Institut für Parallele und Verteilte Systeme

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Lösungen für elastische Complex-Event-Processing Systeme

Benjamin Stutz

Studiengang:	Softwaretechnik
Prüfer/in:	Prof. Dr. Kurt Rothermel
Betreuer/in:	Henriette Röger, M.Sc.
Beginn am:	16. April 2018
Beendet am:	16. Oktober 2018

Kurzfassung

….. Short summary of the thesis ...

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

Immer mehr Geräte werden heute mit dem Internet vernetzt um unter anderem Informationen aus deren Verwendung zu erhalten. Aufgrund der zunehmenden Vernetzung steigt die Menge der verfügbaren Daten enorm an. Die große Anzahl der erzeugter Daten formiert sich dabei an den Endpunkten, zu denen sie geschickt werden, zu einem nicht endenden Zustrom von Daten.

Aus diesem Grund ist das Interesse in der Forschung im Bereich Datenstrom-Verarbeitung in den letzten Jahren stark aufgelebt. Eine Möglichkeit die eingehenden Daten zu verarbeiten ist Complex-Event-Processing (CEP). Bei dieser Methode werden SQL-ähnliche Abfragesprachen verwendet um den Datenstrom zu untersuchen. Die Abfrage kann sich dabei über mehrere Tupel im Datenstrom erstrecken und komplexe Muster erkennen. Wird ein Muster gefunden, wird dies typischerweise mit einem Event an eine Nachfolgende Stelle vermeldet. Diese kann ebenfalls wieder eine Mustererkennung auf den erhaltenen Events durchführen. So können mehrschichtige Systeme entstehen, die immer höherwertig werdende Informationen aus einem einfachen Datenstrom in Realzeit filtern können.

Ein Problem bei der Verarbeitung von Daten in Realzeit ist, dass die eingehenden Datenmengen Schwankungen unterliegen. Ein triviales Beispiel ist, dass Nachts weniger Geräte verwendet werden als tagsüber. Die Schwankung, die Nachts auftritt, ist vorhersehbar. Es gibt jedoch auch Erhöhungen der Datenmengen die nicht zwingend im Voraus erkennbar sind. In solchen Fällen muss sich die Verarbeitung der Daten dennoch an die Schwankung des Datenstroms anpassen. Die Fähigkeit des arbeitenden Systems, eine solche Anpassung an den anfallenden Arbeitsaufwand vorzunehmen, wird als Elastizität bezeichnet. Ein essentieller Beitrag zur Elastizität eines Systems liefern Cloud-Umgebungen, die es möglich machen Ressourcen dynamisch zu mieten mieten.

Um die Möglichkeiten der Cloud zu nutzen, muss das System selbst erst in der Lage sein dynamisch zu skalieren. In den letzten Jahren hat sich die Forschung damit beschäftigt, wie sich Elastizität in der Datenstromverarbeitung umsetzen lässt. Dabei sind verschiedene Problemstellungen zu lösen. Diese Problemstellungen werden in dieser Arbeit behandelt. Nahezu alle der vorgestellten Lösungen wurden dabei für ein bestimmtes System entwickelt. Das Ziel dieser Arbeit ist es, die generelle Einsatzfähigkeit der in der Forschung vorgestellten Lösungen für verschiedene CEP-Systeme zu ermöglichen und zu evaluieren.

Die vorliegende Arbeit beschreibt zuerst die Elastizität von CEP-Systemen und deren Notwendigkeit. Werden die Problemstellungen definiert, die sich für elastische CEP-Systeme ergeben. Anschließend werden verschiedene Lösungen aus der Forschung vorgestellt, die für ein automatisiertes Skalieren des CEP-System entwickelt wurden. Der Fokus in dieser Arbeit liegt dabei auf der Bestimmung des Parallelisierungsgrades der Operatoren in einer CEP-Topologie. Zwei dieser Algorithmen werden gewählt und im Rahmen der Arbeit implementiert und evaluiert.

Um die Algorithmen für verschiedene CEP-Systeme verfügbar zu machen, wird im weiteren Verlauf dieser Arbeit ein Framework implementiert. Zielsetzung des Frameworks ist, dass Algo-

rithmen, die den Parallelisierungsgrad einer Topologie steuern, unabhängig vom zu steuernden CEP-System implementiert werden können. Es soll ermöglicht werden, dass implementierte Algorithmen wiederverwendbar sind und für diverse CEP-Systeme eingesetzt werden können. Dazu stellt das Framwork eine API zur Verfügung, die das in der Forschung verwendete Graphen-Modell nachbildet. Auf Basis dieser API können Algorithmen implementiert werden, sodass sie auf dem abstrahierten Modell einer CEP-Topologie arbeiten. Die Erstellung des Topologie-Modells wird dabei automatisiert vom Framework übernommen und vom realen CEP-System ausgelesen.

Um das Framework an verschiedene CEP-Systeme anbinden zu können wird anschließend eine bestehende Lösung [**goggel_vergleich_2018**] verwendet und erweitert. Die Idee der Lösung ist, dass eine einheitliche REST-Schnittstelle verwendet wird um verschiedene CEP-Systeme zu steuern. Für jedes CEP-System kann so unabhängig ein Adapter erstellt werden, der die Spezifika des CEP-Systems kapselt und über die einheitliche REST-Schnittstelle zur Verfügung stellt. Die REST-Schnittstelle wird in der vorliegenden Arbeit stark erweitert, sodass die für das Graphen-Modell notwendigen Informationen bereit gestellt werden können. Anschließend wird der bestehende Adapter für Heron so angepasst, dass er die aktualisierte Version der REST-Schnittstelle implementiert.

Nachdem die Grundlage durch das Framework geschaffen ist, werden zwei Algorithmen aus der Forschung für die Bibliothek des Frameworks implementiert. Für die Implementation werden entsprechende Anpassungen und Verbesserungen an den Algorithmen umgesetzt und beschrieben. Zuletzt werden die beiden Implementationen evaluiert. Dazu wurde im Rahmen dieser Arbeit eine Topologie für Heron entwickelt, die Tweets von der Streaming-API von Twitter auf Merkmale wie Hashtags oder Länge analysiert. Für die Evaluation der Algorithmen steuern diese den Parallelisierungsgrad der Topologie nacheinander auf dem selben Datensatz. Die Diskussion der Ergebnisse aus der Evaluation findet im letzten Teil der Arbeit statt.

2 Verwandte Arbeiten

Im folgenden Kapitel sollen Arbeiten erwähnt werden, die verwandte Themenbereiche behandeln. Viele Arbeiten im Bereich der Datenstromverarbeitung beziehen sich nicht explizit auf CEP, treffen aber meist für diese Art der Verarbeitung ebenfalls zu.

Assuncao et al. stellen in Iherer Arbeit eine Übersicht über den aktuellen Stand von Datenstromverarbeitung auf [AVB17]. Sie definieren eine Klassifikation von Systemen zur Datenstromverarbeitung auf und stellen sie einzeln vor. Außerdem beschreiben Sie die verschiedenen Ansätze, die für Elastizität eines Systems zur Datenstromverarbeitung vorgeschlagen wurden. Unter anderem beschreiben Sie auch die Algorithmen, die für die Parallelisierung eingesetzt werden. Sattler et al. [SB13] diskutieren in ihrer Arbeit typische Muster, die bei elastischem Verarbeiten von Datenströmen auftreten.

Für die Verarbeitung von Datenströmen wurden bereits einige Systeme vorgeschlagen und implementiert. Die Autoren Lohrmann et al. [LWK14] stellen mit Nephele ein System vor, das die Größe der Ausgangszwischenspeicher dynamisch zur Laufzeit anpassen kann, um so Einfluss auf Latenz und Durchsatz zu nehmen. Akidau et al. [Aki+13] stellen in ihrer Arbeit MillWheel vor. Das von Google stammende System wurde speziell für Skalierbarkeit und Fehlertoleranz implementiert. Ein anderes von Twitter entwickeltes System wird von den Autoren Kulkarni et al. beschrieben [Kul+15]. Heron wurde auf der Basis von Apache Strom implementiert, da dieses die von Twitter gestellten Anforderungen nicht mehr erfüllen konnte.

Es gibt jedoch auch System, die speziell für CEP ausgelegt sind. In [WDR06] stellen Wu et al. das CEP-System vor SASE vor, das speziell für die Abfrage von RFID-Events implementiert wurde. Für die Abfrage der Events wurde von den Autoren eine eigene Sprache entwickelt. Ein weiteres CEP-System wurde durch Cugola et al. vorgestellt [CM12]. Das System T-Rex verwendet die von den selben Autoren erstellte Sprache TESLA [CM10] zur Definition der Abfragen. Eines der bekannteren Frameworks für CEP ist Esper [Noac], welches auch als kommerzielle Version für Hochverfügbarkeit angeboten wird.

Außerdem beschäftigen sich viele Arbeiten mit der Elastizität der Systeme. Die Arbeiten auf diesem Feld teilen sich in verschiedene Teilgebiete auf. Zum einen gibt es Arbeiten, die den Parallelisierungsgrad eines Operators mithilfe von Algorithmen bestimmen. Diese werden in Kapitel 3 dieser Arbeit genauer beschrieben.

Um die Parallelisierung möglich zu machen, ist es notwendig dass ein Mechanismus existiert, der Datenströme aufteilt. Zacheilas et al. beschreiben einen Ansatz, der parallelisierte Datenströme anhand ihrer unterschiedlichen Last optimal auf Rechner eines Clusters verteilt [Zac+16]. In [BTz13] beschreiben Balkesen et al. eine Vorgehensweise, die mehrere eingehende Datenströme aufteilen. In ihrem Paper schlagen die Autoren eine spezielle Vorgehensweise für Operatoren vor, die Fenster verarbeiten. Sie ermöglichen es damit selbst verschiebende Fenster an verschiedenen Instanzen des Operators weiterzugeben. Mayer et al. [MKR15] beschreiben ebenfalls einen Ansatz

für um einen Datenstrom aufzuteilen. Entgegen der oft verwendeten Verfahren, die auf einem Schlüssel im Tupel basieren, definieren die Autoren Partitionen im Splitter. Eine Partition wird durch Prädikate definiert, welche eine Partition starten und wieder schließen. Jedes Tupel wird auf diese Prädikate geprüft. Alle Tupel die zwischen einem Start und einem Ende der Partition den Operator erreichen gehören zur Partition. Partitionen sind den Instanzen des Operators zugewiesen.

Ein weiteres Feld der Forschung ist die Migration des Zustandes eines Operators. Wenn ein Operator eine neue Instanz erhält muss diese den bisherigen Zustand erhalten. Der Zustand muss dabei auch Rechner-übergreifend übergeben werden. Shah et al. lösen das Problem in ihrem Paper mit einer hohen Anzahl kleiner Partitionen [Sha+03]. Diese teilen von Beginn den Zustand des Operators auf und können bei Bedarf verschoben werden. Die Autoren gehen davon aus, dass eine Instanz des Operators zu einem bestimmten Zeitpunkt immer nur eine Untermenge der Partitionen exklusiv nutzt. Der zu dieser Partition korrespondierende Teil des Datenstroms muss über die Instanz , die die Partition nutzt, abgearbeitet werden. Castro Fernandez et al. [CF+13] benutzen einen ähnlichen Ansatz um den Zustand aufzuteilen. Allerdings werden die Partitionen bei jedem Skaliervorgang neu berechnet. Der Schlüsselraum wird neu unter den Instanzen aufgeteilt und die Partitionen entsprechend der Verteilung der Schlüssel neu berechnet. Matteis et al. [DMM16] berechnen die Partitionen ebenfalls bei jedem Skaliervorgang neu. Jedoch verteilen Sie bei ihrem Ansatz die Schlüssel nicht komplett neu. Nur die Partitionen, deren Schlüssel einer neuen Instanz zugewiesen wurde trennen den spezifizierten Bereich ab. Alle anderen Instanzen, deren Partitionen nicht betroffen sind, können so weiterhin die eintreffenden Tupel verarbeiten.

Zuletzt müssen Parallelisierte Operatoren noch auf die Rechner eines Clusters verteilt werden. Dabei sollten die Lasten möglichst gleichmäßig aufgeteilt sein. Zusätzlich gilt es zu beachten, dass Operatoren, die auf dem selben Rechner sind, mit geringerem Aufwand miteinander kommunizieren können. Heinze et al. betrachten das Problem als inkrementelles Behälterproblem [Hei+14b]. Für die Lösung des Problems ordnen Sie alle neu zu verteilenden Instanzen absteigend nach CPU-Anforderungen und fügen Sie dem ersten Rechner zu, der genug CPU frei hat. Zusätzlich werden Rechner bevorzugt auf dem benachbarte Operatoren liegen. Einen anderen Ansatz schlagen Ying Xing et al vor [YZJ05]. Sie berechnen eine Punktzahl für die Kombination zwischen jeder Instanz und jedem Rechner. Anschließend werden die Instanzen einzeln dem Rechner zugewiesen, der am meisten Kapazität zur Verfügung hat. Es wird immer die Instanz zugewiesen, die die höchste Punktzahl in Kombination mit dem Rechner ausweist.

3 Elastizität von CEP-Systemen

In diesem Kapitel wird der Begriff Elastizität für CEP-Systeme und die Notwendigkeit dieser Eigenschaft erläutert. Es werden Konzepte, mit denen ein CEP-System den Durchsatz an Daten erhöhen kann und deren Einsatzmöglichkeiten vorgestellt. Anschließend werden einige Algorithmen aus der Literatur untersucht, die die Elastizität eines CEP-Systems steuern können. Zwei der vorgestellten Algorithmen werden gewählt und im Verlauf dieser Arbeit implementiert und evaluiert.

CEP-Systeme ermöglichen es dem Anwender Datenströme zu verarbeiten. Der Unterschied zu reinen Datenstrom verarbeitenden Systemen liegt darin, dass die Operationen oft komplexer sind und einen Zustand besitzen [CVZ13]. Deshalb verwenden CEP-Systeme oft SQL-ähnliche Abfragesprachen, welche auf dem Datenstrom ausgeführt werden [CVZ13]. Die Festlegung, wie der Datenstrom verarbeitet wird, ist durch eine sogenannte Topologie beschrieben. Die Topologie wird durch Operatoren, die eine Funktion auf den Daten durchführen, und deren Abfolge definiert.

Das Ziel von CEP-Systemen ist die zuverlässige Verarbeitung von Daten in Realzeit. Laut Stonebraker et. al [SeZ05] ist eine der acht Anforderungen an Datenstromverarbeitung in Realzeit, dass das System skalierbar ist. Naturgemäß unterliegen die zu bearbeitenden Datenströme Schwankungen. Um diese Schwankungen abzufangen, muss sich das CEP-System, durch das die Daten abgearbeitet werden, an den ankommenden Datenstrom anpassen. Ist die Kapazität des Systems zu klein, dann verfallen die Daten oder können nicht mehr unter Realzeit entsprechenden Bedingungen abgearbeitet werden. Ein Ansatz ist, dass das System für die maximal auftretende Spitze, sofern diese bekannt ist, konfiguriert wird. Dies verursacht jedoch Kosten für Ressourcen, die gegebenenfalls über große Zeiträume nicht genutzt werden. Im Zeitalter von Cloud-Computing ist es aber möglich Ressourcen nur dann zu mieten, wenn Sie tatsächlich benötigt werden. Es ist daher ökonomisch sinnvoll einen Ansatz zu verfolgen, der die Ressourcen des CEP-Systems so anpasst, dass die Verarbeitung in Realzeit garantiert ist aber die Kosten minimal sind. Die Eigenschaft, dass ein System nach Bedarf Ressourcen dynamisch und automatisiert zuweisen oder entfernen kann und somit auf den aktuellen Arbeitsaufwand reagiert, wird als Elastizität bezeichnet [HKR]. Daraus folgt, dass ein CEP-System, das die Verarbeitung von Daten in Realzeit zum Ziel hat, nur öknomisch sinnvoll eingesetzt werden kann, wenn es elastisch ist.

Neben dem öknonomischen Aspekt gibt es eine weitere technische Erfordernis für die Elastizität von CEP-Systemen. Die genauen Schwankungen und somit auch die Auslastungsspitzen können nicht exakt bestimmt werden. Außerdem kann sich die durchschnittliche Menge der Daten im Datenstrom ändern. Somit ist eine Anpassung des CEP-Systems zumindest in größeren Zeitabständen notwendig. Ein CEP-System, das für eine Anpassung an einen eintreffenden Datenstrom abgeschaltet werden muss, kann währenddesssen keine Daten mehr entgegennehmen. Daher verfallen diese oder müssen durch einen Zwischenspeicher aufgefangen werden. Beide Methoden

verfehlen das Ziel einer konstanten Verarbeitung in Realzeit. Deshalb muss ein CEP-System, das Daten zuverlässig in Realzeit verarbeiten soll, elastisch sein.

3.1 Parallelisierung von Operatoren

Um ein System an einen anfallenden Arbeitsaufwand anzupassen, muss es skaliert werden. Eine triviale Möglichkeit die Leistung eines Systems zu verbessern besteht darin die Rechenleistung oder den Speicher erhöhen. Dieser Vorgang wird auch vertikales Skalieren genannt und ist erfolgreich, wenn die hinzugefügte Kapazität ausgeschöft werden kann. Jedoch stößt diese Methode an eine Grenze, wenn der physische Rechner keinen Platz für weitere Kapazitäten besitzt. Um diese Grenze zu umgehen, kann ein System horizontal skaliert werden. Bei dieser Variante wird die Last eines einzelnen Rechner auf weitere andere Rechner verteilt. Das horizontale Skalieren ist eine wichtige Vorgehensweise um Systeme im Cloud-Umfeld zu skalieren, da so einfach neue Rechenkapazität hinzugebucht werden kann. In der Regel bietet nur horizontales Skalieren die notwenidge Flexibilität für ein elastisches System.

Horizontales Skalieren erfordert jedoch, dass die Anwendung es erlaubt, Tätigkeiten gleichzeitig auf mehreren Rechnern auszuführen zu können. Um die Ressourcengewinnung durch horizontaes Skalieren zu nutzen, müssen die Operatoren einer Topologie parallelisiert werden. Die parallelisierten Teile des Operators können dann auf unterschiedlichen Rechnern ablaufen. Für die Parallelisierung stehen drei Optionen zur Verfügung [AVB17]:

- Parallelisierung durch Fließbandverarbeitung: Operatoren werden dadurch parallelisiert, dass ihre Funktion auf mehrere Operatoren aufgeteilt wird. Diese Operatoren werden hintereinander in der Topologie angeordnet. Jeder Operator der Reihe kann dann jeweils ein Tupel zur gleichen Zeit bearbeiten. Dies erfordert offensichtlich, dass die Funktionalität eines Operators bearbeitet wird und aufgeteilt werden kann. Dieses Erfordernis stellt auch die Limitierung des Verfahren dar, da ein Operator nicht beliebig oft geteilt werden kann. Um Elastizität zu erreichen ist diese Vorgehensweise nicht ausreichend, da sie limitiert ist und die Funktionen des Operators für den Skaliervorgang verändert werden müssen.
- Parallele Funktionen: Operatoren werden dadurch parallelisiert, dass auf dem gleichen Tupel unterschiedliche Funktionen parallel ausgeführt werden. Die Operatoren befinden sich dabei nicht hintereinander sondern werden nebeneinander abgearbeitet. Jedes Tupel wird zu jedem der parallel arbeitenden Operatoren gesendet. Dieses Vorgehen erfordert, dass die Funktionen der Operatoren es erlauben unabhängig voneinander durchgeführt zu werden. Dieses Verfahren wird ebenfalls dadurch limitiert, dass ein Operator nicht in beliebig viele nebenläufige Funktionen aufgeteilt werden kann. Zudem bauen in CEP-Systemen die von Operatoren versendeten Ergebnisse per Definition aufeinander auf, weshalb diese Methode ebenfalls ungenügend ist.
- Parallelisierung des Datenstroms: Bei dieser Form der Parallelisierung wird der Datenstrom, den ein Operator verarbeiten muss, aufgeteilt. Dafür werden mehrere Instanzen des Operators erzeugt, die die identische Funktion ausführen. Die Anzahl der Instanzen ist dabei gleich der Anzahl der geteilten Datenströme und wird als Parallelisierungsgrad des Operators bezeichnet. Jede Instanz bekommt dabei einen anderen Ausschnitt des Datenstroms zur Verarbeitung zugewiesen.

Die Parallelisierung des Datenstroms ist sehr effektiv, da die Funktion des Operators für den Skaliervorgang weitestgehend unbekannt sein kann und nicht verändert werden muss. Allerdings ist das Aufteilen des Datenstroms schwerer umzusetzen als die beiden anderen Varianten. Diese beiden Gründe sorgen dafür, dass diese Art der Parallelisierung die in der Forschung am meisten Untersuchte ist. Sie bietet das größte Potential, da sie weniger limitiert ist als die anderen beiden Verfahren, gibt aber auf folgende Problemstellungen auf:

- Die Höhe des Parallelisierungsgrades muss bestimmt werden.
- Die Aufteilung des Datenstroms muss definiert werden.
- Falls ein Operator einen Zustand besitzt, muss dieser über alle Instanzen gültig sein.

Durch die Parallelisierung darf das Ergebnis nicht anders ausfallen, als wenn es durch eine sequentielle Abfolge der Operators berechnet worden wäre. Dieses Eigenschaft wird auch als ”safety” des Operators bezeichnet [Ged+14]. Deshalb ist essentiell die Aufteilung des Datenstroms und des Zustandes des Operators zu betrachten. In der Forschung gibt es viele Arbeiten, die sich mit dieser Problematik auseinandersetzen. Einige dieser Arbeiten werden in Kapitel zwei genannt. Der Zentrale Themenpunkt der vorliegende Arbeit beschäftigt sich mit der Wahl des Parallelisierungsgrades eines Operators.

3.2 Bestimmung des Parallelisierungsgrades

Das Ziel eines optimalen Parallelisierungsgrades ist, dass das CEP-System die gestellten Anforderungen mit möglichst wenig Ressourcenverbrauch erfüllen kann. Um die optimalen Parallelisierungsgrade für die Operatoren einer Topologie zu finden, sind in der Forschung einige Algorithmen vorgestellt worden. Diese Algorithmen verwenden Messwerte, die vom CEP-System bereitgestellt werden, um Berechnungen durchzuführen. Sinnvollerweise werden die Algorithmen in Cloud-Umgebungen angewandt, da dort Ressourcen zum benötigen Zeitpunkt hinzu gebucht aber abbestellt werden können. Ist diese Elastizität für die Ressourcen nicht vorhanden, ist die Optimierung des Parallelisierungsgrades nicht effektiv. Da bei der Parallelisierung des Datenstroms der Operator als Blackbox angenommen wird, können Algorithmen aus dem Bereich der reinen Datenstromverarbeitung auf ein CEP-System angewendet werden.

Einige der Algorithmen verwenden Ansätze, die rein auf gemessene Überschreitung eines Grenzwertes reagieren. Ein sehr simpler Algorithmus wurde von Vogel et al. vorgestellt [Vog+]. Der Algorithmus prüft ob der Grenzwert für die Latenz eines Operators überschritten wird. Ist dies der Fall wird der Parallelisierungsgrad um eine benutzerdefinierte Schrittweite erhöht. Ist die gemessene Latenz niederer als ein Minimalwert, wird in der gleichen Schrittweite zurück skaliert.

