum u. a. „On graph problems in a semi-streaming model“. In: *Departmental Papers (CIS)* (2005), S. 236.

[43] Eric Miller. „An introduction to the resource description framework“. In: *Bulletin of the American Society for Information Science and Technology* 25.1 (1998), S. 15–19.

[44] Daniele Dell'Aglio u. a. „RSP-QL semantics: A unifying query model to explain heterogeneity of RDF stream processing systems“. In: *International Journal on Semantic Web and Information Systems (IJSWIS)* 10.4 (2014), S. 17–44.

[45] Daniele Dell'Aglio u. a. „On correctness in RDF stream processor benchmarking“. In: *International semantic web conference*. Springer. 2013, S. 326–342.

[46] Daniele Dell'Aglio u. a. „Towards a unified language for RDF stream query processing“. In: *European Semantic Web Conference*. Springer. 2015, S. 353–363.

[47] Jean-Paul Calbimonte, Jose Mora und Oscar Corcho. „Query rewriting in RDF stream processing“. In: *European Semantic Web Conference*. Springer. 2016, S. 486–502.

[48] Abdalrahman Alkamel. „Distributed Pattern Matching on Graph Streams“. Magisterarb. Leipzig University, 2020.

[49] János Dániel Bali. „Streaming graph analytics framework design“. Magisterarb. KTH, School of Information und Communication Technology (ICT), 2015.

[50] János Dániel Bali u. a. *Gelly Streaming*. `https://github.com/vasia/gelly-streaming`. 2019.

[51] Citi Bike. *Citi Bike: NYC's Official Bike Sharing System.* [Online; abgerufen 24. Juni, 2021]. URL: `https://www.citibikenyc.com/homepage`.

[52] Martin Junghanns, André Petermann und Erhard Rahm. „Distributed grouping of property graphs with GRADOOP". In: *Datenbanksysteme für Business, Technologie und Web (BTW 2017)* (2017).

[53] *Unary Logical Graph Operators.* [Online; abgerufen 11. Mai, 2021]. URL: `https://github.com/dbs-leipzig/gradoop/wiki/Unary-Logical-Graph-Operators#Transformation`.

[54] Shuai Ma u. a. „Capturing topology in graph pattern matching". In: *arXiv preprint arXiv:1201.0229* (2011).

[55] Matthew Saltz u. a. „Dualiso: An algorithm for subgraph pattern matching on very large labeled graphs". In: *2014 IEEE International Congress on Big Data.* IEEE. 2014, S. 498–505.

[56] Martin Junghans. *GDL - Graph Definition Language.* `https://github.com/s1ck/gdl`.

[57] Neo4j Team. *The Neo4j Cypher Manual v4.2.* [Online; abgerufen 11. Februar, 2021]. URL: `https://neo4j.com/docs/cypher-manual/current/`.

[58] Christopher Rost u. a. „Exploration and Analysis of Temporal Property Graphs". In: *24th International Conference on Extending Database Technology (EDBT).* OpenProceedings.org, 2021. ISBN: 978-3-89318-084-4.

[59] Jeyhun Karimov u. a. „Benchmarking Distributed Stream Data Processing Systems". In: *2018 IEEE 34th International Conference on Data Engineering (ICDE).* IEEE. 2018, S. 1507–1518.

[60] John L Hennessy und David A Patterson. *Computer architecture: a quantitative approach.* Elsevier, 2011.

[61] Erhard Rahm. *Mehrrechner-Datenbanksysteme: Grundlagen der verteilten und parallelen Datenbankverarbeitung.* Erhard Rahm, 1994.

