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

In diesem Kapitel steht die Einleitung zu dieser Arbeit. Sie soll nur als Beispiel dienen und hat nichts mit dem Buch [Wee+05] zu tun. Nun viel Erfolg bei der Arbeit!

Bei LaTeX werden Absätze durch freie Zeilen angegeben. Da die Arbeit über ein Versionskontrollsystem versioniert wird, ist es sinnvoll, pro *Satz* eine neue Zeile im `.tex`-Dokument anzufangen. So kann einfacher ein Vergleich von Versionsständen vorgenommen werden.

Gliederung

Die Arbeit ist in folgender Weise gegliedert:

Kapitel ?? – ??: Hier werden werden die Grundlagen dieser Arbeit beschrieben.

Kapitel 6 – Zusammenfassung und Ausblick fasst die Ergebnisse der Arbeit zusammen und stellt Anknüpfungspunkte vor.

2 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.

2.0.1 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.

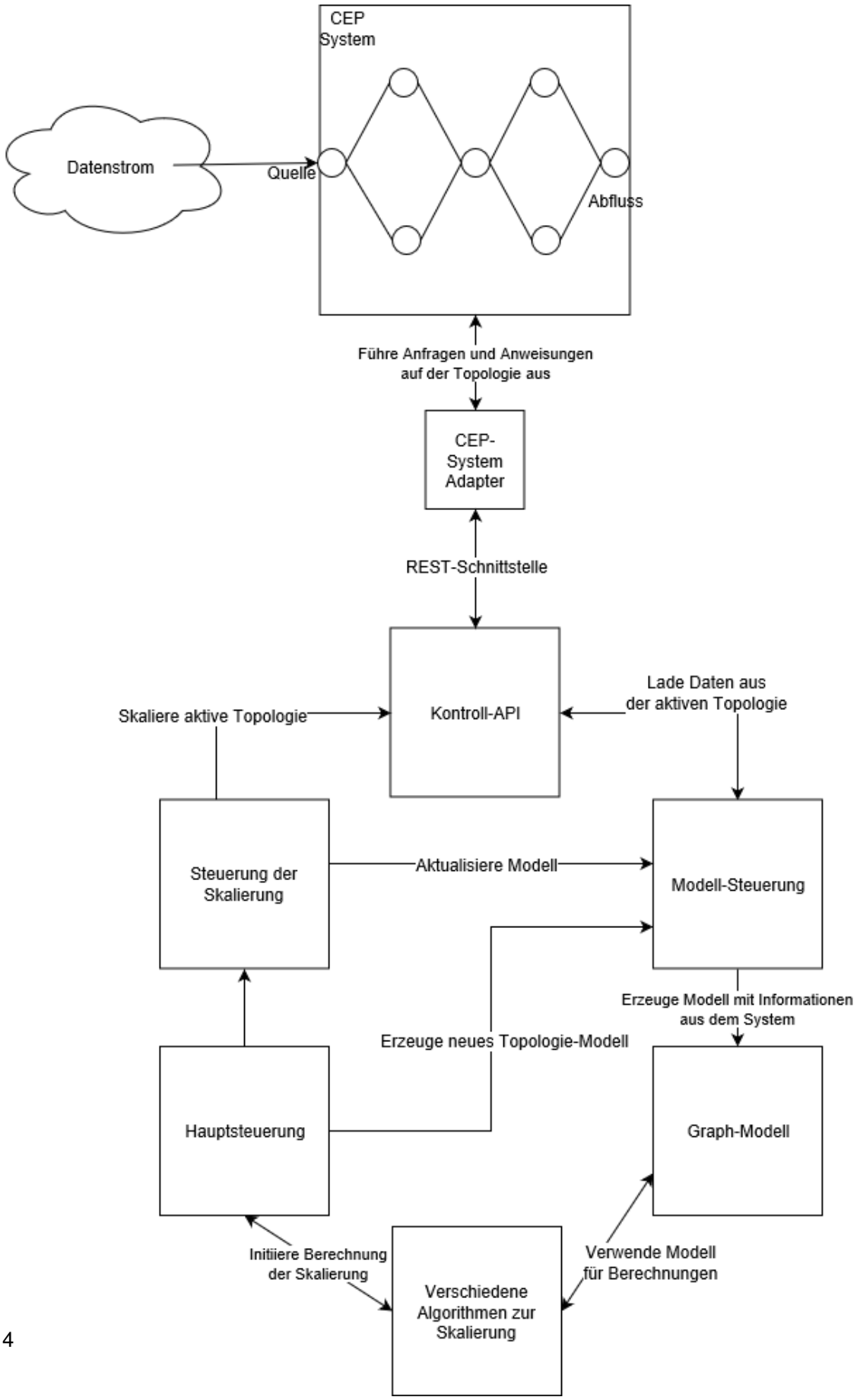

Abbildung 2.1: Architektur des Systems

2.1 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.