Heinze et al. [Hei+14a] implementieren zwei verschiedene Grenzwert-basierte Algorithmen. Der lokale Grenzwert-Ansatz skaliert auf Basis der Grenzwerte für Unter-Auslastung beziehungsweise Überlastung eines Rechners. Dabei gibt es nach einer Skalierung eine Zeitperiode, in welcher der Rechner nicht mehr überprüft wird. Um häufiges Skalieren zu verhindern wird erst skaliert, wenn der Grenzwert mehrfach überschritten wurde. Beim globalen Grenzwert-Ansatz wird, anstatt der Auslastung eines einzelnen Rechners, die durchschnittliche Auslastung des gesamten Systems betrachtet. Der Algorithmus skaliert sobald Grenzwerte überschritten werden. Später haben die

Autoren den bestehenden Ansatz verbessert, indem sie die Grenzwerte automatisiert über die Optimierung einer Kostenfunktion festgelegt haben[Hei+15].

Gedik et al. [Ged+14] verwenden hingegen einen Ansatz bei dem der Grenzwert nicht für die gemessene Auslastung des Operators definiert wird. Sie definieren einen Grenzwert, der beschreibt wie oft der Operator ankommende Tupel nicht aufnehmen konnte. Wird der gesetzte Grenzwert überschritten, wird dies als Stauung erkannt. Außerdem messen sie den Durchsatz eines Operators und skalieren das System nach oben, wenn eine Stauung auftritt aber nur wenn die Skalierung einen positiven Effekt auf den Durchsatz hat. Operatoren werden hingegen nach unten skaliert, wenn keine Stauung auftritt und auf der vorherigen Ebene noch keine Stauung gemessen wurde. Die Parallelität des Operators wird pro Schritt um eins verändert.

In ihrem Paper stellen Liu und Buyya [LB17] einen Algorithmus vor, der ein Profil der Operatoren anlegt. Das Profil speichert die CPU-Dauer für Bearbeitung und Serialisierung je Tupel und den RAM, der jeweils für ein Tupels benötigt wird. Mit der Anzahl momentan ankommender Tupel kann mit dem Profil des Operators der Ressourcenverbrauch bestimmt werden. Außerdem wird über die Serialisierung berücksichtigt, wenn Operatoren auf dem selben Rechner liegen, da diese dann null ist. Liegen viele Operatoren auf dem selben Rechner wird insgesamt weniger CPU-Last erzeugt und somit kann auch der Parallelisierungsgrad kleiner ausfallen. Der Parallelisierungsgrad wird über Grenzwerte für die Parameter CPU für Bearbeitungsdauer, CPU für Serialisierung und RAM-Verbrauch berechnet. Immer wenn der Gesamtverbrauch des Operators einen der Grenzwerte überschreitet, wird er so lange aufgeteilt bis der Grenzwert eingehalten wird.

Hidalgo et al. verwenden ebenfalls einen Grenzwert-basierten Ansatz der die Auslastung des Operators begrenzt. Sie stellen dazu einen Algorithmus vor, der kurzzeitig reaktiv auf Schwankungen reagiert. Allerdings wird zusätzlich noch eine weitere Variante verwendet, die den Parallelisierungsgrad auf Basis von Vorhersagen der ankommenden Tupel berechnet. Diese Vorhersagen werden mithilfe einer Markov-Kette berechnet.

Andere Ansätze beruhen ausschließlich auf der Vorhersage der Menge von Tupeln und der Bearbeitungslatenz. Diese Ansätze versuchen das System präventiv zu skalieren, wenn in der Zukunft eine Änderung des Arbeitsaufkommens vorhergesagt wird. So stellen Kombi et al. [KLL17] einen Ansatz vor, bei dem die Last des Operators aus der zu erwartenden Anzahl ankommender Tupel und der Latenz der Verarbeitung eines Tupels vorhergesagt wird. Außerdem verwendet der Algorithmus zusätzlich eine Vorhersage der Ausgangstupel des Vorgänger-Operators um die Input-Tupel des zu betrachtenden Operators zu bestimmen. Der Operator wird für den höheren Wert aus beiden Vorhersagen skaliert.

Einen anderen Algorithmus der auf Vorhersage basiert schlagen Zacheilas et al. vor [Zac+15]. Die Vorhersage wird dabei durch eine Regression mittels Gauss-Prozessen berechnet. Die Vorhersage bestimmt die Anzahl eingehender und ausgehender Tupel eines Operators zu zukünftigen Zeitpunkten. Draus bestimmen die Autoren die Anzahl Tupel, die vom Operator nicht verarbeitet werden können. Die nicht verarbeiteten Tupel sind eine Variable für die Kostenfunktion aus der ein Graph mit möglichen zukünftigen Zustandsübergängen aufgebaut wird. Der Operator wird anhand der Zustandsübergänge des kürzesten Pfades durch den Graphen skaliert. Außerdem bezieht sich die Arbeit als eine der wenigen direkt auf CEP-Systeme.

Eine weitere Gruppe von Algorithmen benutzt das Modell der Warteschlangen-Theorie um das Verhalten von Tupeln vor einem Operator zu beschreiben. In der Arbeit von Mayer et al. [MKR15] schlagen die Autoren einen Ansatz mit Warteschlangen-Theorie vor. Die Autoren beziehen sich

in Ihrer Arbeit ebenfalls auf CEP-Systeme. Die Autoren verfolgen das Ziel, die Warteschlange von Tupeln vor einen Operator unter einem benutzerdefinierten Grenzwert zu halten. Um die Länge der Warteschlange zu bestimmen wird ein mathematisches Modell aus der Warteschlangetheorie verwendet. Dazu werden die benötigten Werte für das Modell gemessen und mithilfe der Methoden der Zeitreihenanalyse vorhergesagt. Die Autoren berechnen mit den vorhergesagten Messwerten und dem Profil des Operators die Wahrscheinlichkeit, dass die Warteschlange kürzer oder gleichlang wie der maximale Grenzwert ist. Liegt diese Wahrscheinlichkeit über einem benutzerdefinierten Grenzwert, wird der Parallelisierungsgrad so lange angepasst, bis der Grenzwert erreicht wird.

Ein weiterer präventiver Ansatz mit dem Modell der Warteschlangen-Theorie wurde von Matteis et al. vorgeschlagen. Für Ihr Modell messen die Autoren die Rate aller ankommenden Tupel sowie die durchschnittliche Verarbeitungsdauer eines Operators. Anschließend wird eine Vorhersage basierend auf der Zeitreihe der gemessenen Werte durchgeführt. Die Autoren nennen den gleitenden Durchschnitt als möglichen Ansatz, legen aber kein spezielles Vorhersagemodell fest. Mit der Vorhersage kann die zukünftige Antwortzeit eines Operators berechnet werden. Sie ist die Summe aus durchschnittlicher Bearbeitungsdauer und der Zeit, für die sich ein Event in der Warteschlange des Operators befindet. Für die Bestimmung des Parallelisierungsgrades wird eine Kostenfunktion optimiert, welche abhängig vom Parallelisierungsgrad und der Antwortzeit ist. Diese berücksichtigt die Kosten durch Ressourcenverbrauch, Kosten für das Skalieren und die Kosten für Verletzung der maximal festgelegten Antwortzeit eines Operators.

Ein weiterer Algorithmus mit Warteschlangen-Theorie wurde von Lohrmann et al. vorgeschlagen [LJK15]. Das Ziel des Algorithmus es Latenzen für die Bearbeiten eines Tupels einzuhalten. Dazu wird ein benutzerdefinierter Grenzwert festgelegt. Mit den im System erfassten Messwerten berechnet der Algorithmus reaktiv mit Hilfe der Warteschlangen-Theorie die Gesamtlatenz des Tupels. Der Parallelisierungsgrad des Operators wird Schrittweise erhöht, bis die Gesamtlatenz unter dem Maximalwert liegt.

Wie zuvor beschrieben kann die Elastizität eines CEP-Systems nur in einer Cloud-Umgebung sinnvoll genutzt werden. Deshalb gibt es auch Algorithmen die die Bereitstellung von Cloud-Ressourcen berücksichtigen. Ein Algorithmus dieser Art schlagen Hochreiner et al. vor. Für die Elastizität des Systems berücksichtigen die Autoren die Zeiteinheit der Rechnungsstellung um bereits bezahlte Ressourcen nicht vorzeitig zurück zu geben. Ressourcen mit kürzerer Laufzeit werden früher zurückgegeben falls nach unten skaliert wird. Der Algorithmus skaliert die Operatoren, sobald die benutzerdefinierten Grenzwerte für Überlastung oder Unterbelastung überschritten werden.

Für die genauerer Betrachtung in dieser Arbeit wurde je ein Algorithmus der präventiv agiert und einen der reaktiv agiert ausgewählt. Zudem sollten die Algorithmen verschiedene Modelle besitzen. Im weiteren Verlauf der Arbeit wird zum einen der Algorithmus von Lohrmann et al. als rein reaktiver Ansatz mit dem Warteschlangen-Modell implementiert und betrachtet. Dessen Ziel ist die konstante Einhaltung der Latenz eines Tupels. Außerdem wurde der Algorithmus von Zacheilas et al. als rein präventiver Algorithmus gewählt. Dieser wählt den Parallelisierungsgrad durch die Optimierung einer Kostenfunktion über einen Graphen. Die exakte Funktionsweise und die vorliegende Implementation der Algorithmen ist in Kapitel ********** beschrieben.

4 Topologie-Modell

CEP-Systeme bilden das Grundgerüst für die Verarbeitung von Datenströmen. Sie werden benutzt um sogenannte Topologien zu entwickeln, die von den Komponenten des CEP-Systems ausgeführt werden. Topologien definieren über eine Abfolge von Operatoren, wie die Verarbeitung des Datenstroms stattfindet. Die Annahmen, die über eine Topologie getroffen werden, wirken sich auf Messwerte der Topologie und den Ablauf der Verarbeitung des Datenstroms aus. Deshalb sollen in diesem Kapitel das für die vorliegende Arbeit verwendete Modell vorgestellt werden.

4.1 Topologie als Graph

In der Literatur werden Topologien oft als Graph dargestellt. Dabei werden Operatoren als Knoten und der Fluss des Datenstroms zwischen Operatoren als Kanten dargestellt. Das in dieser Arbeit verwendete Topologie-Modell besteht aus zwei unterschiedlichen gerichteten azyklischen Graphen. Die Darstellung der Topologie durch zwei Graphen wurde von [LJK15] verwendet. Ein Graph beschreibt den logischen Aufbau der Topologie, während der andere Graph die physische Ausdehnung der Topologie beschreibt. Die Graphen sind dabei direkt voneinander abhängig und stellen verschiedene Sichtweisen auf die Topologie dar. In Abbildung 5.1 ist der logische Aufbau einer Topologie dargestellt. Sie beschreibt die Abfolge von Operatoren, die die Tupel des Datenstroms verarbeiten.

In der dargestellten Topologie sind zwei Pfade beschrieben. Ein Pfad ist durch die Verkettung der Operatoren mit der grünen Linie dargestellt. Ein Weiterer wird durch die braune Linie beschrieben. Das erste Element eines Pfades ist die sogenannte Quelle, eine spezielle Form des Operators. Sie

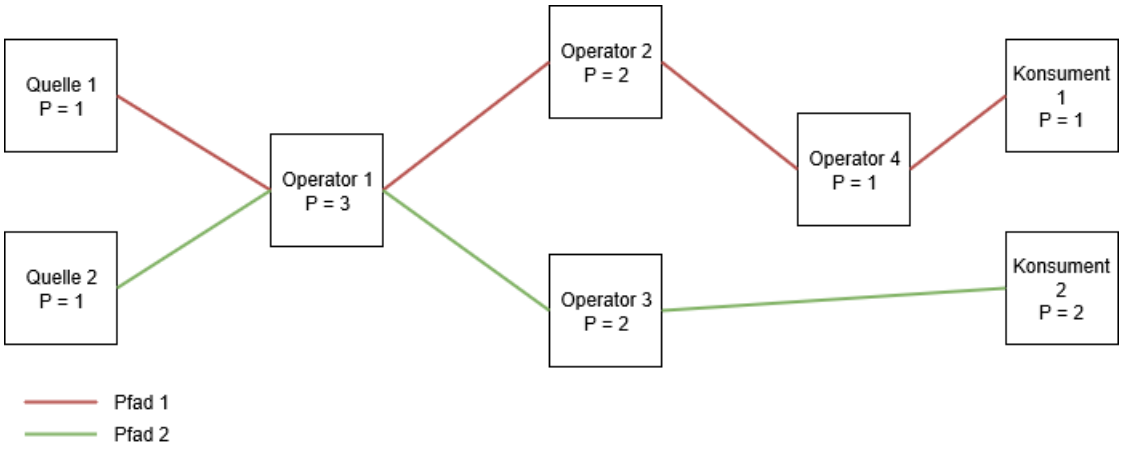

Abbildung 4.1: Modell der logischen Topologie

führt Tupel aus einem externen Datenstrom in die Topologie ein. Tupel sind einzelne Elemente eines Datenstroms, z.B. ein JSON-Objekt. Das Ende eines Pfads ist immer der Konsument. Dieser ist der letzte Operator an denen die Tupel aus der Quelle verarbeitet werden. Oft werden die Tupel hier an ein anderes System weiter gegeben, in eine Datenbank geschrieben oder auch verworfen. Die Tupel bewegen sich entlang eines Pfades.

Zwischen der Quelle und dem Abfluss befinden sich weitere Operatoren. Diese beinhalten Funktionen, die die Tupel bearbeiten. Der Operator sendet das Tupel an den nächsten Operator auf dem festgelegten Pfad. Der nächste Operator wird auch als Nachfolger bezeichnet. Der Operator, von dem der Operator Tupel empfängt wird als Vorgänger bezeichnet. So ist Operator 1 der Vorgänger von Operator 2, der wiederum der Nachfolger von Operator 1 ist. Ein Operator kann, wie in der Abbildung skizziert, auch auf mehreren Pfaden vorhanden sein. Allerdings kann ein spezifischer Operator nur einmal innerhalb eines Pfades auftreten, da es sich um einen azyklischen Graph handelt.

In der gezeigten Topologie wird davon ausgegangen, dass Tupel selektiv weitergeleitet werden können. Das bedeutet, dass ein Operator, welcher sich auf mindestens zwei Pfaden befindet, die Tupel nicht an alle Nachfolger sendet. Er selektiert stattdessen welche Tupel er an welchen Nachfolger sendet. Wie in der Abbildung dargestellt sendet Operator 1 die Tupel von Quelle 2 nur an Operator 3 weiter. Die Tupel von Quelle 1 werden nur an Operator 2 weitergeleitet. Würden die Tupel jeweils an beide Folgeoperatoren (Operator 2 und Operator 3) weitergeleitet werden, würde die Topologie vier Pfade aufweisen. Dann beschreibt ein Pfad den Weg von Quelle 1 zu Konsument 2 und der vierte Pfad von Quelle 2 zu Konsument 1.

Alle Operatoren besitzen einen Parallelisierungsgrad, welches direkt den Aufbau des zweiten Graphen bestimmt. Der Parallelisierungsgrad ist mindestens eins und kann einen beliebigen aber fixierten Maximalwert nicht überschreiten. Die Quelle besitzt laut diesem Modell zwar einen Parallelisierungsgrad, wird jedoch von den nachfolgend implementierten Algorithmen, die die Topologie skalieren, nicht berücksichtigt. Die Quelle wird in vielen Arbeiten als nicht parallelisierbar modelliert und wird deshalb von vielen Algorithmen nicht betrachtet.**********

Abbildung 5.2 zeigt die beiden Pfade in der Form, wie sie im CEP-System physisch ausgeprägt sind. Dies entspricht dem zweiten Graph im vorgestellten Topologie-Modell. Jeder Operator besitzt genau die Anzahl Tasks, die durch den Parallelisierungsgrad festgelegt ist. Ein Task ist eine Instanz, welche die Funktion des logischen Operators ausführt. Zwischen den Tasks befinden sich Kanäle, über die diese miteinander kommunizieren. Ein Task hat dabei immer einen Kanal zu allen Tasks des folgenden Operators. So kann ein Tupel von diesem zu jedem beliebigen Task des Nachfolgers gesendet werden. Die Tasks der Topologie können auf verschiedenen Rechnern verteilt sein und über ein Netzwerk kommunizieren. Das Modell geht dabei davon aus, das allen Tasks die gleiche Größe von Rechenleistung und Speicher für die Ausführung der Operation zur Verfügung stehen. Da Quelle und Konsument jeweils die Enden der Topologie darstellen, gibt es keine eingehenden Kanäle an der Quelle sowie keine ausgehenden Kanäle an dem Abfluss.

Wird ein Tupel an den Nachfolger gesendet, so wird es immer nur an einen einzigen Task des Nachfolgers verschickt. Für die Vorgehensweise wie der Task des Nachfolgers ausgewählt gibt es verschiedene Ansätze. Dies kann zum einen per Zufall geschehen. Eine andere Möglichkeit ist das Mapping zu einem bestimmten Task über ein Feld des Tupels. Eine weitere Möglichkeit ist das Weiterleiten zu einem speziellen Task über einen bestimmten Zeitraum. Die letzten beiden Methoden sind sinnvoll, wenn der Folgeoperator eine Funkion implementiert die ein Fenster

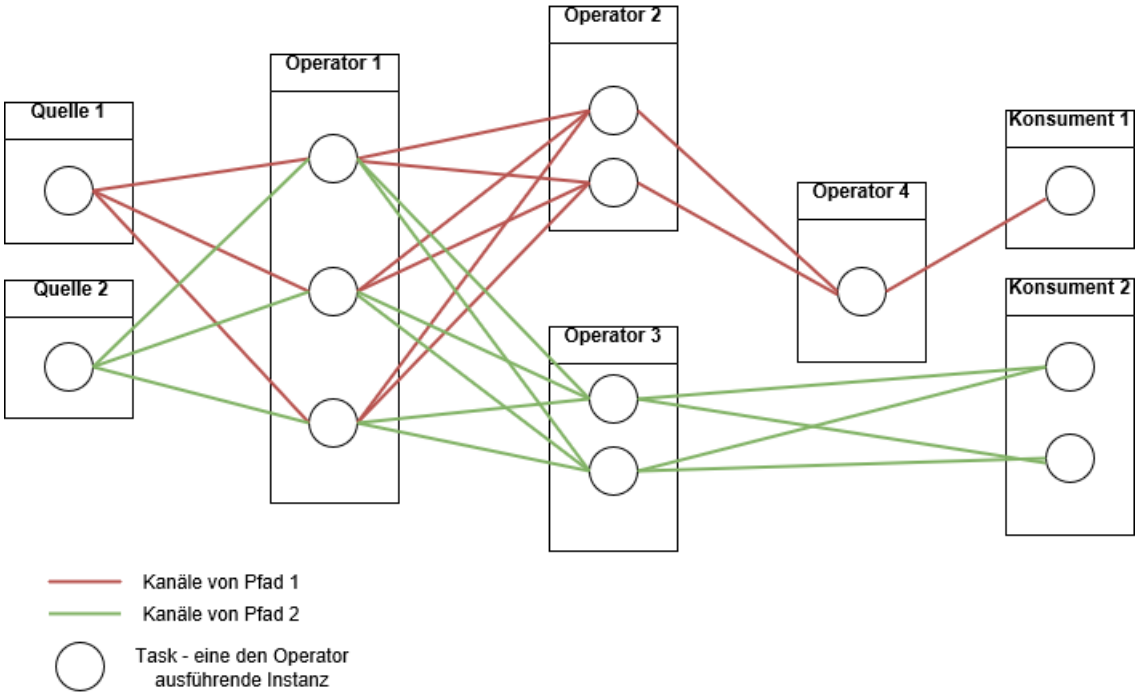

Abbildung 4.2: Modell der physischen Topologie

verwendet. Die Entscheidung, wie die Tupel zwischen Tasks weitergeleitet werden, hat keinen Einfluss auf die selektive Weiterleitung, die zwischen Operatoren auf logischer Ebene definiert ist.

4.2 Messwerte

Um den Datenfluss durch die Topologie zu überwachen, werden Messwerte erfasst. Diese dienen der Analyse der Topologie als Ganzes und der Untersuchung einzelner Operatoren. Wenn ein Operator eines Pfads sehr langsam arbeitet, bremst er alle Tupel des gesamten Pfads. Mit Hilfe der Messwerte können solche Engpässe identifiziert und durch Skalieren des Operators behoben werden. Die in Kapitel 4 vorgestellten Algorithmen berechnen mit Hilfe der Messwerte den neuen Parallelisierungrad eines Operators.

Das Modell geht davon aus, dass das die gemessenen Werte eines Operators unabhängig von dessen Pfad sind. Dies bedeutet, dass ein Operator, der auf zwei oder mehr verschiedenen Pfaden liegt, bei den Messwerten nicht zwischen den Pfaden unterscheidet. Diese Annahme erfordert, dass Tupel unabhängig davon über welchen Pfad sie beim Operator ankommen den gleichen Aufwand beanspruchen. Sind die Messwerte für die Tasks eines Operators bekannt, kann der Messwert für den Operator aus den Messwerten der Tasks aggregiert werden. Zähler werden dabei summiert, für die anderen Messwerte der Durchschnitt ermittelt. Für das Modell werden folgenden Messwerte definiert. Die Messwerte, die für Tasks beschrieben sind, gelten unter der Annahme, dass sie aggregiert wurden, ebenso für Operatoren.

- Pfad-Latenz / Tupel-Latenz: Beschreibt die durchschnittliche Dauer in Millisekunden, die ein Tupel benötigt die Topologie von der Quelle zum Konsumenten zu durchqueren. Die Dauer beginnt mit ab der Abgabe des Tupels durch die Quelle bis es vom Konsumenten erfolgreich abgearbeitet wurde. Die Messung ist nur zuverlässig, wenn Quelle und Konsument auf dem gleichen Rechner platziert sind, da Uhren verschiedene Rechner nicht zu 100% synchron sind.
- Fehlgeschlagene Tupel: Gibt die Anzahl Tupel an, bei denen die Verarbeitung in der Topologie nicht erfolgreich war. Ursache kann unter anderem ein Fehler während der Bearbeitung durch einen Operator sein.
- Bestätigte Tupel: Anzahl der Tupel, die erfolgreich durch die Topologie abgearbeitet wurden.
- Task-Latenz: Die durchschnittliche Dauer in Millisekunden, die ein Task benötigt um ein Tupel an den nächsten Operator weiter zu geben. Diese unterscheidet sich bei Operatoren die ein Fenster verwenden und Operatoren die kein Fenster verwenden [LJK15]. Ein Fenster beschreibt einen Operators der Tupel aufnimmt, um auf allen gesammelten Tupeln eine Operation durchzuführen. Meistens erfordert die Operation mehrere Tupel um Zusammenhänge zu erkennen. Die Latenz des ersten aufgenommenen Tupel ist dann länger als die des Tupels welches zuletzt in das Fenster aufgenommen wurde, da sie alle zusammen wieder weitergeleitet werden wenn das Fenster voll ist. Verwendet der Operator kein Fenster, so wird das Tupel direkt nach der Aufnahme verarbeitet und weitergeleitet. Anschließend wird das nächste Tupel aufgenommen.
- Anzahl Ausführungen des Tasks: Misst die Anzahl wie oft der Task ein neues Tupel aufgenommen und verarbeitet hat.
- Anzahl eingehender Tupel: Misst die Anzahl der Tupel, die beim Task ankommen. Der Messwert ist nicht identisch mit der Anzahl Ausführungen des Tasks, da sich vor dem Task ein Buffer befindet, in dem Tupel zwischengespeichert werden können bevor sie verarbeitet werden.
- Anzahl ausgehender Tupel: Misst die Anzahl der Tupel, die vom Task versendet wurden. Tupel die auf mehreren Pfaden ausgeben werden, werden nicht mehrfach gezählt.
- Abarbeitungsdauer: Beschreibt die durchschnittliche Dauer in Millisekunden, die zwischen dem Aufnehmen von Tupel zur Bearbeitung liegt. Die Abarbeitungsdauer ist für Operatoren ohne Fenster gleich der Latenz [LJK15].
- Tupel-Ankunftsintervall: Beschreibt die durchschnittliche Dauer in Millisekunden, die zwischen der Ankunft von zwei Tupeln am Task vergeht.
- Auslastung: Gibt die Auslastung des Tasks in Prozent an. Die Auslastung kann unter anderem durch $fracAbarbeitungsdauerTupel - Ankunftsintervall$ berechnet werden.
- Varianz der Bearbeitungsdauer: Gibt die Varianz der Stichprobe von Messwerten der Bearbeitungsdauer an.
- Varianz des Tupel-Ankunftsintervalls: Gibt die Varianz der Stichprobe von Messwerten des Tupel-Ankunftsintervalls an.