Erklärung

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

„Flexible Graphstream-Analyse mittels Apache Flinks DataStream-API“

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 15. August 2021

Simon Bordewisch

A. Anhang

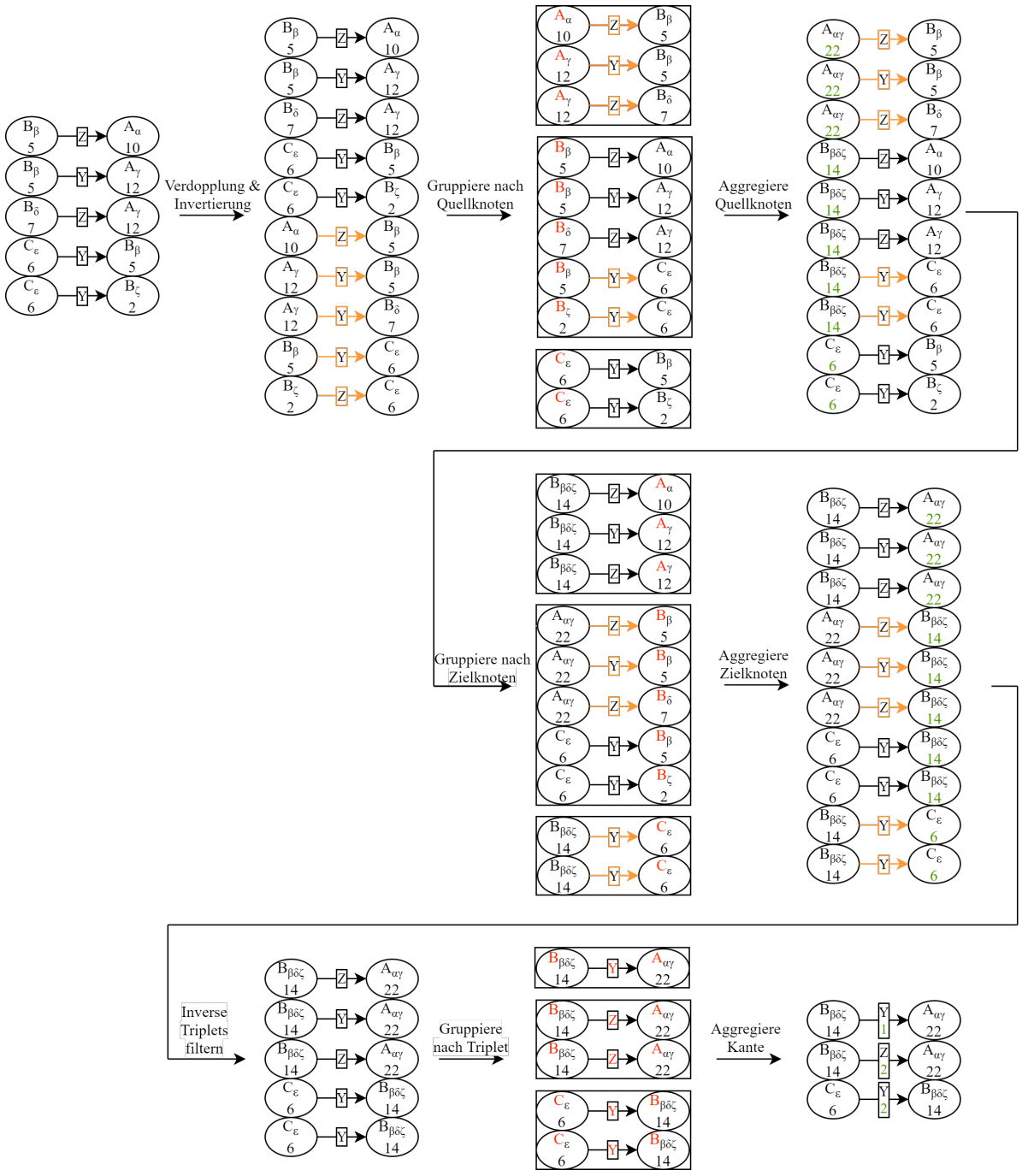


Abbildung A.1.: Beispiel für den gesamten Prozess in Distributed Windowed Grouping.

```
// Edge Transformation
stream.transformEdges(e -> {
  e.setProperty("edgetransformed", PropertyValue.create(true));
  return e;
})
// Vertex Transformation
stream.transformVertices(v -> {
  v.setProperty("vertexTransformed", PropertyValue.create(true));
  return v;
});
// Edge-induced Subgraph
stream.edgeInducedSubgraph(e -> e.hasProperty("gender"));
// Vertex-induced Subgraph
stream.vertexInducedSubgraph(v -> v.hasProperty("name"));
```

Code 12: Für die Evaluation generierte UDFs auf Nicht-Windowed Operatoren. Die für den Knoten-induzierter Subgraph definierte UDF wurde für die Latenztests invertiert, um die Latenz messen zu können.