2.2 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.

2.3 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.

2.4 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.

2.5 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.

2.6 UML-Diagramm des Frameworks

3 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.

3.1 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.

3.2 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 [Noa]. 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.

3.3 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 3.1: 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 3.1: 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 3.1: 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 3.1: 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.

3.4 UML-Diagramm

4 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.

4.1 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.

4.1.1 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.

4.1.2 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.

4.1.3 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.

4.1.4 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.

4.1.5 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.

4.1.6 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 $max(p+1, p_Delta$.

4.1.7 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.

4.1.8 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.

4.2 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.

4.2.1 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.

4.2.2 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.

4.2.3 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.

4.2.4 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.

5 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 [**rasmussen2004gaussian**]. 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

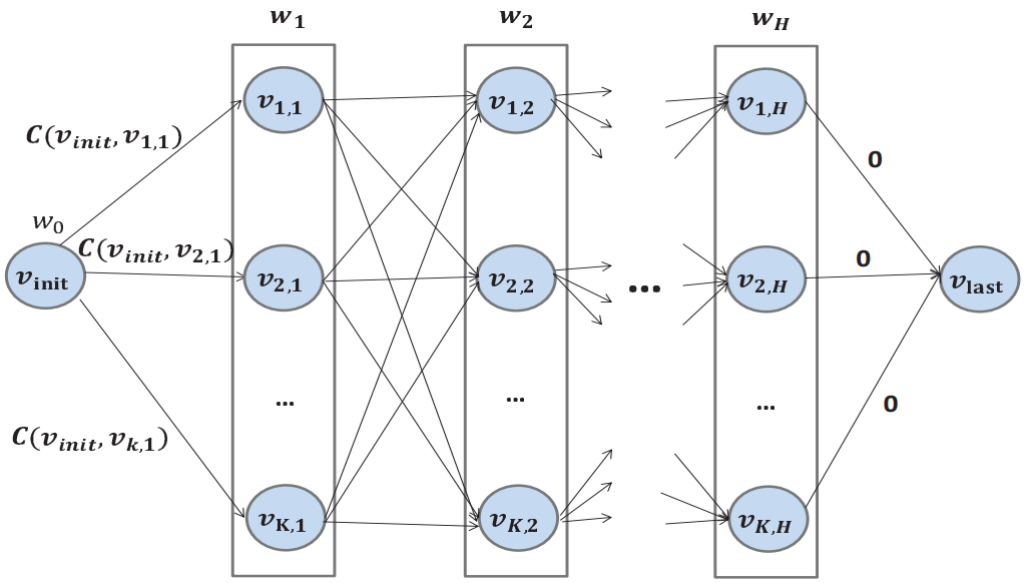

Abbildung 5.1: Graph der Zustandsübergänge [Zac+15].

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.

5.1 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 [**noauthor_welcome_nodate**] 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 [**noauthor_smile_nodate**].

5.1.1 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.

5.1.2 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.

5.1.3 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 [**rasmussen2004gaussian**]. 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.

5.1.4 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.

5.2 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.

5.2.1 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.

5.2.2 Hyperparameter

Tabelle 5.1: 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. Dementsprechend verhalten sich auch die vom Modell vorhergesagten Werte. Lamda wird verwendet um die Komponenten des Eingabevektors zu gewichten. Für den verwendeten Kernel bedeutet das, dass der negative Exponent kleiner wird. Im Umkehrschluss fällt die Korrelation für diesen Wert 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 auch 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 mit gegebenenfalls mit $lambda$ ausgeglichen werden.

5.2.3 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.

6 Zusammenfassung und Ausblick

Hier bitte einen kurzen Durchgang durch die Arbeit.

Ausblick

...und anschließend einen Ausblick

Literaturverzeichnis

[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. 27, 28).

[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. 33).

[Noa] *Heron Metrics*. URL: https://apache.github.io/incubator-heron/docs/operators/heron-tracker-api/#topologies_metrics (zitiert auf S. 20).

[Wee+05] S. Weerawarana, F. Curbera, F. Leymann, T. Storey, D. F. Ferguson. *Web Services Platform Architecture : SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging, and More*. Prentice Hall PTR, 2005. ISBN: 0131488740. DOI: 10.1.1/jpb001 (zitiert auf S. 11).

[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. 35–38).

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