- Latenz des Kanals: Gibt die durchschnittliche Latenz eines Kanals zwischen zwei Tasks in Millisekunden an. Die Dauer beginnt ab der Ausgabe des Tupels durch den Vorgänger-Task bis zur Aufnahme des Tupel durch den Nachfolge-Task. Die Latenz des Kanals beinhaltet insbesondere die Zeit im Ausgangs-Zwischenspeicher des Vorgänger-Task, die Netzwerklatenz und die Zeit im Eingangs-Zwischenspeicher des Nachfolge-Task.
- Latenz der Stapelverarbeitung: Die durchschnittliche Dauer in Millisekunden, die ein Tupel im Ausgangs-Zwischenspeicher des Vorgänger-Task liegt.

5 Implementierug des Frameworks

Dieses Kapitel beschäftigt sich mit der Architektur und Impelentierung des Framworks. Das Framework schafft eine Abstraktionsebene für die Algorithmen, die die aktiven Topologien in den CEP-Systemem skalieren. Das Framework soll ermöglichen, die einzelnen Komponenten in der Topologie zu skalieren. Die logische Struktur der Topologie wird dabei nicht verändert. Das vorliegende Modell ermöglicht die Steuerung mehrerer CEP-Topologien, welche auch über diverse CEP-Systeme verteilt werden können.

5.0.1 Architektur

Die Architektur des Frameworks besteht aus mehreren logischen Komponenten und verfolgt zwei Ziele. Eines der beiden Hauptziele der Architektur ist, dass das System einfach um weitere Algorithmen erweitert werden kann. Um dieses Ziel zu erreichen, wurde eine Abstraktionsebene eingeführt, welche die Eigenschaften des realen CEP-Systems repräsentiert. Diese Abstraktion wird durch ein Graphen-Modell repräsentiert, welches einen gerichteten, azyklischen Graphen modelliert. Die implementierten Algorithmen arbeiten ausschließlich auf dem abstrahierten Modell. Für die Implementation eines neuen Algorithmus, müssen nur die Daten aus dem Modell ausgelesen und anschließend verarbeitet werden. Außerdem ist es möglich, mehrere Algorithmen für das gleiche Modell auszuführen um ihre Ergebnisse zu vergleichen. Um einen Algorithmus für das System zu implementieren ist ausschließlich Wissen über das Modell notwendig.

Ein weiteres Ziel ist, dass das System für weitere CEP-Systeme außer Heron verwendet werden kann. Dazu wurde eine API, die bereits in einer vorhergehenden Bachelorarbeit entwickelt wurde, erweitert. Alle Komponenten des Systems benutzen diese API für die Kommunikation mit dem CEP-System, sodass diese alle system-spezifischen Befehle abstrahieren kann. Die Seite der API, welche mit dem System direkt kommuniziert ist ein eigenständiger Adapter. Dieser Adapter wird auf dem Rechner installiert, von dem das zu steuernde CEP-System kontrolliert wird. Der Adapter ist über eine REST-API ansprechbar. Somit kann das Framework auf einem eigenständigen Rechner installiert werden und das Zielsystem über die REST-Schnittstelle des Adapters kontrollieren. Dies ermöglicht die Kontrolle von mehreren CEP-Systemen zur gleichen Zeit. Eine Erweiterung des Frameworks für weitere Systeme über die Implementierung eines entspreche4nden Adapters zu erreichen. Für die Erstellung eines neuen Adapters ist ausschließlich Wissen über die Spezifikation der REST-Schnittstelle notwendig. Das Framework kann so über die REST-API verschiedene CEP-Systeme über eine einheitlichen Schnittstelle ansprechen.

Die Architektur des Systems ist in der Abbildung 5.1 dargestellt. Im Folgenden werden die einzelnen Komponenten der Architektur genauer erläutert.

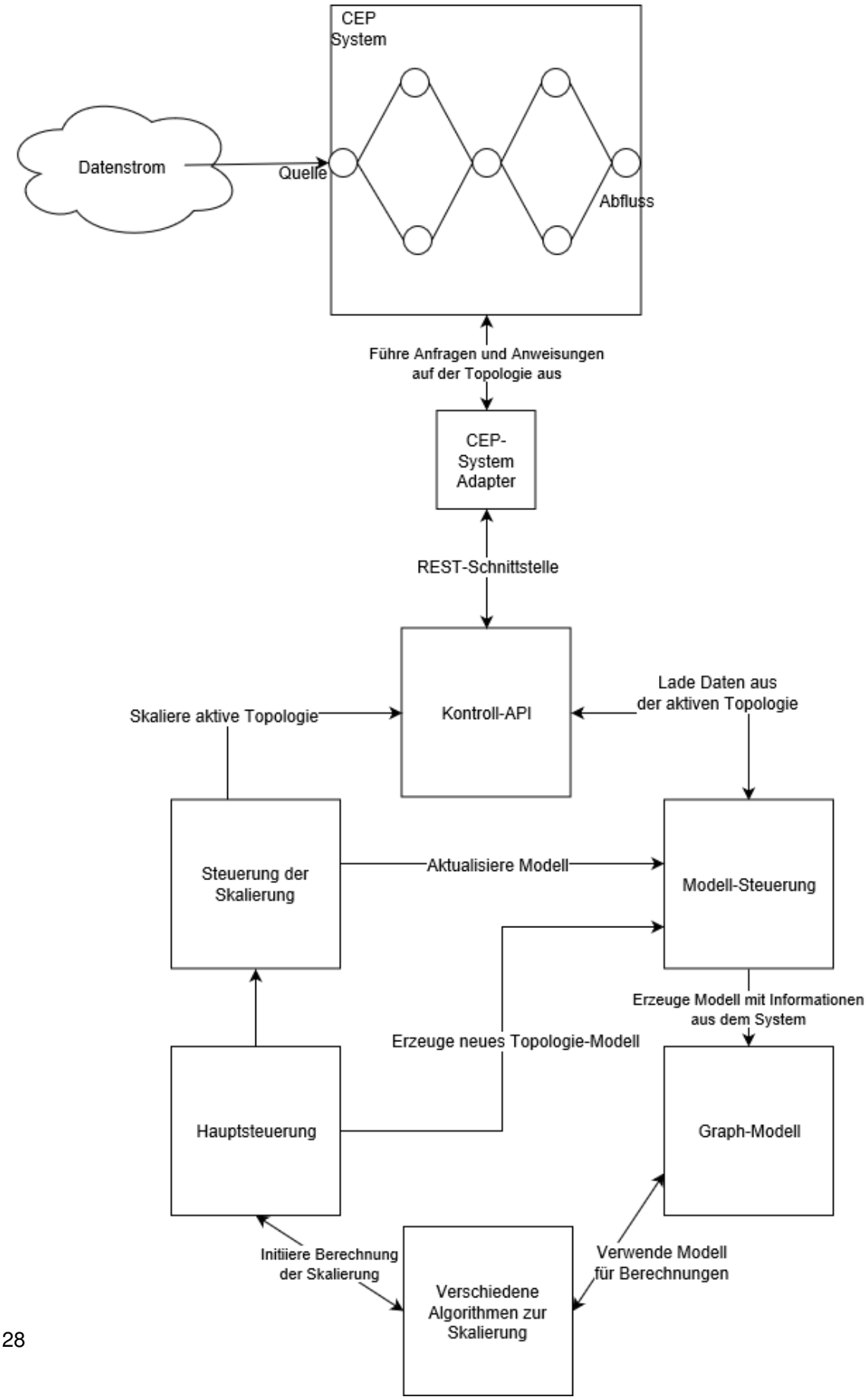

Abbildung 5.1: Architektur des Systems

5.1 Graph-Modell

Das Graph-Modell repräsentiert die Topologie im realen CEP-System. Alle Elemente der Topologie werden nach dem in Kapitel ********** vorgestellten Topologie-Modell abgebildet. Es dient als Cache für Messdaten aus dem realen System. Durch die Zwischenspeicherung wird ermöglicht, dass ein Modell eines bestimmten Zeitpunktes des realen Systems zur Verfügung gestellt werden kann. Würden die Algorithmen die Werte zur Laufzeit abfragen, also immer dann, wenn die Werte benötigt werden, kann dies zu einem inkonsistenten Modell führen. Die Algorithmen verwenden das Modell, um die Inkosistenz von Messungen zu verschiedenen Zeitpunkten zu verhindern.

Jedes Modell besteht aus Pfaden und Operatoren. Jeder Pfad stellt dabei eine geordnete Folge von Operatoren dar. Das Modell erlaubt kreuzende Pfade, sodass Operatoren in mehreren Pfaden verwendet werden können. Momentan kann im Modell keine selektive Weiterleitung der Tupel repräsentiert werden. Dies bedeutet, dass ein Operator alle Tupel, die er versendet, an immer an alle folgenden Operatoren weiterleitet. Operatoren sind über einen Namen identifizierbar und haben eine dem Parallelisierungsgrad entsprechende Anzahl an Tasks. Sobald der Operator einen neuen Parallelisierungsgrad erhält, wird die entsprechende Anzahl Tasks gelöscht oder erzeugt. Somit ist die Anzahl Tasks immer gleich dem Parallelisierungsgrad des Operators. Der maximale und minimale Parallelisierungsgrad aller Operatoren kann über zwei Konstanten angegeben werden. Diese gelten für alle Operatoren in allen Modellen. Ein logisch schlüssiger Minimalwert ist der Parallelisierungsgrad 1. Ein Parallelisierungsgrad kleiner als eins ist für einen aktiven Operator nicht gültig, da er keinen ausführenden Task besitzt. Das Maximum kann entsprechend der Ressourcen, die im CEP-System zur Verfügung stehen, angepasst werden. Pfade und Operatoren stellen die logische Struktur der Topologie dar.

Die ausführende Ebene, oder physische Struktur, wird durch Tasks und Kanäle repräsentiert. Tasks sind die Recheninstanzen welche die Operation eines Operators auf Tupel ausführen. In der vorliegenden Implementation des Graph-Modells ist die Zwischenspeicherung der Messwerte auf Task-Ebene vorgesehen. Somit wird ermöglicht, dass ein detailliertes Modell der Topologie im CEP-System erzeugt werden kann. Die ID der Tasks wird vom Operator automatisch erzeugt und wird aus dem Namen des Operators und einer fortlaufenden Nummer wie folgt gebildet: *<Name>_<Nummer>*. Die niedrigste Nummer ist immer die eins. Die höchste Nummer entspricht immer dem aktuellen Parallelisierungsgrad des Operators. Für Tasks sind folgende Messwerte in der aktuellen Version des Modells vorgesehen:

- Latenz des Tasks
- Anzahl Ausführungen des Tasks
- Anzahl eingehender Tupel
- Anzahl ausgehender Tupel
- Auslastung
- Bearbeitungsdauer
- Tupel-Ankunftsintervall
- Varianz der Bearbeitungsdauer
- Varianz des Tupel-Ankunftsintervalls

Für die Messwerte der Tasks wird die Annahme getroffen, dass sie unabhängig vom Pfad sind. Dies bedeutet zum Beispiel, dass alle eingehenden Tupel die selbe Bearbeitungsdauer benötigen. Unabhängig davon von welchem vorhergehenden Operator sie stammen. Alle für den Task erfassten Metriken differenzieren nicht die Herkunft der Tupel, sondern erfassen sie gesammelt.

Für viele Algorithmen wird nicht der Messwert eines einzelnen Tasks betrachtet, jedoch können die Werte gemittelt werden um einen aussagekräftigen Messwert für den gesamten Operator zu bekommen.

Die Kommunikation zwischen den Tasks wird durch die Kanäle repräsentiert. Diese speichern Metriken der Kommunikationskanäle zwischen Tasks. Für die Implementation des Modells wurde die Annahme getroffen, dass alle Tasks eines Operators mit allen Tasks der vorhergehenden und nachfolgenden Operatoren kommunizieren können. Die Anzahl an Kanälen zwischen zwei Operatoren ist also das Produkt aus deren Parallelisierungsgraden. Für Kanäle sind folgende Messwerte vorgesehen:

- Latenz des Kanals
- Latenz der Stapelverarbeitung

Details zu den Messwerten werden in Kapitel ********* erläutert.

5.2 Modell-Steuerung

Die Modell-Steuerung erzeugt die Modellstruktur der Topologien und füllt diese anschließend mit Messwerten. Alle Aktionen werden von der Hauptsteuerung initiiert. Im ersten Schritt wird die Topologie vom CEP-System über die Kontroll-API ausgelesen. Welche Topologie ausgelesen wird entscheidet die angegebene Adapteradresse. Ein Adapter ist jeweils für eine Topologie zuständig. Zuerst werden die Pfade der Topologie ausgelesen und im Graph-Modell die entsprechenden Operatoren und Pfade angelegt. Dann wird der Parallelisierungsgrad der Operatoren gesetzt. Diese erzeugen ,wie zuvor beschrieben, durch das setzen des Parallelisierungsgrades die entsprechende Anzahl an Tasks und Kanälen.

Anschließend erzeugt die Modell-Steuerung eine Zuweisung von Tasks im realen System zu den Tasks im Modell. Die Tasks im Modell sind, wie im vorherigen Kapitel beschrieben, durchnummeriert und haben einen durch das Graph-Modell festgelegten Namen. Diese Namen weichen sehr wahrscheinlich von den Namen im realen CEP-System ab. Damit nun die Messwerte eines realen Tasks konsistent einem modellierten Task zugewiesen werden können, wird pro Operator eine Zuweisungstabelle für die Tasks erzeugt. Diese Zuweisung muss mit jeder Änderung des Parallelisierungsgrades des Operators angepasst werden, da Tasks wegfallen oder hinzu kommen. Ist die Zuweisung erfolgt werden zuletzt die Messwerte über die Kontroll-API in das Modell geladen.

5.3 Steuerung der Skalierung

Die Steuerung der Skalierung übernimmt zwei Aufgaben. Zum Einen werden die Ergebnisse der ausgeführten Algorithmen an das CEP-System geben. Dazu wird die Kontroll-API verwendet, der Adapter steuert die Anpassung der Parallelisierungsgrade in der aktiven Topologie. Als Zweites wird nach erfolgreicher Anpassung der realen Topologie noch das Modell angepasst. Hier werde ebenfalls die neuen Parallelisierungsgrade, welche die Algorithmen berechnet haben, gesetzt.

5.4 Hauptsteuerung

Die Hauptsteuerung ist die zentrale Steuereinheit des Frameworks. Sie enthält die Logik, die die anderen Komponenten kontrolliert und die zuvor beschriebenen Abläufe steuert. Sie instanziiert die anderen Komponenten für jede Topologie, die gesteuert werden soll. Außerdem werden die Parameter definiert, die für Berechnungen in den Algorithmen verwendet werden. Die zeitliche Abfolge aller Abläufe zu steuern ist die Kernaufgabe der Komponente.

5.5 Kontroll-API

Die Kontroll-API abstrahiert die APIs der verschiedenen CEP-Systeme. Grundsätzlich besteht Sie aus zwei Teilen. Ein Teil nimmt die Aufrufe aus dem Framework entgegen und wandelt sie in REST-Anfragen um. Die Anfragen werden anschließend an den entsprechenden Adapter gesendet. So wird die REST-Schnittstelle über die zur Verfügung gestellten Funktionen gekapselt. Dieser Teil ist auf der gleichen Maschine wie die anderen Komponenten des Frameworks und wird direkt von den anderen Komponenten angesprochen.

Der zweite Teil, der Adapter, befindet sich auf der Maschine des zu steuernden CEP-Systems. Er nimmt REST-Anfragen entgegen und setzt diese in Befehle für die API des CEP-Systems um. Der Adapter ist somit der Teil, der die API des CEP-System auf eine einheitliche REST-Schnittstelle abstrahiert. Außerdem kapselt der Adapter die zurückgelieferten Messwerte des CEP-Systems und vereinheitlicht sie für das Graph-Modell. Deshalb ist die Implementation des Adapters essentiell für die Qualität des Graph-Modells. Die Implementation des Adapters für das CEP-System Heron wird in Kapitel ********** ausführlich behandelt.

Die Kommunikation über die REST-API ermöglicht es, dass die beiden teile auf verschiedenen Maschinen installiert sein können. Durch die Fern-Steuerung über die Schnittstelle können mit einer einzelnen Installation des Frameworks mehrere CEP-Systeme und Topologien gleichzeitig gesteuert werden.

5.6 UML-Diagramm des Frameworks

6 Adapter für Heron

Wie bereits in der Architektur des Frameworks beschrieben, stellt der Adpater die Komponente dar, die eine uniforme Steuerung verschiedener CEP-Systeme ermöglicht. Der Adapter kapselt die spezifischen Eigenschaften eines CEP-Systems und dessen API. Die Funktionen des Adapters werden über eine einheitliche REST-Schnittstelle veröffentlicht. So können von einer zentralen Komponente mehrere Adapter über das Netzwerk mit dem HTTP-Protokoll angesprochen werden. Dies ermöglicht die unabhängige Platzierung von CEP-Systemen und des Frameworks. Der Adapter wird normalerweise auf dem Rechner platziert, auf dem die Steuer-Komponente des CEP-Systems liegt. Allerdings muss dies je nach Art der API des CEP-Systems nicht zwingend notwendig sein. Das folgende Kapitel befasst sich mit der Umsetzung des Adapters für das CEP-System Heron.

6.1 Implementierung

Das Ziel des Adapters ist eine ausgewählte Topologie im System ansprechen zu können. Ein Adapter ist daher immer nur für eine spezifische Topologie zuständig. Es ist aber möglich mehrere Adapter auf unterschiedlichen Ports zu starten um mehrere Topologien ansprechen zu können. Diese Entscheidung wurde zur Vereinfachung der REST-API getroffen. So müssen nicht bei jedem Aufruf alle Parameter angegeben werden, die eine spezifische Topologie identifizieren. Die korrekte IP-Adresse und der korrekte Port genügen um den Adapter für eine Topologie anzusprechen. Außerdem können die Merkmale, die eine Topologie identifizieren, bei unterschiedlichen CEP-Systemen variieren. Mit der gewählten Architektur können die Merkmale beim Start des Adapter angegeben werden. Jede Implementation eines Adapters kann die Start-Parameter beliebig festlegen und bietet somit Flexiblität für die Definition der Topologie.

Um den Adapter für Heron zu Starten werden folgende Parameter benötigt:

- URL: Definiert die Adresse unter der der Adapter erreichbar ist.
- Tracker URL: Definiert die Adresse unter der die Heron-API erreichbar ist.
- Cluster: Bestimmt den Cluster in dem die Ziel-Topologie ausgeführt wird.
- Topologie: Der Name mit dem die Topologie identifiziert wird.
- Umgebung: Wird von Heron ebenfalls benutzt um die Topologie zu identifizieren.
- Heron CLI: Spezifiziert den Pfad zu den ausführbaren Programmen von Heron.
- CLI Clustername: Spezifiziert welcher wie das Cluster über die CLI angesprochen werden kann.

- Messwert-Interval: Definiert die Zeitspanne der Messwerte, die ausgelesen werden, in Sekunden.

Der Adapter ist in eine eigenständige JAR-Datei verpackt, die auf jedem Rechner mit installierter Java Laufzeitumgebung ausgeführt werden kann. Beim Ausführen wird ein glassfish-Server gestartet. Dieser nimmt Anfragen über die REST-API entgegen.

Werden über die REST-API Metriken oder der Aufbau der Topologie angefragt, leitet der Adapter die Anfrage an Heron weiter. Heron bietet für die Abfrage von Messwerten selbst eine REST-Schnittstelle an. Die Implementation des Adapters kapselt die REST-Schnittstelle von Heron und abstrahiert sie. Messwerte werden immer für die in den Startparametern definierte Topologie abgefragt. Wie Metriken von Heron bereit gestellt und wie diese vom Adapter angepasst werden ist im folgenden Kapitel detailliert ausgeführt.

6.2 Metriken in Heron

Ein essentieller Teil des Adapters ist die Anpassung der vom CEP-System gelieferten Messwerte an die im Graph-Modell erwarteten Werte. CEP-Systeme erfassen Messwerte auf unterschiedliche Arten oder legen unterschiedliche Messpunkte für den Wert fest. Das Wissen, wie CEP-Systeme die Messwerte erfassen, muss durch den Adapter gekapselt werden. Die Annahmen die das Graph-Modell trifft, werden im Kapitel ********** diskutiert.

Heron arbeitet mit einem verteilen System für die Erfassung der Messdaten. Alle Container, die einen Task enthalten, sammeln die Messwerte autonom. Jede Minute werden die Werte von den einzelnen Containern zu einer zentralen Einheit, dem sogenannten Metrik-Manager, gesendet. Dieser sammelt die Daten und sichert sie zentral für die Bedienung der Anfragen.

Auf der Webseite von Heron wird angegeben dass die erfassten Messwerte bis zu drei Stunden vorgehalten werden [Noab]. Alle Werte die vor dieser Zeitspanne erfasst wurden sind nicht mehr verfügbar. Die Messwerte von Heron werden in einer Auflösung von einer Minute erfasst und gesichert. Die Messwerte können von Heron als Zeitreihe mit der Auflösung von einer Minute abgefragt werden. Alternativ können die Messwerte von Heron direkt aggregiert abgerufen werden. Da das Graph-Modell nur aggregierte Messwerte und keine Zeitreihen unterstützt, werden vom Adapter direkt die aggregierten Messwerte abgefragt. Der Parameter Messwert-Interval, der definiert werden muss, um den Adapter zu starten, bestimmt die Zeitspanne für die die Messwerte aggregiert werden. Der Endzeitpunkt der Spanne ist immer der Zeitpunkt t, an dem der Adapter die Anfrage an Heron sendet. Wenn der Parameter *Messwert-Interval* durch s repräsentiert wird so werden alle Werte die in der folgenden Zeitspanne $[t - s, t]$ aggregiert. Heron bietet über die REST-API standardmäßig die untenstehenden auf den Operator bezogenen Messwerte an. Die Metriken unterscheiden sich bei Quell-Operatoren und anderen verarbeitenden Operatoren. Betrachten wir zuerst die Quell-Operatoren:

- __ack-count: Dieser Wert beschreibt die Anzahl an Tupel, die durch das CEP-System bereits erfolgreich abgearbeitet an den Quell-Operator zurückgemeldet wurden. Dieser Messwert wird nur erfasst, falls die Topologie für die Bestätigung der Tupel konfiguriert und implementiert ist.

- __fail-count: Dieser Wert gibt die Anzahl an Tupel an, bei denen während der Verarbeitung Fehler aufgetreten sind. Tritt ein Fehler auf wird dies bei entsprechender Implementation an den Quell-Operator gemeldet. Dieser Messwert wird nur gemessen, falls die Topologie für die Bestätigung der Tupel konfiguriert ist.
- __emit-count: Gibt die Anzahl von Tupel an, die von dem Quell-Operator ausgegeben wurden.
- __complete-latency: Gibt die Gesamtlatenz für eine erfolgreiche Bearbeitung eines Tupels in der Topologie an. Dieser Wert kann von der Quelle berechnet werden, wenn die erfolgreiche Verarbeitung des Tupels zurückgemeldet wird.