Senkenparallelität	**Fest (96)**					**Skaliert wie Op.**				
Operatorparallelität	6	12	24	48	96	6	12	24	48	96
Kein Operator	*	*	*	*	447	139	243	291	310	447
Kanten-induzierter Subgraph	68	120	203	247	473	69	130	221	307	427
Knoten-induzierter Subgraph	134	230	340	338	499	117	206	299	330	430
Kanten-Transformation	65	113	196	243	468	65	126	213	304	429
Knoten-Transformation	65	109	195	220	469	68	123	184	302	431

Tabelle A.1.: Durchschnittlicher Durchsatz pro Millisekunde der Nicht-Windowed Operatoren. Da die Nutzung von maximalen Parallelismus für die Senke für keine Operation bedeutet, dass keinerlei Skalierung stattfindet, ist der Durchschnittswert dort überall gleich.

```
// Dual Simulation
stream.dualSimulationMatching("MATCH (v1)-[]->(v2)-[]->(v1)");
// Grouping-Operators (Distributed Grouping has the same parameters)
stream.callForGraph(AllWindowedGrouping.createGrouping()
        .addVertexGroupingKey("name") //Group by station name on vertices
        .addVertexAggregateFunction(new Count("used")) // Count the vertices in each
            super vertex
        .addEdgeGroupingKey("bike_id") //Group by bike IDs on edges
        .addEdgeAggregateFunction(new Count("used"))
        .build()); // Count the edges in each super edge
// Citi Bike-Pipeline
stream.vertexTransformation(v -> {
    var lat = v.getPropertyValue("lat").getFloat();
    var lon = v.getPropertyValue("long").getFloat();
    var gridCell = GeoUtils.mapToGridCell(lon, lat);
    v.setProperty("gridCell", gridCell);
    return v;
}).callForGraph(DistributedWindowedGrouping.createGrouping()
    .addVertexGroupingKey("gridCell")
    .addVertexAggregateFunction(new Count("stationsInGridCell"))
    .build()
).vertexInducedSubgraph(v -> v.getPropertyValue(gridCellKey).getInt() > 10);
```

Code 13: Für die Evaluation generierte Parameter auf Windowed Operatoren. Auf die Windowing-Informationen und -Zuweisung wird aus Übersichtsgründen verzichtet.

Window Size [ms]:	5	50	500	5.000
Dual Simulation	15,0	8,7	8,4	9,3
Zentrales Grouping	14,3	17,4	17,5	17,4
Zentrales Grouping (ein Superknoten)	18,1	19,1	18,0	19,6
Verteiltes Grouping (P=6)	22,9	24,6	21,7	20,1
Verteiltes Grouping (P=12)	43,5	42,2	41,7	32,6
Verteiltes Grouping (P=24)	61,1	66,3	71,0	41,2
Verteiltes Grouping (P=48)	92,7	97,4	92,5	67,9
Verteiltes Grouping (P=96)	135,7	134,8	137,6	90,1
Verteiltes Grouping (P=96, ein Superknoten)	7,4	5,5	4,5	4,4
Citi Bike-Pipeline (P=6)	19,8	19,9	19,4	12,8
Citi Bike-Pipeline (P=12)	32,7	35,8	34,9	28,1
Citi Bike-Pipeline (P=24)	50,8	49,8	54,1	36,1
Citi Bike-Pipeline (P=48)	61,0	72,6	71,4	53,7
Citi Bike-Pipeline (P=96)	81,3	86,2	89,4	58,9

Tabelle A.2.: Durchschnittlicher Durchsatz pro Millisekunde der Window-Operatoren bei verschiedenen Time Window Sizes.

Window Size [# Elemente]:	100.000	200.000	300.000	400.000	500.000
kein Operator	13767	23815	32653	44265	54275
Dual Simulation	22220	45095	71978	95605	122803
Zentrales Grouping	17330	30882	45347	58976	73645
Zentrales Grouping (ein Superknoten)	17436	29509	44129	56211	69000
Verteiltes Grouping (P=6)	16157	30053	42036	57238	66197
Verteiltes Grouping (P=12)	16246	30020	40904	56484	69873
Verteiltes Grouping (P=24)	16630	29614	42650	56824	65906
Verteiltes Grouping (P=48)	16234	29723	41133	55906	67879
Verteiltes Grouping (P=96)	16210	29628	41287	55801	68063
Verteiltes Grouping (P=96, ein Superknoten)	29109	60842	88401	126134	149077
Citi Bike-Pipeline (P=6)	15178	26251	36383	49352	60301
Citi Bike-Pipeline (P=12)	14891	26146	36779	49047	59660
Citi Bike-Pipeline (P=24)	14708	26036	36654	48985	59068
Citi Bike-Pipeline (P=48)	14739	26569	36676	48831	59933
Citi Bike-Pipeline (P=96)	14639	26573	36065	49403	59409

Tabelle A.3.: Durchschnittliche Verarbeitungszeit in Millisekunden der Window-Operatoren bei verschiedenen Windows Sizes (Anzahl der Elemente).