Verarbeitende Operatoren:

- __execute-count: Jeder verarbeitende Operator besitzt eine Methode, die von Heron aufgerufen wird und ankommende Tupel verarbeitet. Dieser Wert gibt an, wie oft die verarbeitende Methode aufgerufen wurde. Dieser Messwert gibt keine Aussage über die Anzahl ausgegebener Tupel.
- __ack-count: In der Methode, die die Tupel verarbeitet, kann zu einem beliebigen Zeitpunkt die erfolgreiche Verarbeitung des Tupels an den Quell-Operator signalisiert werden. Dieser Messwert gibt die Anzahl der Tupel an, die von diesem Operator an den Quell-Operator als ërfolgreich Verarbeitetßurückgemeldet wurden.
- __fail-count: Ebenso kann in der Methode, die die Tupel verarbeitet, das Auftreten eines Fehlers an Quell-Operator signalisiert werden. Der Zeitpunkt der Benachrichtigung ist wiederum vollständig von der Implementierung der Methode abhängig. Der Messwert gibt die Anzahl der an den Quell-Operator signalisierten Fehler an.
- __execute-latency: Dieser Messwert beschreibt die durchschnittliche Laufzeit der Methode, welche die Tupel verarbeitet in Nanosekunden. Ein Tupel kann zu jedem beliebigen Zeitpunkt der Methode ausgegeben werden. Tupel können auch an mehreren Zeitpunkten während der Ausführung der Methode ausgegeben werden. Dies bedeutet, dass dieser Messwert nicht zwingend die Dauer bis zur Ausgabe des Tupels darstellt. Dies trifft ebenso für die Benachrichtigungen über Erfolg oder Fehlschlag an den Quell-Operator zu. Ob der Messwert für die Latenz des Tupels im Operator aussagekräftig ist, hängt von der Implementation des Operators ab.
- __process-latency: Dieser Wert gibt die Dauer in Nanosekunden an, die benötigt wurde um die Verarbeitung eines Operators als erfolgreich oder fehlerhaft zu melden. Er misst jedoch nicht die Gesamtlaufzeit des Tupels im System. Gemessen wird ausschließlich die Zeit von Beginn der verarbeitenden Methode bis zu dem Zeitpunkt, an dem die Benachrichtigung, ob das Tupel erfolgreich verarbeitet oder fehlerbehaftet war, erzeugt wird. Die Rückmeldung innerhalb der Methode kann, wie oben beschrieben, zu einem beliebigen Zeitpunkt während der Ausführung geschehen.
- __emit-count: Gibt die Anzahl von Tupel an, die von dem verarbeitenden Operator ausgegeben wurden.

Eine exakte Definition der Messwerte, die von Heron erfasst werden, ist dem momentanen Stand der Heron-Dokumentation nicht zu entnehmen. Dehalb wurde die Bedeutung, der von Heron bereitgestellten Messwerte, empirisch festgestellt.

Alle vorangegangenen Ausführungen beziehen sich auf die Standardeinstellungen von Heron in der Version 17.05. Alle Konfigurationen können in den Files *metrics_sinks.yaml* und textitheron_internals.yaml in dem Verzeichnis des verwendeten Clusters angepasst werden. Außerdem bietet Heron die Möglichkeit Messdaten in Drittsysteme zu exportieren. Somit bietet sich die Möglichkeit für Algorithmen, die auf Basis von alten Daten das Verhalten der Topologie erlernen, Messdaten länger aufzubewahren.

6.3 REST-Schnittstelle

Welche Operationen der Adapter implementieren muss, wird durch die REST-Schnittstelle festgelegt. Im Detail implementiert der Adapter ein Java Interface. Der glassfish-Server veröffentlicht dann die im Interface definierten Methoden über die REST-Schnittstelle. Die Schnittstelle stellt für jeden Messwert im Graph-Modell eine Methode bereit, über die der Wert ausgelesen werden kann. Die vollständige Dokumentation der Schnittstelle ist in Tabelle ********** zu sehen. Die REST-Schnittstelle antwortet im JSON Format.

Betrachtet man die im vorherigen Kapitel die standardmäßig von Heron gelieferten Messwerte so ist ersichtlich, dass Heron nicht alle benötigten Messwerte für die REST-Schnittstelle liefert. Deshalb verlangt die Implementation des Adapters, dass die fehlenden Werte bestmöglich approximiert werden. Sämtliche Latenzen werden vor der Weitergabe über die Schnittstelle auf Millisekunden umgerechnet. Im Modell wird von einer Read-Read Latenz ausgegangen, sodass die Bearbeitungsdauer gleich der Latenz des Operators ist. Ein weiterer Punkt ist der Input von Tupeln am Operator. Dieser wird in Heron nicht erfasst. Da im Graph-Modell aber davon ausgegangen wird, dass keine selektive Weiterleitung von Tupeln stattfindet, sind die ausgegebenen Tupel des Vorgänger-Operators gleich den eingehenden Tupel. Die Varianzen der jeweiligen Werte werden mit Hilfe der von Heron gelieferten Zeitreihe berechnet. Die Größe der Stichprobe ist die Anzahl der minütlichen Werte, die von Heron geliefert werden. Sie ist somit von dem gewählten Messwert-Intervall abhängig.

Tabelle 6.1: Operationen der REST-Schnittstelle

Operation	/v1/getalloperators
Ergebnis	[”Operator1”, ”Operator3”, ”Operator2”]
Beschreibung	Liefert alle Operatoren unsortiert in einem Array mit Strings zurück
Operation	/v1/getlogicaltopology
Ergebnis	{”Pfad1”:[”Operator1”,”Operator3”], ”Pfad2”:[”Operator1”,”Operator2”]}
Beschreibung	Liefert alle Pfade der Topologie als Objekt zurück. Jedes Objekt besitzt ein Array mit den Operatoren in korrekter Reihenfolge
Operation	/v1/getoperatorparallelism?operator=Operator3

Tabelle 6.1: Operationen der REST-Schnittstelle

Ergebnis	4
Beschreibung	Liefert den Parallelisierungsgrad des Operators als Integer
Operation	/v1/getpathlatency?operator=Operator1
Ergebnis	80.3
Beschreibung	Liefert die durchschnittliche Latenz der Pfade, bei denen der Operator die Quelle ist, als Gleitkommazahl in Millisekunden zurück. Die Operation ist nur für Quell-Operatoren erlaubt.
Operation	/v1/getfailedtuplescount?operator=Operator1
Ergebnis	0.0
Beschreibung	Liefert die Anzahl der Tupel, bei denen der Operator die Quelle und die Verarbeitung fehlgeschlagen ist. Die Operation ist nur für Quell-Operatoren erlaubt.
Operation	/v1/getackedtuplescount?operator=Operator1
Ergebnis	100000.0
Beschreibung	Liefert die Anzahl der Tupel, bei denen der Operator die Quelle ist und erfolgreich bearbeiten wurden. Die Operation ist nur für Quell-Operatoren erlaubt.
Operation	/v1/getlatency?operator=Operator2&operation=AVG
Ergebnis	5.615
Beschreibung	Liefert die Latenz des Operators als Gleitkommazahl in Millisekunden. Dazu werden die Werte der einzelnen Tasks aggregiert. Erlaubte Operationen sind AVG, SUM, MAX, MIN. Die Operation ist für Quell-Operatoren nicht erlaubt
Operation	/v1/getoperatorlatency?operator=Operator2
Ergebnis	{”Task1”:5.32, ”Task2”:5.01}
Beschreibung	Liefert ein Objekt, das die Tasks des Operators beinhaltet. Jeder Task hat als Wert die Latenz des Tasks als Gleitkommazahl zugewiesen. Die Operation ist für Quell-Operatoren nicht erlaubt.
Operation	/v1/getoperatorlatencyvariance?operator=Operator2
Ergebnis	{”Task1”:3.33, ”Task2”:3.87}
Beschreibung	Liefert ein Objekt, das die Tasks des Operators beinhaltet. Jeder Task hat als Wert die Varianz der Latenz des Tasks als Gleitkommazahl zugewiesen. Die Operation ist für Quell-Operatoren nicht erlaubt.
Operation	/v1/getoutputcount?operator=Operator1&operation=SUM
Ergebnis	100000.0

Tabelle 6.1: Operationen der REST-Schnittstelle

Beschreibung	Liefert die Anzahl der Tupel, die der Operator ausgegeben hat. Dazu werden die Werte der einzelnen Tasks aggregiert. Erlaubte Operationen sind AVG, SUM, MAX, MIN.
Operation	/v1/getoperatoroutputcount?operator=Operator1
Ergebnis	{”Task1”:50000, ”Task2”:50000}
Beschreibung	Liefert ein Objekt, das die Tasks des Operators beinhaltet. Jeder Task hat als Wert die Anzahl der Tupel, die der Task ausgegeben hat, zugewiesen.
Operation	/v1/getoperatoroutputcountvariance?operator=Operator1
Ergebnis	{”Task1”:3.33, ”Task2”:3.87}
Beschreibung	Liefert ein Objekt, das die Tasks des Operators beinhaltet. Jeder Task hat als Wert die Varianz der Tupel, die der Task ausgegeben hat, als Gleitkommazahl zugewiesen.
Operation	/v1/getexecutioncount?operator=Operator2&operation=SUM
Ergebnis	100000.0
Beschreibung	Liefert zurück wie oft der Operator ausgeführt wurde. Dazu werden die Werte der einzelnen Tasks aggregiert. Erlaubte Operationen sind AVG, SUM, MAX, MIN. Die Operation ist für Quell-Operatoren nicht erlaubt.
Operation	/v1/getoperatorexecutioncount?operator=Operator2
Ergebnis	{”Task1”:50000, ”Task2”:50000}
Beschreibung	Liefert ein Objekt welches die Tasks des Operators beinhaltet. Jeder Task hat als Wert die Anzahl wie oft der Task ausgeführt wurde. Die Operation ist für Quell-Operatoren nicht erlaubt.
Operation	/v1/getoperatorexecutioncountvariance?operator=Operator2
Ergebnis	{”Task1”:3.33, ”Task2”:3.87}
Beschreibung	Liefert ein Objekt, das die Tasks des Operators beinhaltet. Jeder Task hat als Wert die Varianz der Anzahl wie oft der Task ausgeführt wurde als Gleitkommazahl zugewiesen. Die Operation ist für Quell-Operatoren nicht erlaubt.
Operation	/v1/getinputcount?operator=Operator2&operation=SUM
Ergebnis	100000.0
Beschreibung	Liefert die Anzahl der Tupel, die beim Operator angekommen sind. Dazu werden die Werte der einzelnen Tasks aggregiert. Erlaubte Operationen sind AVG, SUM, MAX, MIN. Die Operation ist für Quell-Operatoren nicht erlaubt.
Operation	/v1/getoperatorinputcount?operator=Operator2
Ergebnis	{”Task1”:50000, ”Task2”:50000}

Tabelle 6.1: Operationen der REST-Schnittstelle

Beschreibung	Liefert ein Objekt, das die Tasks des Operators beinhaltet. Jeder Task hat als Wert die Anzahl der Tupel, die beim Task angekommen sind, zugewiesen. Die Operation ist für Quell-Operatoren nicht erlaubt.
Operation	/v1/getoperatorinputcountvariance?operator=Operator2
Ergebnis	{”Task1”:3.33, ”Task2”:3.87}
Beschreibung	Liefert ein Objekt, das die Tasks des Operators beinhaltet. Jeder Task hat als Wert die Varianz der Tupel, die beim Task angekommen sind, als Gleitkommazahl zugewiesen. Die Operation ist für Quell-Operatoren nicht erlaubt.
Operation	/v1/setoperatorparallelism?operator=Operator2?parallelism=4
Ergebnis	true
Beschreibung	Setzt den Parallelisierungsgrad des Operators auf den definierten Wert. Liefert true wenn der Vorgang erfolgreich war, false wenn nicht.
Operation	/v1/setmultipleoperatorparallelism?map=Operator2:4;Operator3:2
Ergebnis	true
Beschreibung	Setzt den Parallelisierungsgrad der Operatoren auf den definierten Wert. Liefert true wenn der Vorgang erfolgreich war, false wenn nicht.

6.4 UML-Diagramm

7 Algorithmus mit Warteschlangen-Theorie

Lohrmann et al. beschreiben in ihrem Paper [LJK15] einen Algorithmus, der das Ziel verfolgt, die Latenz der Tupel einer Topologie unter einem, durch den Benutzer bestimmten, Maximalwert zu halten. Dieses Ziel soll mit einem möglichst geringen Verbrauch von Ressourcen erreicht werden. Die Latenz eines Tupels bestimmt sich aus der Zeit, welche das Tupel benötigt um von der Quelle zum Konsument zu gelangen.

Dementsprechend ist die Wahl des Pfades für die Latenz des Tupels essentiell, da die sie mindestens die Summe der Latenz aller Operatoren in einem Pfad ist. Außerdem fließt die Zeit, in der Tupel sich zwischen Operatoren bewegen, in die Latenz des Tupels mit ein. Einerseits beinhaltet dies die Latenz des Netzwerks, über das die Tupel versendet werden. Diese Latenz kann jedoch nicht durch reines Skalieren der Operatoren beeinflusst werden, sondern ist von deren Platzierung abhängig. Im Modell des Algorithmus wird die Netzwerklatenz nicht explizit berücksichtigt. Der zweite Faktor ist die Zeit, welche zwischen der Ankunft eines Tupels im Zwischenspeicher des Operators und der Bearbeitung des Tupels durch den Operator liegt. Diese Zeit zwischen Ankunft und Bearbeitung wird in dem Algorithmus durch ein Modell aus der Warteschlangentheorie abgebildet. Die Dauer, in der sich ein Tupel in der Warteschlange vor der Bearbeitung durch den Operator befindet, wird folgend als Wartezeit bezeichnet. Der Algorithmus bedient sich der Kingman-Formel aus der Warteschlangen-Theorie um die Wartezeit zu berechnen. Sie modelliert eine Warteschlange mit einem einzelnen Abnehmer. Da die genannten Faktoren alle vom gewählten Pfad des Tupels abhängig sind, kann man auch von der Latenz eines Pfades sprechen.

Der Algorithmus von Lohrmann et al. berechnet mit Hilfe der Warteschlangentheorie die Latenz eines Pfades. Diese Berechnung wird für alle Pfade der Topologie ausgeführt. Er vergleicht pro Pfad die berechneten Werte mit dem für den Pfad gegebenen Maximalwert für die Latenz. Der Maximalwert der Latenz des Pfades muss vom Benutzer angegeben werden, bevor der Algorithmus ausgeführt wird. Der Algorithmus versucht den gegebenen Maximalwert für die Latenz des Pfades mit dem geringsten Ressourcenverbrauch zu erreichen. Die Latenz eines Operators wird im Modell des Algorithmus als konstant angenommen. Die Wartezeit verändert sich nach der Formel von Kingman mit der Parallelisierungsgrad des Operators. So stellt die Wartezeit die einzige Möglichkeit dar um die Latenz des Pfades zu anzupassen.

Um minimale Ressourcen zu verbrauchen, startet der Algorithmus bei dem minimalen Parallelisierungsgrad für alle Operatoren. Schrittweise berechnet er dann, bei welchem Operator eine Erhöhung des Parallelisierungsgrades um eins die größte Verringerung der Latenz des Pfades zur Folge hat. Außerdem bestimmt der Algorithmus den Operator, welcher den zweitgrößten Effekt auf die Latenz des Pfades hat. Für den Operator mit dem größten Effekt wird dann durch die von Lohrmann et al. definierte Funktion P_{Δ} ein neuer Parallelisierungsgrad bestimmt. Diese berechnet den neuen Parallelisierungsgrad in Abhängigkeit zum Operator mit dem zweitgrößten Effekt. So soll verhindert werden, dass der Parallelisierungsgrad eines Operators mehrfach hintereinander um eins erhöht wird. Stattdessen wird der Parallelisierungsgrad des gewählten Operators von P_{Δ}

so weit erhöht, dass in der nächsten Runde ein anderer Operator gewählt wird. Diese Schritte werden so lange durchgeführt, bis der Pfad eine Latenz unter dem definierten Maximalwert aufweist.

7.1 Implementation

Im Folgenden sollen die Besonderheiten und Abweichungen vom Original in der vorliegenden Implementation des Algorithmus ausführlich diskutiert werden. Die Implementation des Algorithmus von Lohrmann et al. verwendet das vorgestellte Graph-Modell.

7.1.1 Latenz eines Tupels

Eine weitere Feststellung ist für die Berechnung der Latenz eines einzelnen Tupels notwendig. Laut Lohrmann et al. wird diese von dem Zeitpunkt an dem das Tupel von der Quelle emittiert wird bis zu dem Zeitpunkt an dem das Tupel an dem Konsument aufgenommen wird berechnet. Dabei ist nicht eindeutig definiert, ob die Latenz des Konsument-Operators berücksichtigt wird. In der vorliegenden Implementation wird die Latenz des Konsumentes ebenfalls zur Gesamtlatenz des Tupels gezählt.

7.1.2 Initialisierung

Für die fehlerfreie Berechnung des Parallelisierungsgrades ist es notwendig, dass das Modell so initialisiert ist, dass die zwischengespeicherten Messwerte größer als 0 sind. Ist dies nicht der Fall kann es zu Divisionen durch 0 führen. Daher wird vor der eigentlichen Ausführung des Algorithmus das Modell auf die korrekte Initialisierung geprüft. Falls dies nicht zutrifft, wird eine IllegalStateException geworfen, welche angibt, dass das Modell nicht ordnungsgemäß initialisiert ist.

7.1.3 Minimaler Parallelisierungsgrad und Flaschenhals

Die Wartezeit von Operator i wird von Lohrmann et. al mit der folgenden Funktion berechnet: [LJK15]:

$$W(p_i^*) = e\left(\frac{\lambda_i S_i^2 p_i}{p_i^* - \lambda_i S_i p_i}\right)\left(\frac{c_{Ai}^2 + c_{Si}^2}{2}\right)$$

Der Wert des Koeffizienten e resultiert aus einer Angleichung der Ergebnisse des Modells an die Messwerte des realen CEP-Systems. Er wird im nächsten Kapitel gesondert diskutiert. λ ist die durchschnittliche Tupel-Ankunftsrate pro Millisekunde welche mit

$$\frac{1}{Tupel - Ankunftsintervall}$$

berechnet wird. S beschreibt die Bearbeitungsdauer in Millisekunden. p ist der aktuelle Parallelisierungsgrad des Operators. Die aktuellen Messwerte aus dem CEP-System sind für diesen Parallelisierungsgrad gültig. p^* ist der potentielle neue Parallelisierungsgrad des Operators. Der

Algorithmus versucht die Wartezeit durch das optimieren von p^* so anzupassen, dass die Latenzbeschränkung des Pfades eingehalten wird. Dabei aber möglichst wenig Ressourcen verwendet werden. Der letzte Teil der Formel berechnet einen Koeffizienten aus den Varianzen der Bearbeitungsdauer und der Tupel-Ankunftsrate. Essentiell für die folgenden Ausführungen ist vor allem der Nenner $p_i^* - \lambda_i S_i p_i$.

Ein wichtiges Detail ist, dass ein negativer Nenner für die Funktion nicht sinnvoll ist. Da die anderen beiden Koeffizienten immer positiv sind würde ein negativer Nenner im Ergebnis zu einer negativen Wartezeit führen. Eine negative Wartezeit ist aber real nicht möglich, deswegen ist die Formel für diesen Anwendungsfall ungültig, wenn sie einen negativen Nenner besitzt. Des Weiteren ist die Funktion offensichtlich ungültig wenn nach der Subtraktion im Nenner eine Null steht.

Um diese Probleme zu verhindern legen Lohrmann et al. fest, dass die Berechnungen des Algorithmus nur gültig sind, wenn in der Topologie keine Flaschenhälse vorhanden sind. Ein Flaschenhals wird als ein Operator mit einer Auslastung von 100% oder nahe 100% definiert. Um einen Flaschenhals aufzulösen wird im vorgeschlagenen Algorithmus der Parallelisierungsgrad des Operators verdoppelt. Falls die Auslastung höher als eins ist, wird der Parallelisierungsgrad noch mit der momentanen Auslastung multipliziert. Da der Parallelisierungsgrad eine Ganzzahl sein muss, muss das Produkt zu einem Integer umgewandelt werden. In der vorliegenden Implementation werden die Kommastellen nach der Multiplikation abgeschnitten und nicht gerundet. Bei einer Verdoppelung des Parallelisierungsgrades übt die maximale Differenz von 1 keinen starken Einfluss aus und der Code ist leichter zu lesen. Lohrmann et al. treffen selbst keine Aussage zu dieser Problematik.

Jedoch ist die Methode, Flaschenhälse in der Topologie aufzulösen, nicht ausreichend um die zuvor beschriebenen Probleme zu verhindern. Die Auflösung der Flaschenhälse sorgt nur dafür, dass die Bedingung $0 < \lambda S < 1$ erfüllt ist. Um einen Nenner > 0 zu erhalten muss aber die Bedingung $p^* > \lambda S p$ erfüllt sein. Nehmen wir den Grenzwert 1 für die Auslastung (λS an. Dies bedeutet immer wenn $p \geq p^*$ zutrifft ist der Nenner null oder negativ. Lässt man die Auslastung gegen den Grenzwert 0 gehen, so ist der Nenner unter der Bedingung $p \gg p^*$ null oder negativ. Wie im ersten Teil dieses Kapitels beschrieben, berechnet der Algorithmus zu Beginn die Wartezeit mit dem minimalen Parallelisierungsgrad. Wie in Kapitel ******** beschrieben ist es gewöhnlich, dass dieser eins beträgt. Wir nehmen an der Algorithmus startet mit dem potentiellen Parallelisierungsgrad $p^* = 1$. Der momentan im System aktive Parallelisierungsgrad p bleibt für einen gesamten Durchlauf des Algorithmus konstant. Da $p^* = 1$ ist es nicht unwahrscheinlich, dass $p \gg p^*$ zutrifft und somit der Nenner ≤ 0 ist. Dies ist für die Anwendung aber nicht zulässig und stellt deshalb ein Fehler im von Lohrmann et al. vorgestellten Algorithmus dar. Die Berechnung des Algorithmus darf nicht in allen Fällen mit dem im Modell festegelegten minimalen Parallelisierungsgrad von eins beginnen. Stattdessen ist die Auswahl des minimalen Parallelisierungsgrades eines Operators wie folgt zu definieren: $max(p_min, \lambda S p+1$. Die Erhöhung um eins ist notwendig um zu Verhindern, dass der Nenner null wird. Diese Änderung wurde in der vorliegenden Implementation des Algorithmus umgesetzt.

7.1.4 Ausnahme bei Flaschenhals mit maximalem Parallelisierungsgrad

Wie zuvor beschrieben wird beim Auftreten einen Flaschenhalses der Parallelisierungsgrad des Operators verdoppelt. Dabei kann es vorkommen, dass der maximale Parallelisierungsgrad eines Operators überschritten wird. Die Implementation des Algorithmus wirft in diesen Fall eine Ausnahme, welche besagt dass der Operator trotz maximalem Parallelisierungsgrad ein Flaschenhals ist. Es wird keine weitere Anpassung unternommen.

7.1.5 Ausnahme bei nicht ausreichendem Parallelisierungsgrad

Nachdem die Flaschenhälse aufgelöst wurden, prüft der Algorithmus, ob es mit der momentanen Konfiguration möglich ist den Grenzwert für die Latenz des Pfades zu erreichen, Dazu wird die Latenz für den Fall berechnet, dass alle Operatoren bis zum maximalen Parallelisierungsgrad skaliert sind. Ist dies nicht der Fall, werden in der originalen Version des Algorithmus ohne weitere Mitteilung alle Operatoren maximal skaliert. In der vorliegenden Implementation wurde dieses Verhalten geändert, sodass eine IllegalStateException geworfen wird. Diese sagt aus, dass der maximale Parallelisierungsgrad nicht ausreichend ist. Eine Anpassung des Parallelisierungsgrades findet demnach nicht statt.

7.1.6 Verhindere Loop durch zu kleinen Parallelisierungsgrad

Wenn der Algorithmus schrittweise die Parallelisierungsgrade der Operatoren optimiert, wird die Funktion P_δ aufgerufen. Sie bestimmt den nächsten Parallelisierungsgrad des Operators. Entgegen der Intention der Funktion, dass in der nächsten Runde des Algorithmus ein anderer Operator gewählt wird, liefert Sie teilweise den aktuellen Parallelisierungsgrad des Operators. Der Parallelisierungsgrad des gewählten Operators ändert sich dementsprechend nicht. Dies führt offensichtlich dazu, dass er in der nächsten Runde wieder gewählt wird, da alle Parameter der Warteschlangen-Funktion identisch geblieben sind. Dieses Verhalten führt zu einer endlosen Schleife im Algorithmus. Um das Problem zu beheben wird in der Implementation das Ergebnis der Funktion P_Δ geprüft. Sei p der momentane Parallelisierungsgrad des Operators. Das Ergebnis der Funktion P_Δ ist p_Delta. Dann bestimmt sich der zukünftige Parallelisierungsgrad des Operators aus dem Maximum $max(p+1, p_Delta$.

7.1.7 Parallelisierungsgrad des letzten Operators

In einem Spezialfall des Algorithmus wird die von Lohrmann et al. definierte Funktion P_w verwendet. Sie bestimmt den Parallelisierungsgrad des letzten verbleibenden Operators, wenn alle anderen Operatoren bereits maximal skaliert sind. Allerdings weist diese im von Lohrmann et al. definierten Algorithmus keine Beschränkung durch den maximalen Parallelisierungsgrad des Operators auf. Es treten somit Fälle auf, in denen die Funktion einen Parallelisierungsgrade über dem Maximum zurückliefert. Theoretisch sollte der Maximalwert nicht überschritten werden, denn zu Beginn überprüft der Algorithmus ob der Grenzwert für die Latenz unter maximaler Skalierung erreicht werden kann. Dass dieser Fall dennoch eintritt hängt mit der im nächsten Kapitel beschriebenen Problematik zusammen. Um diesen Fehler zu vermeiden wird in dieser

Implementation das Ergebnis der Funktion geprüft und auf den maximalen Parallelisierungsgrad gesetzt, falls es diesen übersteigt.

7.1.8 Koeffizient e

Im Folgenden wird der Einfluss des Koeffizienten e auf die Funktion P_w untersucht. Die Funktion ist wie folgt definiert:

$$P_w(i, w) = \lceil \frac{a_i}{w} + \lambda_i S_i p_i \rceil$$

$$a_i = \lambda_i S_i^2 p_i \left(\frac{c_{Ai}^2 + c_{Si}^2}{2} \right)$$

Der Parameter i bestimmt dabei den Index des Operators. Der Parameter w ist definiert als die für den Operator maximal erlaubte Wartezeit, um den Grenzwert für die Gesamtlatenz für den Pfad nicht zu überschreiten. Diese ist bekannt, da i der letzte Operator ist, der noch nicht maximal skaliert ist. Außerdem folgt aus der initialen Prüfung, dass die maximale Latenz zumindest bei maximaler Parallelisierung erreicht werden kann.

Die Funktion P_w ist die nach p_i^* umgestellte Variante der Funktion $W(p_i^*$, die die Wartezeit eines Operators abhängig vom Parallelisierungsgrad bestimmt. Die umgestellte Formel bestimmt nun den Parallelisierungsgrad eines Operators abhängig von der maximal erlaubten Warteschlangenzeit w. Allerdings wurde bei der Umstellung der Formel der Koeffizient e nicht berücksichtigt oder aus Pw absichtlich entfernt.

Koeffizient e beschreibt die prozentuale Abweichung der gemessenen Wartezeit zur berechneten Wartezeit aus der Kingman-Formel. Er wird in der Funktion W benutzt um die Abweichung von Modell und realem System auszugleichen. e wird wie folgt berechnet:

$$e_i = \frac{l_{ji} - obl_{ji}}{Kingman_i}$$

wobei l_{ji} die Latenz des Kanals zwischen Operator i und dessen Vorgänger j beschreibt. Die Latenz der Stapelverarbeitung am Ausgang von j wird durch obl_{ji} repräsentiert. Ist die Netzwerklatenz in l_{ji} nicht Berücksichtigt ist die Differenz der beiden Latenzen ist die effektive Wartezeit eines Tupels im realen System.

Durch die fehlende Berücksichtigung von e kann der aus P_w resultierende Parallelisierungsgrad stark von dem in W angenommenen Wert abweichen. Angenommen die maximale Warteschlangendauer $w = 1$ wird in der Funktion W durch den Parallelisierungsgrad $p^* = 1$ erfüllt. Es gilt also $W(1) = 1$. Gleichzeitig nehmen wir an, dass $e = 0,5$ beträgt. Der gemessene Wartezeit beträgt also nur 50% des berechneten Wartezeit $W(1) = 2 * 0,5$. Der Koeffizient passt den berechneten Wert entsprechend an. Berechnet man nun Pw für $w = 1$ wird ein Parallelisierungsgrad von 2 zurückgeliefert, da dieser nicht um den Faktor $e = 0,5$ angeglichen wurde. Der aus Pw resultierende Parallelisierungsgrad verschwendet also Ressourcen wenn angenommen wird, dass $W(p)$ korrekt ist und der Parallelisierungsrad $p = 1$ genügt um $w = 1$ zu erfüllen.

Nehmen wir nun an, dass $p = 1$ der maximale Parallelisierungsgrad eines Operators ist. Um zu prüfen ob der Operator die maximal erlaubte Warteschlangenzeit $w = 1$ erfüllen kann, wird mit

$W(1) = 1$ geprüft ob er mit maximaler Auslastung diesen Wert erreicht . Pw bestimmt nun die tatsächliche Ausprägung des Parallelisierungsgrades und liefert den Wert 2, der den maximalen Parallelisierungsgrad überschreitet.

Drastischer ist das Problem für den Fall, dass der Faktor $e > 1$ ist. Dann würde der gemessene Wert größer als der berechnete Wert sein. Nehmen wir an $e = 1.5, W(3) = 1$ und $W(3) = (2/3) * 1.5$ sowie $w = 1$. Das Ergebnis von W ergibt, dass sich die maximale Warteschlangenzeit durch den Parallelisierungsgrad von 3 erfüllen lässt. Wird nun der Parallelisierungsgrad durch Pw berechnet resultiert daraus 2. Somit würde durch den geringeren Parallelisierungsgrad der Maximalwert für die Latenz des Pfades verletzt.

Um diesem Problem entgegen zu wirken, wird in der Implementation des Algorithmus das Ergebnis von P_w mit dem Koeffizienten e multipliziert.

7.2 Parameter

Dieser Bereich beschreibt die Parameter, mit welchen der Algorithmus gesteuert werden kann. Parameter die in der originalen Version des Algorithmus von Lohrmann et al. fix definiert waren, wurden für die Implementation parametrisiert um flexibler zu sein. Der Algorithmus berücksichtigt die neben den speziell dem Algorithmus zugewiesenen Parametern noch den maximalen und minimalen Parallelisierungsgrad aus dem Graph-Modell. Die folgenden Parameter sind als finale statische Attribute in der Klasse für den Algorithmus zu finden.

7.2.1 Flaschenhals Grenzwert

Name: *BOTTLENECK_THRESHOLD*

Standardwert: *1.0*

Wie von Lohrmann et al. beschrieben ist der Algorithmus ungültig wenn die Topologie einen Flaschenhals aufweist. Ein Operator wird als Flaschenhals definiert, wenn er eine Auslastung von 100% oder nahezu 100% hat. Für die Implementation wurde der Grenzwert von 100% nicht als Konstante sondern als konfigurierbarer Parameter umgesetzt. Der Wert kann als Dezimalzahl angegeben werden, sodass 1.0 = 100% Auslastung entspricht. Ein Wert von größer als 1.0 zu setzen ist für den Algorithmus nicht zulässig. Allerdings könnte es für eine schnellere Skalierung der Operatoren interessant sein, dass der Grenzwert für einen Flaschenhals auf einen niedrigeren Wert gesetzt wird.

Allerdings ist zu beachten, dass der Wert nicht zu niedrig angesetzt sein darf, da sonst immer abwechselnd gegensätzliche Aktionen durch den Algorithmus angestoßen werden. Im ersten Durchlauf erkennt er einen Flaschenhals, zum Beispiel mit einem Grenzwert von 50%. Daraufhin wir der Parallelisierungsgrad des Operators verdoppelt. Beim nächsten Lauf ist der Flaschenhals nicht mehr vorhanden und der Algorithmus verwendet die Funktion der Wartezeit. Er versucht die maximale Latenz des Pfades mit möglichst wenig Ressourcen zu unterschreiten. Deshalb ist es sehr wahrscheinlich, dass er den zuvor verdoppelten Parallelisierungsgrad wieder zurück setzt um Ressourcen zu sparen. Im darauf Folgenden lauf wird dies dazu führen, dass der Operator wieder als Flaschenhals erkannt wird.

7.2.2 Koeffizient für Stapelverarbeitung

Name: *ADAPTIVE_BATCHING_COEFFICIENT*

Standardwert: *0.8*

Wertebereich $0 \leq x < 1$

Die Idee der Stapelverarbeitung ist, dass Tupel am Ende eines Operators gesammelt werden und gleichzeitig über das Netzwerk gesendet werden können. Durch die Zusammenfassung der Tupel kann der Overhead für Netzwerkprotokolle verringert und somit der Durchsatz an Tupel verringert werden. Allerdings steigt durch die Wartezeit im Stapel die Latenz. Lohrmann et al. beschreiben in [LWK14] adaptive Stapelverarbeitung. Sie beschreiben ein System welches die Stapelgröße am Ende eines Operator dynamisch ändert, anstatt wie bei anderen CEP-Systemen, die Größe systemweit zu fixieren. Der Name des Parameters ist darauf zurück zu führen, dass der vorgestellte Algorithmus auf dem CEP-System Nephele aufbaut, für das die adaptive Stapelverarbeitung implementiert wurde. Jedoch ist die Aufgabe des Parameters im Algorithmus generell für jede Stapelverarbeitung zutreffend.

Der Parameter ist ein Koeffizient welcher die Dauer beschreibt, die ein Tupel in dem Ausgangsstapel eines Operators liegt. Er wird als Anteil der maximalen Latenz des Pfades angegeben. Die effektive maximale Latenz eines Pfades berechnet der Algorithmus durch das Produkt aus diesem Parameter und der maximalen Latenz des Pfades.

7.2.3 Schrittweite

Name: *DELTA_STEP_SIZE*

Standardwert: *1*

Wertebereich $1 \leq x$

Dieser Parameter legt die Schrittweite fest, mit der der Algorithmus den Operator bestimmt, der den größten Einfluss auf die Latenz des Pfades hat. In jeder Runde berechnet der Algorithmus die Wartezeit aller Operatoren. Die Berechnung wird mit dem Parallelisierungsgrad , der um die definierte Schrittweite erhöht ist, durchgeführt. Anschließend vergleicht er, welcher Operator die Gesamtlatenz mit dem neuen Parallelisierungsgrad am stärksten verringern würde. Die Schrittweite dient aber ausschließlich zur Auswahl des Operators. Der Parallelisierungsgrad des Operators wird anschließend von der Funktion P_{Δ} festgelegt. Deswegen ist der Standardwert von eins durchaus sinnvoll. Ein höherer Wert wäre interessanter, wenn die Operatoren um die Schrittweite ebenfalls direkt erhöht werden würden. Für die vorliegende Version des Algorithmus führt die Abweichung vom Standardwert sehr wahrscheinlich zu einem höheren Verbrauch an Ressourcen.

7.2.4 Verwendung der Latenz der Kanäle

Name: *USE_LATENCY_ADAPTION*

Standardwert: *true*

Wertebereich *Boolean*

Im einem vorhergehenden Kapitel wurde die Berechnung des Koeffizienten e untersucht. Mit dem Koeffizienten wird versucht die berechnete Wartezeit an die Tatsächliche Wartezeit im System anzupassen. Dazu wird die mit dem aktuellen Parallelisierungsgrad errechnete Wartezeit eines Operators mit der tatsächlich gemessenen Wartezeit ins Verhältnis gesetzt. Das Messen der Wartezeit im realen System ist je nach Support des CEP-Systems nicht trivial. Lohrmann et al. definieren die gemessene Wartezeit als $LatenzdesKanals - LatenzderStapelverarbeitung$. Die Messung der Latenz des Kanals ist jedoch ein nicht triviales Problem, sobald Operatoren sich über mehrere Rechner verteilen. Die Uhren der Hosts exakt synchron zu halten ist nicht realisieren. Somit ist die Messung der Latenz des Kanals nie korrekt.

Ein weiteres Detail ist das Verhältnis der Netzwerklatenz zu der Wartezeit. Wie zu Beginn dieses Kapitels beschrieben, ist die Wartezeit die variable Größe, die durch den Algorithmus optimiert wird. Die Netzwerklatenz wird, wie die Latenz des Operators, als konstant angenommen. Sind Operatoren über mehrere Rechner verteilt ist die Netzwerklatenz in der Latenz des Kanals enthalten. Aus empirischer Erfahrung in der Testumgebung ist die Netzwerklatenz ein essentieller Teil der Latenz des Kanals. Nehmen wir an dass die Netzwerklatenz einen Anteil von 90% an der Gesamtlatenz von 10 ms hat. Somit wäre die gemessene Wartezeit bei 1 ms. Das Ergebnis für die Berechnung der Wartezeit durch die Kingman Formel liefert ebenfalls 1 ms. Sie trifft also die reale Wartezeit des Tupels exakt. Wenn die Latenz der Stapelverarbeitung mit null beziffern wird der Koeffizient e wird nun wie folgt berechnet:

$$e = \frac{LatenzdesKanals}{Kingman} = \frac{10/1}{=}10$$

Dies würde bedeuten dass der Algorithmus die berechnete Wartezeit des Operators verzehnfacht. Anschließend würde er versuchen die verzehnfachte Wartezeit über den Parallelisierungsgrad des Operators zu verringern. Real ist der Anteil der Netzwerk-Latenz aber konstant. Im realen System werden durch die Skalierung somit nur 10% der berechneten Verringerung der Wartezeit effektiv erreicht. Der Algorithmus neigt deshalb dazu sehr hohe Parallelisierungsgrade zurück zu geben, die in der realen Topologie aber nur einen geringen Effekt auf die Latenz des Pfades bewirken.

Daher ist eine Adaption der berechneten Wartezeit durch den Koeffizienten e nur sinnvoll, wenn die effektive Wartezeit im realen System bestimmt werden kann. In der Implementation wurde daher der hier beschriebene Parameter eingeführt, um die Adaption durch den Koeffizienten e abzuschalten zu können. Dieser wird sinnvoller Weise auf *false* gesetzt wenn die Wartezeit im CEP-System nicht bestimmt werden kann.

8 Algorithmus mit Regression

Als weitere Option zum Skalieren der Topologie wurde der Algorithmus von Zacheilas et al. [Zac+15] für das Framework implementiert. Er berechnet, im Gegensatz zum von Lohrmann et al., nicht die gegenwärtige Auslastung des Systems sondern bedient sich der Regression um den zukünftigen Zustand des Systems vorherzusagen. Dies bedeutet konkret, dass der Algorithmus versucht präventiv auf zukünftige Ereignisse zu reagieren, während der der Algorithmus mit Warteschlangen-Theorie reaktiv agiert. Das Ziel des Algorithmus ist die Topologie möglichst vorausschauend anzupassen.

Um den zukünftigen Zustand des Systems vorherzusagen verwenden Zacheilas et al. Gauss-Prozesse. Gauss-Prozesse sind nicht-lineare Regressions-Verfahren und können für mehrdimensionale Regressionen verwendet werden [Ras04]. Im Speziellen werden mit Hilfe des Gauss-Prozesses die verpassten Tupel eines Operators vorhergesagt. Verpasste Tupel sind alle Tupel die über ein Zeitfenster am Operator angefallen aber in diesem nicht abgearbeitet worden sind. Mit Hilfe eines Gauss-Prozesses können die verpassten Tupel mit Abhängigkeit zur Zeit und zum Parallelisierungsgrad des Operators vorhergesagt werden.

Für eine vorausschauende Planung der Topologie, werden Vorhersagen für mehrere in der Zukunft liegende Zeitfenster und für verschiedene Parallelisierungsgrade getroffen. Diese werden wie in Abbildung ++++++++++++++++ zu sehen in einem Graph modelliert. Jeder Knoten v_{kw} stellt dabei eine Vorhersage für ein zukünftiges Zeitfenster w mit einem Parallelisierungsgrad k dar. Der Operator kann zu einem gegebenen Zeitfenster einen beliebigen aber festen Parallelisierungsgrad zwischen minimalem und maximalem Parallelisierungsgrad annehmen. Der Knoten v_{init} stellt den momentan messbaren Zustand des Operators dar. So entsteht ein Graph der die möglichen Zustandsübergänge des Operators zu verschiedenen Zeitpunkten beschreibt.

Um die optimalen Zustandsübergänge zu finden werden nun alle Kanten mit Gewichten versehen. Die Gewichte der Kanten werden mit einer von Zacheilas et al. definierten Kostenfunktion bestimmt. Sie ist eine gewichtete Summe mit drei Summanden und wird durch folgenden Term definiert[Zac+15]:

$$C(v_{ij}v_{i'j'}) = VT_{i'j'} \times C_A + p' \times C_B + S(i,i') \times C_C$$

$$S(i,i') = \begin{cases} 0 & \text{wenn } i = i' \\ 1 & \text{sonst} \end{cases}$$

Der Erste der Summanden ist die Anzahl der verpassten Tupel im zukünftigen Zustand $VT_{i'j'}$, die mit einem benutzerdefinierten Kostenfaktor C_A multipliziert werden. Der zweite Summand erfasst die Kosten der Ressourcen, die benötigt werden um einen Operator auszuführen. Die Kosten

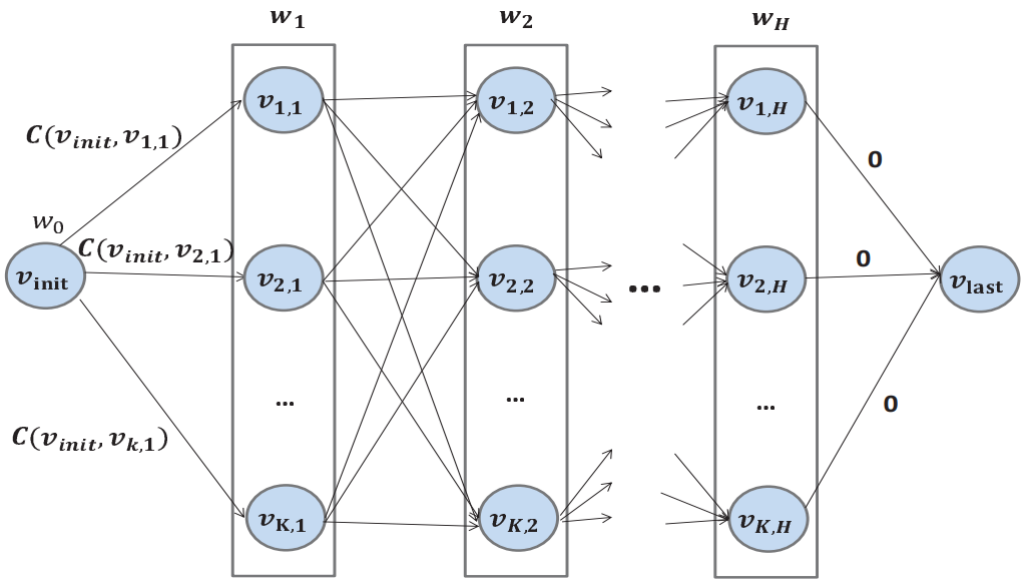

Abbildung 8.1: Graph der Zustandsübergänge [Zac+15].

errechnen aus dem Produkt des zukünftigen Parallelisierungsgrades p' und den benutzerdefinierten Kosten C_B eines einzelnen Tasks. Der dritte Teil der Summe berücksichtigt den Aufwand, der durch skalieren eines Operators entsteht. Das CEP-System muss hierzu die Verarbeitung der Tupel kurzfristig stoppen und das Routing in der Topologie anpassen. Die Kostengewichtung C_C ist ebenfalls durch den Benutzer zu bestimmen. Sind alle Kanten gewichtet kann anschließend mittels eines Algorithmus, der den kürzesten Pfad ermittelt, die optimale Abfolge von Zustandsübergängen berechnet werden. Der erste Zustand in der gefundenen Abfolge bestimmt den Parallelisierungsgrad des Operators.

8.1 Implementation

Im Folgenden sollen die Besonderheiten der Implementierung des Algorithmus für das Framework diskutiert werden. Für die Implementation des Graphen der Zustandsübergänge wurde die Bibliothek JGraphT [Noaf] verwendet. Diese liefert ebenfalls eine Implementation des Dijkstra-Algorithmus um den kürzesten Pfad im Graphen zu bestimmen. Die Umsetzung der Regression mittels Gauss-Prozessen erfolgte mit der Bibliothek Smile [Noae].

8.1.1 Training des Vorhersagemodells

Die Implementation verwendet ebenfalls das Graph-Modell des Frameworks. Allerdings wird für die Vorhersage eine Historie der Messdaten aus dem CEP-System benötigt. Da das Graph-Modell des Frameworks aber nur den aktuellen Status der Topologie repräsentiert, ist dieses nicht ausreichend. Um das Modell für die Regression zu trainieren, werden deshalb CSV-Dateien eingelesen. Das Regressions-Modell wird zu Beginn für alle Operatoren trainiert, sodass während des aktiven Betriebs nur noch das Graph-Modell verwendet wird. Während dem Betrieb können

die Regressions-Modelle der einzelnen Operatoren jeweils einzeln aktualisiert werden. Ddie Laufzeit eines Training-Vorgangs beträgt $O(n^3)$ [Zac+15]. Aufgrund der Laufzeit ist es sinnvoll Operatoren während der Laufzeit einzeln aktualisieren zu können. Bei der Aktualisierung des Modells eines Operators wird das alte Modell verworfen und durch ein neues ersetzt. Das neue Modell wird ausschließlich aus den zuletzt eingelesenen Daten erzeugt.

Außerdem werden in der momentan implementierten Version des Algorithmus andere Eingabewerte als in der Originalversion verwendet. Ursprünglich werden der momentane Zeitstempel, die Tageszeit und der Wochentag für die Vorhersage der eingehenden Tupel verwendet. Um die ausgehenden Tupel vorherzusagen wird zusätzlich der Parallelisierungsgrad des Operators verwendet. Da die Evaluation für Zeiträume in der Größenordnung weniger Stunden vorgesehen ist, ist die Verwendung dieser Werte nicht sinnvoll. Die Tageszeit und der Wochentag wurden daher entfernt und durch die Zeit ersetzt, die seit Beginn der Evaluation vergangen ist.

8.1.2 Vorhersage von verpassten Tupeln

Ein essentieller Teil des Algorithmus ist die Kostenfunktion die auf der korrekten Vorhersage von verpassten Tupeln beruht. Zacheilas et al. berechnen die Tupel wie folgt [Zac+15]:

$$\text{Verpasste Tupel} = \text{Anzahl eingehender Tupel} \times \text{Selektionsrate} - \text{Anzahl ausgegebener Tupel}$$

Die Selektionsrate stellt dabei das Verhältnis der Anzahl ausgegebener Tupel pro eingehendem Tupel dar. Die Selektionsrate eines Operators wird als konstant angenommen. Die Vorhersage von Verpassten Tupeln basiert daher auf der Vorhersage der Anzahl eingehender Tupel und der Anzahl ausgehender Tupel. Ausgehende Tupel werden im von den Autoren vorgestellten System aber nicht gemessen. Sie berechnen die Anzahl der ausgehenden Tupel basierend auf der vergangenen Zeit, dem Parallelisierungsgrad, der vorhergesagten Latenz des Operators und der Anzahl eingehender Tupel. Da das in dieser Arbeit vorgestellte Framework die Daten für ausgehende Tupel liefert wird die Anzahl ausgehender Tupel für die Umsetzung des Algorithmus nicht berechnet. Stattdessen werden anstatt der Latenz des Operators direkt die Anzahl ausgehender Tupel vorhergesagt.

8.1.3 Kern der Gauss-Prozesse

Für die Regression der Anzahl eingehender und ausgehender Tupel werden vom Algorithmus sogenannte Gauss-Prozesse verwendet. Sie ermöglichen zu einem n-dimensionalen Eingabevektor $x \in \mathbb{R}^n$ eine Schätzung des Zielwertes y zu berechnen. Für diese Schätzung bedienen sich Gaußprozesse einer sogenannten Kovarianz-Matrix. Die Kovarianz-Matrix wird mit einer Kovarianz-Funktion aus einem gegebenen Datensatz, den Trainingsdaten, berechnet. Die Trainingsdaten bestehen aus mehreren n-dimensionalen Eingabevektoren x und deren zugehöriger Zielwert y. Die Kovarianz-Funktion $k(x, x'$ bestimmt die Korrelation zwischen zwei Eingabevektoren $x \in \mathbb{R}^n$ aus den Trainingsdaten. Die Wahl der Kovarianz-Funktion hat dabei maßgeblichen Einfluss auf die berechnete Korrelation der Werte und somit auch auf das Modell des Gauss-Prozesses. Eine Kovarianz-Funktion muss per Definition symmetrisch sein [Ras04]. Die Symmetrie der Kovarianz-Funktion bedeutet, dass $k(x, x') = k(x, x')$ gilt. Im Bereich des maschinellen Lernens wird die

Kovarianz-Funktion auch Kern genannt. Somit besitzt jeder Gauss-Prozess einen symmetrischen Kern, mit dessen Hilfe die Kovarianz-Matrix für die Vorhersage berechnet wird.

Zacheilas et al. verwenden für Ihre Vorhersagemodelle folgenden Kern:

$$k(x, x') = \sigma^2 \exp\left(-\frac{1}{2}\sum_{d=1}^{n}\frac{x_d - x'_d}{\lambda_d}\right)$$

Wobei σ und λ hyperparameter des Modells darstellen. Sie werden im Kapitel ********** Parameter betrachtet. Es ist offensichtlich erkennbar, dass der von den Autoren vorgeschlagene Kernel nicht symmetrisch ist. Das Ergebnis der Subtraktion von $x_d - x'_d$ kann sowohl positiv als auch negativ sein. Somit ist der Kern keine gültige Kovarianz-Funktion. Für die vorliegende Implementation wurde der Kern wie folgt geändert, sodass der Ergebnis der Subtraktion als absoluter Betrag verwendet wird.

$$k(x, x') = \sigma^2 \exp\left(-\frac{1}{2}\sum_{d=1}^{n}\frac{|x_d - x'_d|}{\lambda_d}\right)$$

Grundsätzlich ist der Kern in Smile als Java-Interface implementiert und somit durch jede beliebige Kovarianz-Funktion austauschbar.

8.1.4 Pfad-Zurückweisung

Zacheilas et al. sehen in der Originalversion des Algorithmus eine Zurückweisung eines gefundenen kürzesten Pfades vor. Die durch die Verwendung des Gauss-Prozesses vorhergesagten Werte unterliegen einer Normalverteilung. Mit Hilfe der Standardabweichung der Normalverteilung kann berechnet werden, wie wahrscheinliches es ist, dass der vorhergesagte Mittelwert eintrifft [Zac+15]. Diesen Umstand nutzen die Autoren um einen gefundenen Pfad zurückzuweisen, falls der Eintritt des vorhergesagten Wertes zu unwahrscheinlich ist. Diese Funktionalität ist in der momentanen Version der vorliegenden Implementation im Framework nicht vorhanden.

8.2 Parameter

Dieser Abschnitt beschreibt die Parameter die es ermöglichen den Algorithmus anzupassen. In dem Fall des Regression-Algorithmus dienen Sie hauptsächlich dazu das unterliegende Kosten- und Regressionsmodell anzupassen. Wie bei dem Algorithmus mit Warteschlangentheorie werden die im Graph-Modell festgelegten minimalen und maximalen Parallelisierungsgrade berücksichtigt. Die folgenden Parameter sind als finale statische Attribute in den Klassen der Implementation des Algorithmus zu finden.

8.2.1 Trainingsdaten

Name: INPUT_TRAINING_DATA_FOLDER / OUTPUT_TRAINING_DATA_FOLDER

Standardwert: null

Wertebereich: alphanumerisch

Gibt den Pfad zu dem Ordner mit den Dateien an, die die Trainingsdaten für die Regressionsmodelle beinhalten. Die Daten müssen im CSV-Format bereitgestellt werden. Dabei ist zu beachten, dass als Trennzeichen das Leerzeichen erwartet wird. Für jedes Vorhersagemodell wird eine eigene Datei mit Daten benötigt. Es wird somit erwartet, dass in dem angegebenen Verzeichnis zwei Dateien pro Operator vorliegen. Eine Datei beinhaltet die Daten für die Vorhersage der Anzahl eingehender Tupel. Die Andere stellt die Daten für die Vorhersage der Anzahl ausgehender Tupel bereit. Die Datei für eingehende Tupel muss folgenden Namen besitzen: "<Operator>_input_train.csv". Analog gilt für ausgehende Tupel "<Operator>_output_train.csv".

Die Trainingsdaten für eingehende Tupel bestehen momentan aus drei Spalten. Als Daten für den Eingabevektor: Unix-Zeitstempel in Sekunden, Sekunden seit Start der Anwendung. In der letzten Spalte wird die Zielvariable Anzahl eingegangener Tupel erwartet. Die Trainingsdaten für ausgehende Tupel umfassen vier Spalten. Eingabevektor: Unix-Zeitstempel, Sekunden seit Start der Anwendung, Parallelisierungsgrad. Wieder in der letzten Spalte muss die Anzahl ausgehender Tupel stehen. Die Anzahl der Spalten für eingehende und ausgehende Tupel wird über einen Enumerator gesteuert. Dieser muss angepasst werden, falls sich die Spaltenanzahl ändert. Ebenso muss die Dimension von λ mit der Dimension der Eingabewerte übereinstimmen.

8.2.2 Hyperparameter

Tabelle 8.1: Hyperparamter

Name	Standardwert	Wertebereich
SIGMA_INPUT	1.0	$x \in \mathbb{R} > 0$
LAMBDA_INPUT	(1.0, 1.0)	$x \in \mathbb{R}^2 > 0$
SIGMA_OUTPUT	1.0	$x \in \mathbb{R}0$
LAMBDA_OUTPUT	(1.0, 1.0, 1.0)	$x \in \mathbb{R}^3 0$

Die Hyperparameter dienen hauptsächlich der Konfiguration des Kernels. Sigma ist eine Größe mit der der Kernel multipliziert wird. Der Parameter kann dazu verwendet werden die Korrelation der Werte grundsätzlich zu verkleinern oder zu vergrößern. Lamda wird verwendet um die Komponenten des Eingabevektors zu gewichten. Für den verwendeten Kernel bedeutet das, dass der negative Exponent kleiner wird wenn $lambda$ größer wird. Im Umkehrschluss fällt die Korrelation für diesen Wert des Eingabevektors höher aus.

Der Wert für den Zeitstempel verdient besondere Beachtung. Da der Zeitstempel einen absoluten Zeitpunkt markiert, verändert sich die Korrelation mit jeder Anwendung des Kernels. Hier muss das $lambda$ gegebenenfalls nachjustiert werden. Das $lambda$ für die Vorhersage der ausgehenden Tupel sollte zusätzlich berücksichtigen, dass der Parallelisierungsgrad im Normalfall eine kleinere Größenordnung als die anderen Werte besitzt. Daher weist der Parallelisierungsgrad tendenziell

höhere Korrelationen auf als die anderen Komponenten. Dieser Umstand sollte gegebenenfalls mit $lambda$ ausgeglichen werden.

8.2.3 Kostenfunktion

Die vorgestellte Kostenfunktion bietet drei verschiedene Eingabegrößen, die durch den Benutzer getätigt werden müssen. Die Kostenfunktion implementiert ein Interface und wird bei der Erzeugung der Regressionsmodelle als Parameter übergeben Somit ist die Kostenfunktion gegen andere Varianten austauschbar. Die Kostenfaktoren der Funktion können über den Konstruktor angegeben werden. Bei einem parameterlosen Aufruf des Konstruktors initialisiert dieser die Funktion mit eins.

9 Evaluation

Im diesem letzten Teil der Arbeit sollen die beiden Implementationen der Algorithmen evaluiert und miteinander verglichen werden. Für die Evaluation sollen die Algorithmen das CEP-System Heron über das implementierte Framework kontrollieren, während ein möglichst realitätsnaher Strom von Tupeln vom System abgearbeitet werden muss.

9.1 Testdaten

Um einen realitätsnahen Datenstrom erzeugen zu können, wurden während dem Zeitraum August 2018 Tweets von der Twitter-Streaming API abgerufen und gespeichert. Neben kostenpflichtigen Schnittstellen bietet die API kostenlos einen kontinuierlichen Datenstrom von zufällig gewählten Tweets. Die gewählten Tweets sind dabei für jeden Konsumenten, der sich für den Datenstrom registriert hat, identisch [Noaa]. Der Datenstrom liefert ca. 1% der gesamten anfallenden Twitter-Daten in Realzeit [Noad]. Dies sind neben neuen Statusupdates (Tweets) auch Löschanfragen zu bestehenden Tweets. Aus Gründen der besseren Lesbarkeit werden im Folgenden beide Arten der Statusänderung als Tupel bezeichnet. Die für die Evaluation werden die gesammelten Tupel der Woche vom 7. - 13. August 2018 verwendet. In dieser Zeit wurden 32.416.487 Tupel erfasst die ca. 160 GB Textdaten entsprechen Die Abbildung ********** zeigt die Verteilung der Anzahl erfasster Statusmeldungen pro Stunde im Verlauf der Woche. Beginn der Aufzeichnung ist Dienstag, 07.08.2018 00:00 UTC.

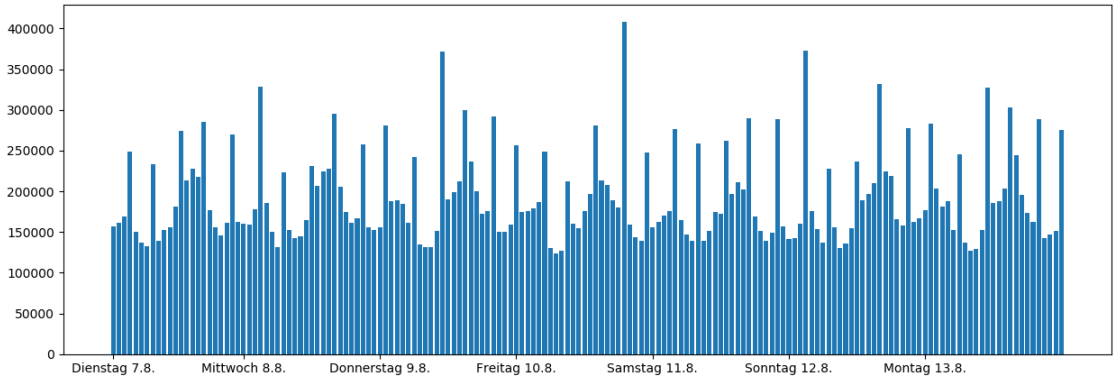

Abbildung 9.1: Anzahl Tupel pro Stunde.

9.2 Systemaufbau

Die Evaluation wurde auf dem internen OpenStack-Cluster des Instituts für Verteile und Parallele Systeme der Universität Stuttgart durchgeführt. Für die Evaluation wurde ein Cluster aus vier virtuellen Maschinen aufgebaut. Die Maschinen werden mit dem Betriebssystem Ubuntu 14.04 betrieben. Drei der virtuellen Maschinen sind je mit 24 CPU-Kernen, 32 GB Arbeitsspeicher und 50 GB persistentem Speicher ausgestattet. Diese Maschinen führen während der Evaluation die aktiven Tasks aus. Die vierte Maschine wird als Steuer-Knoten genutzt und arbeitet mit 4 CPU-Kernen, 8 GB RAM sowie 10 GB persistentem Speicher. Auf der vierten Maschine wird ausschließlich der Ablauf der Evaluation gesteuert. Somit sind die Maschinen, die Tasks der Topologie ausführen, identisch. Der Cluster wird über den Apache Aurora-Scheduler in der Version 0.13 gesteuert. Der Scheduler von Aurora so wie die Steuerung des CEP befinden sich auf diesem Knoten. Auf jedem der drei anderen Rechner ist die ausführende Instanz von Aurora ebenfalls in der Version 0.13 installiert. Um die Arbeitspakete an die ausführenden Knoten auszuliefern, wird das verteile Dateisystem HDFS von Apache Hadoop in der Version 3.0.3 verwendet. Der Adapter für das implementierte Framework wird ebenfalls auf diesem Rechner gestartet. Zusätzlich benötigen die Systeme Apache Zookeeper um den verteilten Zustand zu speichern. Dieser ist ebenfalls als Cluster über alle vier virtuellen Maschinen konfiguriert.

9.3 Heron

Für die Evaluation wurde das CEP-System Heron gewählt [Kul+15]. Heron ist ein Nachfolger von Apache Storm. Die Twitter Inc. entwickelte Heron als Nachfolger für den produktiv verwendeten Storm Cluster. Sie entwickelten Heron als schnelleres und besser skalierendes System, da Storm den Anforderungen nicht mehr gerecht werden konnte [Kul+15]. Heron wird von Twitter momentan als produktives System für die Analyse von Datenströmen eingesetzt. Heron wurde für diese Evaluation aus zwei Gründen gewählt. Einerseits ist aufgrund der produktiven Verwendung und aktiven Entwicklung bei Twitter ist davon auszugehen, dass das System aktuellen realen Anforderungen gerecht wird. Zweitens wurde ein System gesucht, mit dem die Algorithmen möglichst unabhängig bewertet werden können. Viele der in Kapitel ******** vorgestellten Algorithmen wurden auf Basis eines bestimmten CEP-Systems entwickelt. Einige dieser CEP-Systeme sind Eigenentwicklungen oder erweiterte Frameworks auf bestehenden CEP-Systemen, die von den Autoren entwickelt werden. Apache Storm wird ebenso, als eines der bekanntesten CEP-Systeme, oft als Basis für die Entwicklung von neuen Algorithmen verwendet. Mit Heron können verschiedene Algorithmen getestet werden, ohne dass manche den Vorteil besitzen nativ für das CEP-Systems entwickelt worden zu sein.

Eine wichtige Neuerung bei Heron ist besonders hervorzuheben: Das System bietet die Möglichkeit eine Topologie über eine autonome Strom-Regulation zu steuern [Kul+15]. Bei vielen anderen Systemen werden Tupel einfach verworfen, wenn die Kapazität eines Operators nicht mehr ausreicht. Dieses Verhalten wird als sogenannter Lastabwurf bezeichnet. Dies führt dazu, dass Tupel entweder verloren gehen oder von der Quelle erneut ausgegeben werden. Wird ein Tupel neu ausgegeben, muss es alle Operatoren erneut durchqueren. Diese Verhalten zusätzliche Last auf dem System verursachen und den Engpass noch verstärken. Die neue Regulation in Heron wird aktiviert, wenn ein Operator die ankommende Menge von Tupel nicht mehr verarbeiten

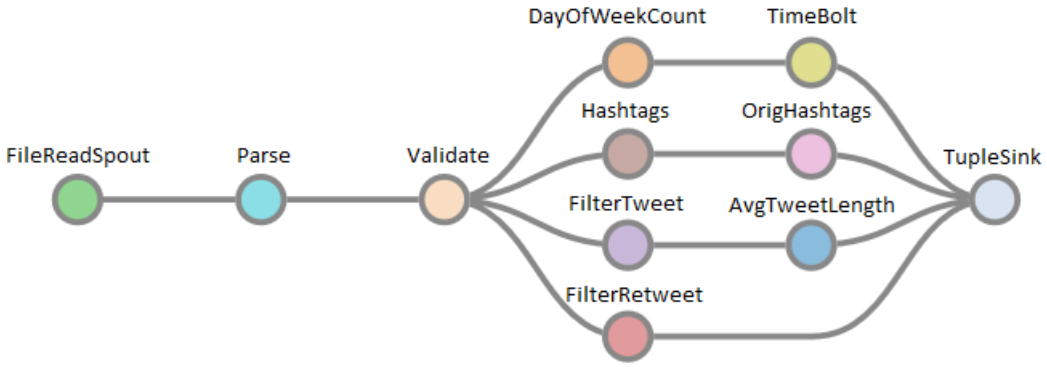

Abbildung 9.2: Darstellung der logischen Topologie durch Heron UI.

kann. Dies wird von der Topologie erkannt und die Quellen werden daraufhin gestoppt. Somit gelangen keine neuen Tupel mehr in die Topologie. Der Stopp wird so lange aufrecht erhalten, bis der Operator wieder genügend Kapazität für neue Tupel geschaffen hat.

Diese Methode, den Datenstrom zu drosseln, verändert die Erkennung von Operatoren die Flaschenhälse bilden gänzlich. Messwerte, auf die viele der in Kapitel 4 vorgestellten Algorithmen reagieren, verhalten sich anders. Dies trifft auch für die beiden implementierten Algorithmen zu. Diese gehen im Grundprinzip davon aus, dass bei einem Flaschenhals die Anzahl ankommender Tupel größer als die Anzahl abgearbeiteter Tupel ist. Dieser Zustand kann aber nicht mehr auftreten, wenn die Quelle den Zufluss von neuen Tupeln automatisch stoppt. Somit würde ein Operator gar nicht oder nur sehr schwach von den Algorithmen skaliert werden. Deswegen wurde dieses Verhalten für die Evaluation der beiden Implementationen deaktiviert.

Die Evaluation wurde mit Heron unter der aktuellen Version 0.17.5 durchgeführt. Die System-Parameter wurden im Standard belassen und nicht verändert.

9.4 Topologie

Für die Durchführung der Evaluation wurde eine eigene Topologie entwickelt. Diese liest die von der Twitter API gesammelten Tupel aus und analysiert sie auf deren Eigenschaften. Der logische Aufbau der Topologie ist in der Abbildung 10.2 dargestellt. Ziel bei der Erstellung der Topologie war, dass es Operatoren mit unterschiedlich großen Lasten gibt. Entweder wurde der Datenstrom durch eine Filteroperation für den Folgeoperator verkleinert oder die Operation des Operators rechenintensiv gestaltet.

Topologien in Heron besitzen Parameter, durch die der Entwickler das Verhalten beeinflussen kann. Für nahezu alle Parameter definiert Heron einen Standardwert. Im Folgenden werden die Parameter beschrieben, bei der die Evaluations-Topologie vom Standard abweicht. Die Parameter sind in Tabelle 10.1 mit den zugewiesenen Werten aufgelistet.

Wie schon im vorherigen Kapitel beschrieben wurde die Regulation des Tupel-Stroms deaktiviert. Die Topologie wurde so konfiguriert, dass jedes Tupel mindestens einmal erfolgreich verarbeitet

Tabelle 9.1: Parameter der Topologie

Parameter	Wert
TopologyDropTuplesUponBackpressure	True
TopologyReliabilityMode	ATLEAST_ONCE
MessageTimeoutSecs	60
MaxSpoutPending	1000000

werden muss. Deshalb werden Tupel, die in der Folge eines Flaschenhalses verworfen werden, mehrfach von der Quelle ausgegeben. Dies ist möglich, weil die Bestätigung von Tupel aktiviert wurde. Dies bedeutet, dass jeder Operator die Bearbeitung des Tupels an die Quelle zurückmeldet. Erst wenn alle Operatoren die Verarbeitung des Tupel bestätigen, wird das Tupel als bestätigt markiert. Durch diesen Mechanismus ist es ebenfalls möglich die Latenz des Tupels zu bestimmen. Die Quelle wurde so konfiguriert, dass ein Tupel, das innerhalb von einer Minute nicht erfolgreich verarbeitet wurde, ungültig ist und wiederholt werden muss. Außerdem sendet die Quelle maximal 1.000.000 Tupel, ohne dass eines davon bestätigt wurde. Das bedeutet, dass sich zu jedem Zeitpunkt maximal 1.000.000 Tupel zur Bearbeitung in der Topologie befinden. Dieser Wert wurde bewusst sehr groß gewählt, um die Ausgabe von Tupeln nicht aufgrund von Flaschenhälsen in der Topologie zu verringern. So ist sichergestellt, dass die Quelle dauerhaft in der Lage ist Tupel auszugeben, um den realistischen Arbeitsaufwand zu rekonstruieren.

In der momentanen Version kann Heron die Operatoren nur skalieren, wenn die Tupel zufällig den Tasks der Operatoren zugewiesen werden. Deswegen wurde diese Konfiguration für die Operatoren der Topologie gewählt. Außerdem erhält jeder Folgeoperator immer alle ausgegebenen Tupel des Vorgängers. Es gibt also keine selektive Weiterleitung. Jedem Task wurde eine CPU, 1,25 GB RAM und 1,25 GB persistentem Speicher zugewiesen. Die Topologie wurde mit der Java API von Heron entwickelt. Diese erlaubt das definieren eigener Logik für die Quelle und die anderen Operatoren. Im Folgenden werden die einzelnen Operatoren der Topologie beschrieben.

9.4.1 Quelle

Die Quelle ”FileReadSpout” liest die gesammelten Daten von der Festplatte. Dies geschieht Zeile für Zeile was jeweils einem Tupel entspricht. Jedes erfasste Tupel besitzt einen Zeitstempel mit dem Zeitpunkt, an dem es erzeugt wurde. Um den Twitter-Datenstrom realistisch nachzubilden wird dabei immer der Zeitstempel des eingelesenen Tupels überprüft. Um die Topologie auszulasten ist der Datenstrom in der realen Geschwindigkeit, mit der er erfasst wurde, nicht ausreichend. Deshalb wird die Zeit in der die Tupel verarbeitet wurden, auf eine kleineren Zeitraum komprimiert. So bleibt die Struktur des Arbeitsaufwandes wie Lastspitzen und Lastabfall erhalten, wird aber in ein kürzeres Intervall geschoben um so insgesamt mehr Last zu erzeugen. Das Ziel ist, dass in jeder realen Minute in der Evaluation 84 Minuten der aufgezeichneten Daten verarbeitet werden. Somit werden in einer realen Stunde 84 Stunden der aufgezeichneten Daten. Da eine Woche 168 Stunden hat dauert ein Durchlauf der Evaluation 2 Stunden. Die Quelle wurde so implementiert, dass sie den Startzeitpunkt der Topologie sowie den Zeitstempel des ersten Tupels in Millisekunden speichert. Während der Evaluation wird geprüft wie viele Millisekunden seit dem Start der Topologie vergangen sind. Mit diesen Werten kann der maximale Zeitstempel errechnet werden, mit dem ein Tupel die Quelle verlassen darf. Der maximale Zeitstempel errechnet sich

aus der Differenz von aktuellem Zeitstempel und der Startzeit der Topologie die mit dem Faktor 84 multipliziert und zu dem Zeitstempel des ersten Tupels addiert wird. Mit dieser Methode kann nicht garantiert werden, dass Tupel nicht langsamer ausgegeben werden, aber es wird verhindert, dass die Tupel schneller als der realistische Arbeitsaufwand ausgeben werden. Die Leserate vom permanenten Speicher ist in der verwendeten Implementation wesentlich höher als die Rate des realistischen Arbeitsaufwandes, sodass eine langsamere Abgabe der Tupel unrealistisch ist.

9.4.2 Andere Operatoren

Um die folgenden Beschreibungen zu verstehen, sind folgende Erläuterungen notwendig. Ein Retweet ist ein Statusupdate welches das Statusupdate eines anderen Nutzers ohne weitere Anmerkung weiterverbreitet. Ein Antwort-Tweet ist ein Statusupdate, welches auf das Statusupdate eines anderen Nutzers antwortet. Ein Tweet zitiert einen anderen Tweet wenn er ihn weiterverbreitet und Anmerkungen hinzufügt. Außerdem ist zu bemerken, dass alle Operatoren Tupel direkt weiterverarbeiten und keine Fenster bilden.

- ”Parse”: Dieser Operator liest das Tupel, das einen String, der ein JSON-Objekt beschreibt, und liest alle Daten des JSON Objektes aus. Anschließend werden ausgewählte Daten weiter gesendet. Löschanfragen für Statusmeldungen werden nicht weitergeleitet.
- ”Validate”: Dieser Operator prüft die Lesbarkeit der Daten und gibt sie auf der Standardausgabe aus.
- ”DayOfWeekCount”: Hier wird der Zeitstempel des Tupels in den Wochentag, an dem das Tupel erzeugt wurde, umgerechnet. Jedes Tupel erhöht den Zähler für den jeweiligen Tag.
- ”TimeBolt”: Dieser Operator gibt den genauen Datumstext für den Zeitstempel des Tupels auf der Standardausgabe aus.
- ”Hashtags”: Hier werden die Hashtags des Tweets ausgelesen. In einer Tabelle werden identische Hashtags gezählt.
- ”OrigHashtags”: Zählt ebenso Hashtags, allerdings nur für Original-Tweets eines Retweet oder Antwort-Tweet.
- ”FilterTweet”: Sendet nur Tweets weiter, die kein Retweet und kein Antwort-Tweet sind sowie keinen anderen Tweet zitieren.
- ”AvgTweetLength”: Berechnet die durchschnittliche Länge des verfassten Textes.
- ”FilterRetweet”: Sendet ausschließlich Retweets weiter.
- ”TupleSink”: Hat keine Operation außer die Tupel final zu bestätigen.

9.5 Ablauf

In der Evaluation mussten die zuvor implementierten Algorithmen die Topologie steuern, während Sie die Tupel im Zeitfenster von zwei Stunden verarbeitet. Alle Operatoren starten dabei mit einem Parallelisierungsgrad von eins. Für jeden der beiden Algorithmen von Lohrmann et al. und Zacheilas et al. wurden fünf Durchläufe gestartet. Jeder Durchlauf startet mit dem aktivieren des Clusters. Die Hauptsteuerung wurde so implementiert, dass sie alle fünf Minuten die Topologie durch den gewählten Algorithmus prüfen lässt. Zuerst werden alle Messwerte des Graph-Modells mit den aktuellen Messwerten aus der Topologie aktualisiert. Anschließend wurden die folgenden Daten zur Evaluation erfasst:

- Minuten seit Beginn der Evaluation
- Sekunden seit Beginn der Evaluation
- Durchschnittliche Latenz der Tupel
- Anzahl fehlgeschlagener Tupel
- Anzahl bestätigter Tupel
- Summe der Parallelisierungsgrade der Operatoren

Die Anzahl der Minuten und Sekunden seit Beginn der Evaluation ist dabei um fünf Minuten verschoben, da der Erste Start der Hauptsteuerung erst zur ersten Prüfung erfolgt, nachdem die Topologie schon 5 Minuten aktiv ist. Wenn alle Messwerte erfasst sind, startet der gewählte Algorithmus und die Topologie wird entsprechend der Empfehlung des Algorithmus skaliert. Erst nachdem der Vorgang erfolgreich von Heron zurückmeldet wurde, beginnt das Zeitfenster von 5 Minuten bis zur nächsten Prüfung.

Während Heron die Topologie skaliert ist diese gestoppt, sodass während dieser Zeit keine neuen Tupel von der Quelle ausgegeben werden. Dies wirkt sich offensichtlich auf die Zeitbeschränkung für die Tupel-Ausgabe der Quelle aus, da die Zeit, die seit Beginn der Evaluation vergangen ist, trotzdem weiter läuft. Dies spiegelt auch das reale Verhalten wieder, da der reale Datenstrom ebenfalls zwischengespeichert werden müsste, solange die Topologie skaliert. Während des Vorgangs löscht Heron außerdem alle Messdaten, da die alten Daten für den aktuellen Zustand der Topologie nicht mehr gültig sind. Nachdem die Topologie wieder gestartet wurde, müssen sich die Zwischenspeicher vor und nach den Operatoren erst wieder mit Tupel füllen, um aussagekräftige Messwerte zu erhalten. Deswegen ist es sinnvoll erst nach Abschluss des Vorgangs weitere fünf Minuten zu warten, bis die Messwerte im System wieder aussagekräftig sind. Außerdem werden aus diesem Grund bei der Abfrage der Messwerte, die ebenfalls im Interval von fünf Minuten ausgelesen werden, nur die letzten drei Minuten aus dem CEP-System angefordert. Dies bedeutet, dass die Messwerte der ersten zwei Minuten, nachdem die Topologie skaliert wurde, weder für die Berechnung des Parallelisierungsgrades noch für die Ergebnisse der Evaluation verwendet wurden. Die Evaluation stoppt sobald alle Tupel verarbeitet wurden.

Tabelle 9.2: Parameter für den Warteschlangenalgorithmus

Parameter	Wert
BOTTLENECK_THRESHOLD	1.0
ADAPTIVE_BATCHING_COEFFICIENT	0.0
DELTA_STEP_SIZE	1
USE_LATENCY_ADAPTION	False

9.6 Parametrisierung der Algorithmen

Für die Durchführung der Evaluation mussten die beschriebenen Parameter der Algorithmen definiert werden. Das GraphModell war so konfiguriert, dass der minimale Parallelisierungsgrad eins und der maximale zehn beträgt. Die maximale Latenz des Pfades wurde auf 10 ms gesetzt. Da jeder Task eines Operators eine CPU, und 1,25 RAM verbraucht, konnte der maximale Parallelisierungsgrad nicht höher als 10 gesetzt werden.

Für die Evaluation wurden für den Warteschlangen-Algorithmus die in der Tabelle 10.3 gelisteten Parameter gewählt. Wie im Original von Lohrmann et al. wurde die Erkennung eines Flaschenhalses ab einer Auslastung von 100% gesetzt. Der Koeffizient für adaptives Batching wurde auf null gesetzt, da Heron diese Methode nicht verwendet. Der Parameter bestimmt ohnehin nur den Anteil der maximalen Latenz des Pfades, der für adaptives Batching reserviert ist. So kann dieser Anteil auch direkt bei der Angabe der maximalen Latenz berücksichtigt werden, wenn der Parameter auf null steht. Die Schrittweite des Algorithmus wurde ebenfalls wie im Original auf eins gesetzt. Die Verwendung des Koeffizienten e wurde deaktiviert, da die gemessene Latenz im Vergleich zu der vom Algorithmus berechneten Latenz sehr hoch war. Dies hätte wie in Kapitel 8.2.4 beschrieben dazu geführt, dass der Algorithmus sehr hohe Parallelisierungsgrade vorgeschlagen hätte. Dies hätte in Kombination mit dem maximalen Parallelisierungsgrad von zehn zur Folge gehabt, dass die Topologie während der Evaluation konstant maximal skaliert ist und nie angepasst wird.

Für den Algorithmus, der Regression zur Vorhersage verwendet wurden die in Tabelle 10.4 gezeigte Parametrisierung gewählt. Die Trainingdaten für den Algorithmus wurden während der Evaluation des Lohrmann Algorithmus gesammelt. Alle fünf Minuten, in denen die neuen Metriken aus dem System abgerufen wurden, wurden für jeden Operator die Anzahl eingegangener und ausgegangener Tupel der letzten drei Minuten mit Zeitstempel versehen und gespeichert. Da die aus der Topologie bezogenen Messdaten vom aktuellen Zustand der Topologie und nicht von den verwendeten Twitter-Daten abhängen, sind auch Fehler reflektiert, die vom Algorithmus von Lohrmann et al. gemacht werden. Somit ist garantiert, dass die verwendeten Trainingsdaten das Vorhersage-Modell nicht speziell für den vorliegenden Twitter-Datensatz übertrainiert wird.

Die Hyperparameter wurden für die Evaluation nicht mathematisch optimiert. Sie wurden lediglich so gewählt, dass die Regression Vorhersagen im richtigen Größenbereich trifft. Dazu wurden die Zeitstempel sehr schwach gewichtet. Die Zeit, die seit der Evaluation vergangen ist, wurde stark gewichtet, da sie einen größeren Zusammenhang mit dem Zustand der Topolgie aufweiste als der Zeitstempel. Da das Modell für eingehende Tupel nur auf den beiden Dimensionen Zeitstempel und vergangene Zeit beruht, muss die vergangene Zeit stark gewichtet werden um zu verhindern, dass der Zeitstempel das Modell verfälscht. Für die Vorhersage der ausgehenden

Tabelle 9.3: Parameter für den Algorithmus mit Regression

Parameter	Wert
SIGMA_INPUT	1.0
LAMBDA_INPUT	(1.0, 10000.0)
SIGMA_OUTPUT	1.0
LAMBDA_OUTPUT	(1.0, 10000.0 10.0)
Kostenfunktion	(1.0, 1.0, 1.0)
Abstand Zeitfenster	300
Anzahl Zeitfenster	6

Tupel wurde die Gewichtung des Parallelisierungsgrad so gewählt, dass er sich in etwa gleich wie die vergangene Zeit auf die Korrelation auswirkt. Wie in Kapitel 9.2.2 erwähnt besitzen der Parallelisierungsgrad und die Zeit, die seit der Evaluation vergangen ist, verschiedene Größenordnungen. In der Evaluation bewegen sich Parallelisierungsgrade im Einerbereich währen die Zeit bis zu 7200 steigt. Diese Tatsache wird durch den gewählten Wert des Hyperparameters ausgeglichen.

Die Kostenfunktion wurde für alle Kostenfaktoren mit eins initialisiert. Dies fördert die Flexibilität der Topologie, da die Kosten, die Topologie zu ändern, und die Kosten pro verwendeten Task sehr gering sind. Die Anzahl verpasster Tupel kann jedoch schnell 1000 übersteigen, sodass die Kosten für verpasste Tupel ausschlaggebend für den kürzesten Pfad im Graphen der Zustandsübergänge sind. Die Vorhersagen wurden für Zeitfenster von fünf Minuten getroffen, um dem gleich dem Intervall zu sein, in dem die Topologie geprüft wird. Es wurden jede Runde Vorhersagen für sechs Zeitfenster getroffen, sodass immer Zustandsübergänge für die nächsten 30 Minuten berücksichtigt wurden.

9.7 Ergebnisse

Im folgenden werden für die Evaluation der beiden Algorithmen die Messwerte verwendet, die jeweils vor der Prüfung durch den Algorithmus erfasst wurden. Diese Reflektieren kein festes Intervall von fünf Minuten, da die fünf Minuten jeweils erst nach der Bestätigung, dass erfolgreich skaliert wurde, beginnen. Außerdem ist zu beachten, dass die gezeigten Auswertungen jeweils zur Minute fünf der Evaluation beginnen, da zu Beginn der Evaluation offensichtlich keine Messwerte vorliegen. Im folgenden Werden folgende Abkürzungen für die Algorithmen verwendet um die Lesbarkeit zu bessern: Algorithmus mit Warteschlangen Theorie (WT), Algorithmus mit Regression (RG).

9.7.1 Parallelisierungsgrad der Topologie

Um die weiteren Messwerte zu verstehen ist es essentiell zu betrachten in welchen Zustand die Topologie während des Zeitraums befand, in dem die Werte erfasst wurden. Deshalb wurde jeweils der summierte Parallelisierungsgrad der Topologie gemessen. Wie in Abbildung 10.3 zu sehen ist, verlaufen die Parallelisierungsgrade unter den beiden Algorithmen sehr unterschiedlich.

Zu Beginn skalieren beide Algorithmen die Topologie nach oben. Es ist ersichtlich, dass WT den Parallelisierungsgrad der Operatoren schrittweise verdoppelt, während RG etwas schneller ansteigt. Nach dem initialen Anstieg schwankt der Parallelisierungsgrad unter WT sehr stark. WT nimmt über die Evaluation hinweg viele Anpassungen vor und tendiert dazu, stark zwischen hohen und niederen Werten zu schwanken, da er nur reaktiv auf die letzten drei Minuten reagiert. Bei hohem Parallelisierungsgrad geht dabei die Latenz der Tupel stark zurück, sodass der Algorithmus ihn verringert. Ist der Parallelisierungsgrad anschließend niedrig steigt die Latenz und der Algorithmus schlägt wieder hoch skaliert. So schwankt die Topologie stark zwischen zwei extremen. Den Parallelisierungsgrad zu ändern stoppt ebenfalls den Zufluss von Tupeln, sodass diese anschließend in größeren Mengen in die Topologie fließen, was diesen Effekt noch verstärkt.

RG hingegen bewegt sich jedoch sehr konstant und ähnelt der gezeigten Regressionskurve von WT. Die Ähnlichkeit kann mit der Methode der Vorhersage, die ebenfalls eine Regression ist, zusammenhängen. Obwohl hier ausgehende und eingehende Tupel vorhergesagt werden, hängen diese vom Parallelisierungsgrad ab. Im Modell von Zacheilas et al. ist nicht explitzit berücksichtigt, dass die Anzahl der eingehenden Tupel ansteigt, sobald Vorgängeroperatoren skaliert werden. In den Trainigsdaten für die Regression ist dieser Umstand jedoch abgebildet, sodass der Parallelisierungsrad zum Messzeitpunkt der Trainigsdaten Einfluss auf beide Regressions-Modelle auswirkt. Da die Trainingsdaten immer mit einem bestimmen Parallelisierungsgrad gemessen werden, fällt es RG schwer die Operatoren auf Parallelisierungsgrade außerhalb der erfassten Trainingsdaten zu skalieren. Für diese Parallelisierungsgrade sind keine Informationen über eingehende und ausgehende Tupel vorhanden und das Regressions-Modell nähert sich eher an bekannte Parallelisierungsgrade an. Für bessere Vorhersagen ist es daher essentiell, dass Messungen mit allen Möglichen Parallelisierungsgraden der Topologie in die Trainingsdaten mit einfließen.

Wie erwartet, hält der RG Algorithmus den Parallelisierungsgrad konstanter als der reaktive Ansatz von WT. Einerseits ist die Vorhersagemethode verantwortlich, da dieses einen Mittelwert aus den Trainingsdaten bildet und der Mittelwert konstanter verläuft als die jeweils aktuellen Messdaten, die WT zugrunde liegen. Außerdem wird bei RG der Zustand der Topologie jeweils für eine halbe Stunde voraus geplant was den Effekt verstärkt. So zeigt sich fast keine Schwankung, obwohl die verwendete Kostenfunktion eine Änderung der Topologie gleich gewichtet wie das verpassen eines einzelnen Tupels. Außerdem ist noch zu erwähnen dass der maximale Parallelisierungsgrad von 10 Tasks per Operator nie benötigt wurde. Somit war dieser Maximalwert kein Flaschenhals für die Topologie.

9.7.2 Latenz der Tupel

Wenn man den Verlauf der Latenzen der Tupel betrachtet, kann man feststellen, dass diese sich gegenläufig zum Verlauf des Parallelisierungsgrad verhalten. Auffällig ist, dass die Messwerte von WT im Schnitt deutlich niederer sind, als die Messwerte von RG. Dies ist durch mehrere Ursachen zu erklären. Zum einen wird die Topologie wesentlich häufiger durch WT angepasst, als es bei RG der Fall ist. Wir die Topologie wieder gestartet ist der Fluss der Tupel ungehemmter, als wenn sie konstant läuft. Dieser Effekt wurde versucht damit abzufangen, dass immer nur die letzten drei Minuten des fünf Minuten Intervalls gemessen wurden. Trotz kann dies Einfluss auf die Latenzen auswirken, da die Topologie eventuell länger als zwei Minuten benötigt bis sie einen stabilen Zustand erreicht. Des weiteren ist WT speziell dafür entwickelt die Latenz der Tupel

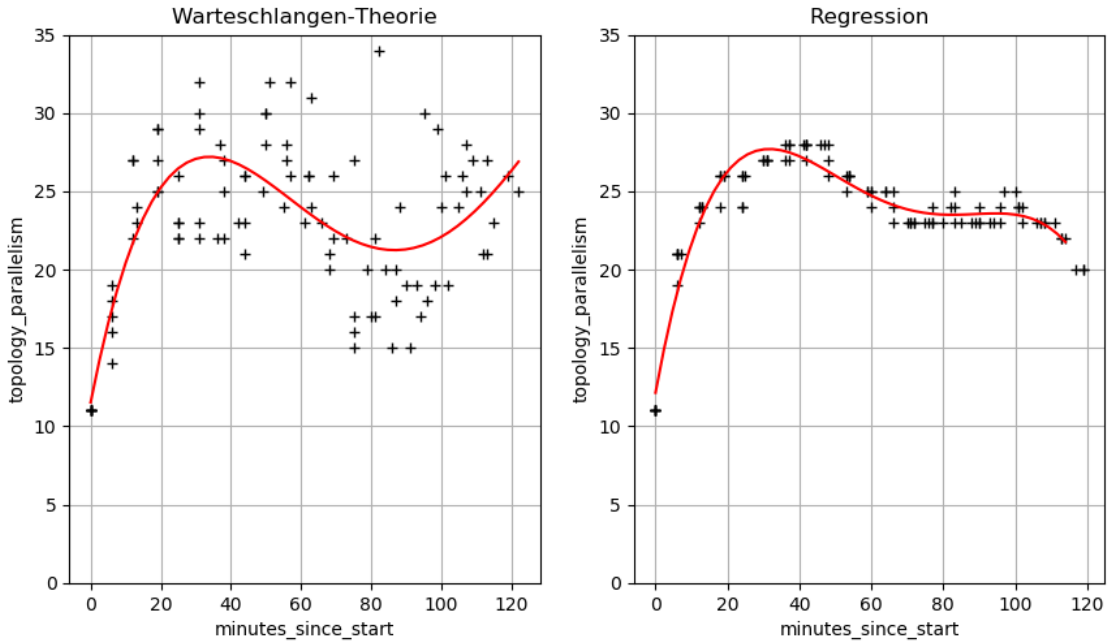

Abbildung 9.3: Parallelisierungsgrade während der Evaluation.

unter einen gewissen Grenzwert zu halten, sodass diese ein essentieller Bestandteil der Steuerung ist. Dies ist bei RT nicht der Fall. Jedoch ist festzustellen, dass der gesetzte Maximalwert von zehn Millisekunden für die Latenz des Pfades nicht eingehalten wurde. Der Grund dafür ist, dass die Berücksichtigung der gemessenen Latenz des Kanals für die Evaluation nicht aktiviert war. Somit wird die Netzwerklatenz und die reale Warteschlange vor dem Operator nicht berücksichtigt und die berechnete nicht über den Koeffizienten e angepasst wird.

Dass der Latenzwert bei RT vergleichsweise hoch ist, könnte daran liegen, dass das Modell eines Operators schlechte Vorhersagen getroffen hat. Dieser Operator erzeugt dann einen Flaschenhals der die Latenz des Tupels sehr erhöht. Dass die Latenz am Schluss von RT stark ansteigt, könnte daran liegen, dass die Trainingsdaten einen Abfall der eingehenden Tupel gegen Ende Aufweisen und deshalb der Parallelisierungsgrad geändert wird. Diese Struktur der Trainingsdaten wäre damit zu erklären, dass durch die vielen Umstellungen durch WT die Topologie oft gestoppt wird und so die Lesegeschwindigkeit die Ausgabe der Tupel von der Quelle beschränkt. Dies führt zu hohen Spitzenwerten für die Anzahl ankommender Tupel. Gegen Ende der Evaluation ist der Verlauf aber weniger Schwanken und die Spitzenwerte für ankommenden Tupel sinken.

9.7.3 Tupel-Fehlerrate

Zuletzt soll noch die Fehlerrate der Tupel betrachtet werden. Die Fehlerrate berechnet sich aus dem Verhältnis der Anzahl Tupel, deren Verarbeitung fehlgeschlagen ist, und der Anzahl Tupel deren Verarbeitung erfolgreich war. Ein Tupel wird von der Topologie als fehlgeschlagen gewertet, wenn es innerhalb von 60 Sekunden nicht erfolgreich verarbeitet wurde. In beiden Diagrammen der Abbildung 10.5 ist deutlich zu sehen, dass die Fehlerrate zu Beginn sehr hoch ist. Erst wenn die Operatoren ausreichend skaliert sind, stabilisiert sich die Fehlerrate. Für WT ist zu sehen, dass

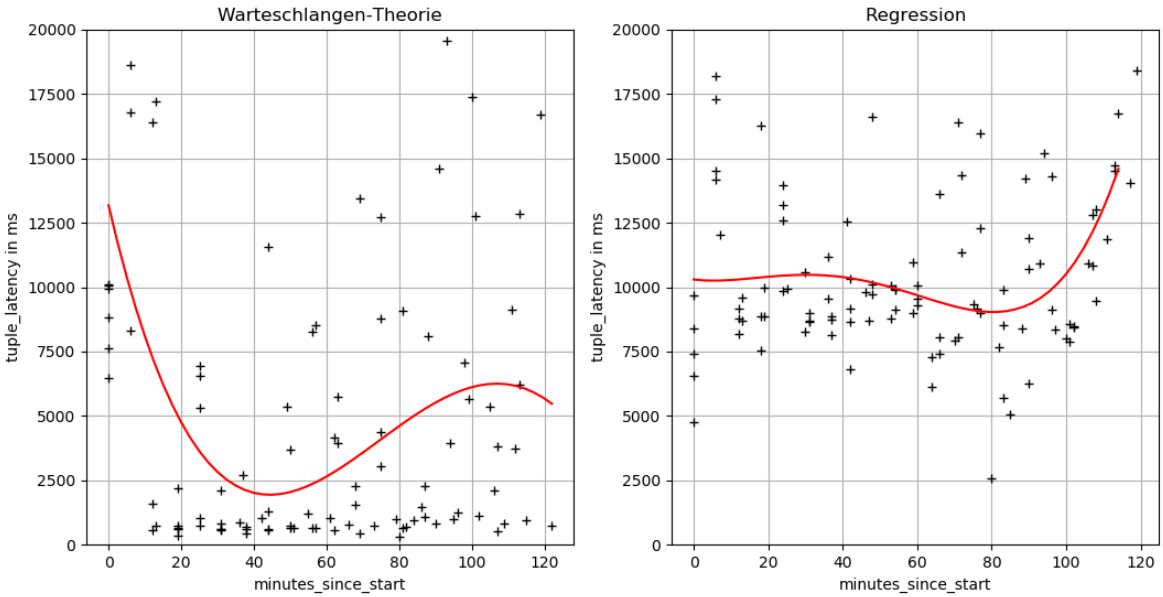

Abbildung 9.4: Latenz der Tupel während der Evaluation.

die Fehlerrate sehr oft nahe bei 0% liegt. Dies ist dadurch erklärbar, dass fehlerhafte Tupel erneut von der Quelle ausgegeben werden. Der Algorithmus reagiert anschließend stark auf die erhöhte Anzahl ausgegebener Tupel und kann den Überschuss kurzfristig ausgleichen. Diese Verhalten wurde in Kapitel 10.7.1 bereits beschrieben.

RG weist hingegen eine sehr hohe Fehlerrate auf. Dies kann darauf zurückgeführt werden, dass der Algorithmus langfristige Prognosen betrachtet und die Fehlerrate nicht kurzfristig ausgleicht. Zudem basieren die Prognosen auf Messwerten von WT. Da WT die Fehlerrate kurzfristig ausgleicht, aber RG auf einem Mittelwert der gemessenen Werte basiert, können diese Spitzen nicht abgefangen werden. Dies erklärt ebenfalls warum die Latenz der Tupel unter RG in der zweiten Hälfte konstant hoch bleibt. Im Vergleich zu WT beleibt der Parallelisierungsgrad von RG in der zweiten Stunde konstant und ist somit höher als bei WT. Allerdings ist die Latenz der Tupel unter RG dennoch höher als bei WT. Dies erklärt sich durch die konstant hohe Fehlerrate die nicht abgearbeitet werden kann, und somit die Operatoren dauerhaft stark belastet.

9.7.4 Schlussfolgerungen

Für das in der Evaluation verwendete Muster des Arbeitsaufwands scheint WT besser geeignet zu sein. Die Latenz und die Fehlerrate sind beide niedriger als bei RG. Allerdings benötigt WT zeitweise mehr Ressourcen, die zur Verfügung stehen müssen. Im Mittel sind die Kosten für Operatoren der Topologie ähnlich. Da durch die zeitliche Komporession der Daten die Spitzen und Tiefen schneller aufeinander folgen, ist es erklärbar, dass WT durch die eher kurzfristige reaktive Herangehensweise besser auf die schnelle Abfolge der Extreme reagieren kann. Dies setzt allerdings voraus, dass die ankommenden Tupel während der häufigen Änderungen der Topologie zwischengespeichert werden können. Der Algorithmus von Lohrmann et al. konnte trotz vielen Änderungen an der Topologie den Rückstand, der Tupeln die bearbeitet werden müssen, ausgleichen.

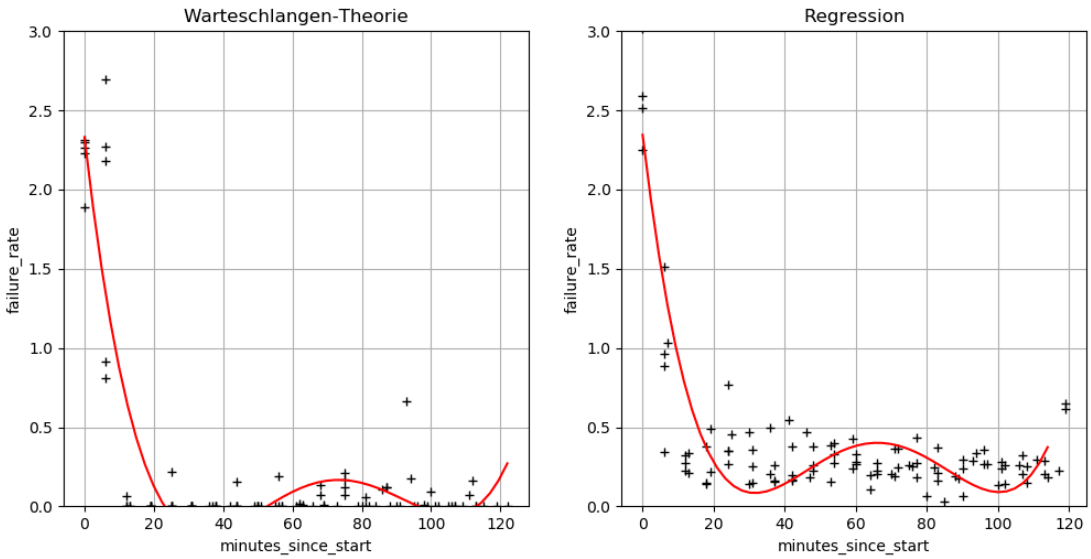

Abbildung 9.5: Fehlerrate der Tupel während der Evaluation.

Der Algorithmus von Zacheilas et al. ist relativ stark an die Mittelwerte der Trainingsdaten gebunden. Damit der Algorithmus erfolgreich eingesetzt werden kann, ist es notwendig, dass Messdaten für möglichst viele Parallelisierungsgrade eines Operators vorhanden sind. Sind diese Daten nicht vorhanden unterscheiden sich die Vorhersagen für hohe und niedere Parallelisierungsgraden nicht stark genug. Durch eine Optimierung der Hyperparameter kann diesem Verhalten ebenfalls entgegengewirkt werden. Ein weiterer Punkt ist, dass RG den letzten Operator der Topologie nie skalieren wird, da die Anzahl der ausgegebenen Tupel immer null entspricht. Somit ist die Anzahl verpasster Tupel des Operators und die damit verbundenen Kosten immer null. Für einen Workload der wenige Ausreisser besitzt und sich eher moderat verändert wäre der Algorithmus von Zacheilas et al. vermutlich besser geeignet als der Algorithmus mit Warteschlangen-Theorie.

10 Zusammenfassung und Ausblick

In der vorliegenden Arbeit wurde zuerst die Grundlage für das Skalieren von Operatoren in CEP-Topologien erläutert. Um den Parallelisierungsgrad der Operatoren zu steuern, wurden in den letzten Jahren mehrere Algorithmen vorgeschlagen. Verschiedene Algorithmen wurden betrachtet und deren Stärken und Schwächen analyisert. Anschließend wurden zwei Algorithmen gewählt die implementiert und ausgiebig evaluiert werden sollen.

Für die Implementation der Algorithmen wird ein Framework erstellt, welches es erlaubt dass diese Algorithmen unabhängig vom gesteuerten CEP-System arbeiten könne. Das Framework bietet als Schnittstelle für die Algorithmen ein Graph-Modell, das die Werte der Topologie autonom ausliest und bereitstellt. Das Graph-Modell ermöglicht es neue Algorithmen zu implementieren, ohne dass diese sich mit der Beschaffung von Messdaten aus den CEP-Systemen befassen müssen. Damit das Framework universell einsetzbar ist, muss für das entsprechende CEP-System ein Adapter implementiert werden. Der Adapter muss dabei die Spezifika des CEP-Systems kapseln und eine REST-Schnittstelle für das Framework bereitstellen. Um die Algorithmen auf dem CEP-System Heron evaluieren zu können, wurde ein bestehender Adapter für Heron so erweitert, dass er alle benötigten Funktionen bereitstellt.

Anschließend wurden die beiden Algorithmen implementiert. Dabei wurden zum Teil leichte Modifikationen unternommen um sie robuster und flexibler zu machen. Einer der Algorithmen wurde von Lohrmann et al. vorgeschlagen und basiert auf der Verwendung der Warteschlangentheorie. Er verwendet das Modell, dass Tupel vor einem Operator warten müssen, bis sie verarbeitet werden. Der Algorithmus versucht die Wartezeit von Tupeln über das Skalieren von Operatoren unter einen benutzerdefinierten Grenzwert zu senken Der zweite Algorithmus basiert auf Vorhersagen über den zukünftigen Zustand der Topologie und wurde von Zacheilas et al. vorgeschlagen. Die Vorhersagen werden mit Hilfe von maschinellem Lernen getroffen. Anschließend werden die Vorhersagen mit einer Kostenfunktion gewichtet und in einen Graph von Zustandsübergängen modelliert. Der kürzeste Pfad durch den Graph bestimmt anschließend den nächsten Parallelisierungsgrad des Operators.

Zuletzt wurde ein Test-Cluster mit Heron aufgebaut, um die Algorithmen zu evaluieren. Für die Evaluation wurden eine Menge von Daten benötigt, die eine Topologie auslasten kann. Deshalb wurden über eine Woche Daten von der Twitter Streaming API ausgelesen. Außerdem wurde eine Topologie implementiert, die Twitter-Daten analysiert und für die Evaluation eingesetzt wurde. Während der Evaluation mussten die Algorithmen die implementierte Topologie steuern, während diese die gesammelten Daten innerhalb von zwei Stunden verarbeitete. Dabei wurden verschiedene Messwerte erhoben und gegenüber gestellt. Das Ergebnis war, dass der Algorithmus von Lohrmann et al. für das Verhaltensmuster der verarbeiteten Daten deutlich bessere Werte für Tupel-Latenz und Fehlerrate erzielte. Die Erklärung dafür liegt darin, dass der Operator reaktiv auf Schwankungen in der Topologie reagiert. Der Algorithmus von Zacheilas et al. hält den Parallelisierungsgrad der Operatoren eher konstant und gleicht kurzfristige Schwankungen nicht

aus. Er ist daher besser für einen sich stetig ändernden Arbeitsaufwand mit wenig Minima und Maxima geeignet.

Ausblick

Neben der durchgeführten Evaluation sollten noch weitere Evaluationen mit den implementierten Algorithmen durchgeführt werden. Die Parameter dabei sind sehr vielfältig. Zum einen könnte die Rate, mit der die Quelle Tupel ausgibt, angepasst werden. Ebenso kann das Zeitfenster für das Erfassen der Messwerte und deren Überprüfung durch den Algorithmus verändert werden. Da in der Evaluation nur drei Minuten des fünf minütigen Zeitfensters betrachtet werden, wäre es interessant wie sich der Zustand direkt nach dem Skalieren auf die evaluierten Messwerte auswirkt. Interessant wäre ebenfalls ein andere Daten oder ein anderes Muster des anfallenden Arbeitsaufwandes zu testen.

Außerdem kann die Konfiguration von Heron angepasst werden, sodass mehr Tupel in den Buffer vor den Operatoren geladen werden können. Diese Änderung würde sich auf die Anzahl fehlgeschlagener Tupel auswirken. Ebenso könnte die Konfiguration der Topologie abgewandelt werden. Wenn die Topologie mit der ”AT_MOST_ONCE” Garantie konfiguriert wird, werden fehlerhafte Tupel nicht mehr wiederholt, was die Last der Operatoren deutlich verringern könnte. Ein weiterer interessanter Punkt wäre zu sehen, wie sich die Messwerte der Algorithmen verhalten wenn die Drosselung des Tupel-Stroms in der Topologie aktiviert ist.

Ein weiterer Punkt ist die Optimierung der Trainigsdaten für die Vorhersage. Mit einem größeren Datensatz könnte der Algorithmus von Zacheilas et al. exaktere Vorhersagen treffen und somit besser agieren. Hier spielen auch die Hyperparamter eine große Rolle. Diese wurden für die vorliegende Evaluation nicht optimiert. Die Qualität der Vorhersagen würde sich auch mit den aktuellen Trainingsdaten verbessern lassen, wenn das jeweilige Optimum der Hyperparameter berechnet wurde.

Desweiteren könnte das Framework so erweitert werden, dass es möglich ist weitere CEP-Systeme zu steuern. Momentan ist nur der Adapter für Heron verfügbar. Die Erweiterung würde es ermöglichen die Algorithmen auf anderen Systemen zu testen. Die Möglichkeit noch mehr verschiedene Algorithmen miteinander zu vergleichen sollte ebenfalls in Betracht gezogen werden. Das Framework stellt das Graph-Modell als Grundlage bereit, sodass neue Algorithmen mit weniger Aufwand implementiert werden können.

Literaturverzeichnis

[AVB17] M. D. de Assuncao, A. d. S. Veith, R. Buyya. „Distributed Data Stream Processing and Edge Computing: A Survey on Resource Elasticity and Future Directions“. en. In: *arXiv:1709.01363 [cs]* (Sep. 2017). arXiv: 1709.01363. URL: http://arxiv.org/abs/1709.01363 (besucht am 12. 10. 2018) (zitiert auf S. 13, 16).

[Aki+13] T. Akidau, A. Balikov, K. Bekiroğlu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, S. Whittle. „MillWheel: Fault-tolerant Stream Processing at Internet Scale“. In: *Proc. VLDB Endow.* 6.11 (Aug. 2013), S. 1033–1044. ISSN: 2150-8097. DOI: 10.14778/2536222.2536229. URL: http://dx.doi.org/10.14778/2536222.2536229 (besucht am 06. 06. 2018) (zitiert auf S. 13).

[BTz13] C. Balkesen, N. Tatbul, M. T. Özsu. „Adaptive Input Admission and Management for Parallel Stream Processing“. In: *Proceedings of the 7th ACM International Conference on Distributed Event-based Systems*. DEBS '13. New York, NY, USA: ACM, 2013, S. 15–26. ISBN: 978-1-4503-1758-0. DOI: 10.1145/2488222.2488258. URL: http://doi.acm.org/10.1145/2488222.2488258 (besucht am 06. 06. 2018) (zitiert auf S. 13).

[CF+13] R. Castro Fernandez, M. Migliavacca, E. Kalyvianaki, P. Pietzuch. „Integrating Scale out and Fault Tolerance in Stream Processing Using Operator State Management“. In: *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. SIGMOD '13. New York, NY, USA: ACM, 2013, S. 725–736. ISBN: 978-1-4503-2037-5. DOI: 10.1145/2463676.2465282. URL: http://doi.acm.org/10.1145/2463676.2465282 (besucht am 06. 06. 2018) (zitiert auf S. 14).

[CM10] G. Cugola, A. Margara. „TESLA: a formally defined event specification language“. en. In: *Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems - DEBS '10*. Cambridge, United Kingdom: ACM Press, 2010, S. 50. ISBN: 978-1-60558-927-5. DOI: 10.1145/1827418.1827427. URL: http://portal.acm.org/citation.cfm?doid=1827418.1827427 (besucht am 13. 10. 2018) (zitiert auf S. 13).

[CM12] G. Cugola, A. Margara. „Complex event processing with T-REX“. en. In: *Journal of Systems and Software* 85.8 (Aug. 2012), S. 1709–1728. ISSN: 01641212. DOI: 10.1016/j.jss.2012.03.056. URL: http://linkinghub.elsevier.com/retrieve/pii/S0164121212000842 (besucht am 13. 10. 2018) (zitiert auf S. 13).

[CVZ13] P. Carbone, K. Vandikas, F. Zaloshnja. „Towards Highly Available Complex Event Processing Deployments in the Cloud“. en. In: *2013 Seventh International Conference on Next Generation Mobile Apps, Services and Technologies*. Prague, Czech Republic: IEEE, Sep. 2013, S. 153–158. ISBN: 978-1-4799-2010-5. DOI: 10.1109/NGMAST.2013.35. URL: http://ieeexplore.ieee.org/document/6658116/ (besucht am 12. 10. 2018) (zitiert auf S. 15).

[DMM16] T. De Matteis, G. Mencagli. „Keep calm and react with foresight: strategies for low-latency and energy-efficient elastic data stream processing“. en. In: ACM Press, 2016, S. 1–12. ISBN: 978-1-4503-4092-2. DOI: 10.1145/2851141.2851148. URL: http://dl.acm.org/citation.cfm?doid=2851141.2851148 (besucht am 06. 06. 2018) (zitiert auf S. 14).

[Ged+14] B. Gedik, S. Schneider, M. Hirzel, K.-L. Wu. „Elastic Scaling for Data Stream Processing“. en. In: *IEEE Transactions on Parallel and Distributed Systems* 25.6 (Juni 2014), S. 1447–1463. ISSN: 1045-9219. DOI: 10.1109/TPDS.2013.295. URL: http://ieeexplore.ieee.org/document/6678504/ (besucht am 27. 04. 2018) (zitiert auf S. 17, 18).

[HKR] N. R. Herbst, S. Kounev, R. Reussner. „Elasticity in Cloud Computing: What It Is, and What It Is Not“. en. In: (), S. 6 (zitiert auf S. 15).

[Hei+14a] T. Heinze, V. Pappalardo, Z. Jerzak, C. Fetzer. „Auto-scaling techniques for elastic data stream processing“. In: *2014 IEEE 30th International Conference on Data Engineering Workshops*. März 2014, S. 296–302. DOI: 10.1109/ICDEW.2014.6818344 (zitiert auf S. 17).

[Hei+14b] T. Heinze, Z. Jerzak, G. Hackenbroich, C. Fetzer. „Latency-aware elastic scaling for distributed data stream processing systems“. en. In: ACM Press, 2014, S. 13–22. ISBN: 978-1-4503-2737-4. DOI: 10.1145/2611286.2611294. URL: http://dl.acm.org/citation.cfm?doid=2611286.2611294 (besucht am 27. 04. 2018) (zitiert auf S. 14).

[Hei+15] T. Heinze, L. Roediger, A. Meister, Y. Ji, Z. Jerzak, C. Fetzer. „Online Parameter Optimization for Elastic Data Stream Processing“. In: *Proceedings of the Sixth ACM Symposium on Cloud Computing*. SoCC '15. New York, NY, USA: ACM, 2015, S. 276–287. ISBN: 978-1-4503-3651-2. DOI: 10.1145/2806777.2806847. URL: http://doi.acm.org/10.1145/2806777.2806847 (besucht am 06. 06. 2018) (zitiert auf S. 18).

[KLL17] R. K. Kombi, N. Lumineau, P. Lamarre. „A Preventive Auto-Parallelization Approach for Elastic Stream Processing“. In: *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. Juni 2017, S. 1532–1542. DOI: 10.1109/ICDCS.2017.253 (zitiert auf S. 18).

[Kul+15] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, S. Taneja. „Twitter Heron: Stream Processing at Scale“. In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. SIGMOD '15. New York, NY, USA: ACM, 2015, S. 239–250. ISBN: 978-1-4503-2758-9. DOI: 10.1145/2723372.2742788. URL: http://doi.acm.org/10.1145/2723372.2742788 (besucht am 08. 10. 2018) (zitiert auf S. 13, 56).

[LB17] X. Liu, R. Buyya. „Performance-Oriented Deployment of Streaming Applications on Cloud“. en. In: *IEEE Transactions on Big Data* (2017), S. 1–1. ISSN: 2332-7790. DOI: 10.1109/TBDATA.2017.2720622. URL: http://ieeexplore.ieee.org/document/7962193/ (besucht am 18. 04. 2018) (zitiert auf S. 18).

[LJK15] B. Lohrmann, P. Janacik, O. Kao. „Elastic Stream Processing with Latency Guarantees“. en. In: IEEE, Juni 2015, S. 399–410. ISBN: 978-1-4673-7214-5. DOI: 10.1109/ICDCS.2015.48. URL: http://ieeexplore.ieee.org/document/7164926/ (besucht am 18. 04. 2018) (zitiert auf S. 19, 21, 24, 41, 42).

[LWK14] B. Lohrmann, D. Warneke, O. Kao. „Nephele streaming: stream processing under QoS constraints at scale“. en. In: *Cluster Computing* 17.1 (März 2014), S. 61–78. ISSN: 1386-7857, 1573-7543. DOI: 10.1007/s10586-013-0281-8. URL: http://link.springer.com/10.1007/s10586-013-0281-8 (besucht am 23. 04. 2018) (zitiert auf S. 13, 47).

[MKR15] R. Mayer, B. Koldehofe, K. Rothermel. „Predictable Low-Latency Event Detection With Parallel Complex Event Processing“. In: *IEEE Internet of Things Journal* 2.4 (Aug. 2015), S. 274–286. ISSN: 2327-4662. DOI: 10.1109/JIOT.2015.2397316 (zitiert auf S. 13, 18).

[Noaa] *GET statuses/sample — Twitter Developers*. URL: https://developer.twitter.com/en/docs/tweets/sample-realtime/overview/GET_statuse_sample (besucht am 09. 10. 2018) (zitiert auf S. 55).

[Noab] *Heron Metrics*. URL: https://apache.github.io/incubator-heron/docs/operators/heron-tracker-api/#topologies_metrics (zitiert auf S. 34).

[Noac] *Home page*. en-US. URL: http://www.espertech.com/ (besucht am 13. 10. 2018) (zitiert auf S. 13).

[Noad] *How much is 1%?* en. Nov. 2017. URL: https://twittercommunity.com/t/how-much-is-1/95878 (besucht am 09. 10. 2018) (zitiert auf S. 55).

[Noae] *Smile - Statistical Machine Intelligence and Learning Engine*. URL: http://haifengl.github.io/smile/ (besucht am 08. 10. 2018) (zitiert auf S. 50).

[Noaf] *Welcome to JGraphT - a free Java Graph Library*. URL: https://jgrapht.org/ (besucht am 08. 10. 2018) (zitiert auf S. 50).

[Ras04] C. E. Rasmussen. „Gaussian processes in machine learning“. In: *Advanced lectures on machine learning*. Springer, 2004, S. 63–71 (zitiert auf S. 49, 51).

[SB13] K.-U. Sattler, F. Beier. „Towards Elastic Stream Processing: Patterns and Infrastructure.“ In: *BD3@ VLDB* 1018 (2013), S. 49–54 (zitiert auf S. 13).

[SeZ05] M. Stonebraker, U. Çetintemel, S. Zdonik. „The 8 Requirements of Real-time Stream Processing“. In: *SIGMOD Rec.* 34.4 (Dez. 2005), S. 42–47. ISSN: 0163-5808. DOI: 10.1145/1107499.1107504. URL: http://doi.acm.org/10.1145/1107499.1107504 (besucht am 27. 04. 2018) (zitiert auf S. 15).

[Sha+03] M. A. Shah, J. M. Hellerstein, S. Chandrasekaran, M. J. Franklin. „Flux: an adaptive partitioning operator for continuous query systems“. In: *Proceedings 19th International Conference on Data Engineering (Cat. No.03CH37405)*. März 2003, S. 25–36. DOI: 10.1109/ICDE.2003.1260779 (zitiert auf S. 14).

[Vog+] A. Vogel, D. Griebler, D. D. Sensi, M. Danelutto, L. G. Fernandes. „Autonomic and Latency-Aware Degree of Parallelism Management in SPar“. en. In: (), S. 12 (zitiert auf S. 17).

[WDR06] E. Wu, Y. Diao, S. Rizvi. „High-performance complex event processing over streams“. en. In: ACM Press, 2006, S. 407. ISBN: 978-1-59593-434-5. DOI: 10.1145/1142473.1142520. URL: http://portal.acm.org/citation.cfm?doid=1142473.1142520 (besucht am 27. 04. 2018) (zitiert auf S. 13).

[YZJ05] Ying Xing, S. Zdonik, Jeong-Hyon Hwang. „Dynamic Load Distribution in the Borealis Stream Processor“. en. In: *21st International Conference on Data Engineering (ICDE'05)*. Tokyo, Japan: IEEE, 2005, S. 791–802. ISBN: 978-0-7695-2285-2. DOI: 10.1109/ICDE.2005.53. URL: http://ieeexplore.ieee.org/document/1410193/ (besucht am 13. 10. 2018) (zitiert auf S. 14).

[Zac+15] N. Zacheilas, V. Kalogeraki, N. Zygouras, N. Panagiotou, D. Gunopulos. „Elastic complex event processing exploiting prediction“. In: IEEE, Okt. 2015, S. 213–222. ISBN: 978-1-4799-9926-2. DOI: 10.1109/BigData.2015.7363758. URL: http://ieeexplore.ieee.org/document/7363758/ (besucht am 06. 06. 2018) (zitiert auf S. 18, 49–52).

[Zac+16] N. Zacheilas, N. Zygouras, N. Panagiotou, Kalogeraki, Vana, Gunopulos, Dimitrios. „Dynamic Load Balancing Techniques for Distributed Complex Event Processing Systems“. en. In: *Distributed applications and interoperable systems*. New York, NY: Springer Berlin Heidelberg, 2016. ISBN: 978-3-319-39576-0 (zitiert auf S. 13).

Alle URLs wurden zuletzt am 14. 10. 2018 geprüft.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift