Admission Control für kontinuierliche Anfragen

Timo Michelsen, Marco Grawunder, Cornelius Ludmann, H.-Jürgen Appelrath

Universität Oldenburg, Department für Informatik
Escherweg 2, 26129 Oldenburg
{timo.michelsen, marco.grawunder, cornelius.ludmann, appelrath}@uni-ol.de

Abstract: Überlast bedeutet, dass an ein System mehr Anforderungen gestellt werden, als es erfüllen kann. Im schlechtesten Fall ist es nicht mehr ansprechbar oder stürzt ab. Für Datenbankmanagementsysteme (DBMS) existiert eine spezielle Komponente, Admission Control (AC) genannt, welche die Systemlast überwacht, eintreffende Anfragen vor deren Ausführung überprüft und ggfs. zurückstellt. Für kontinuierliche Anfragen, welche permanent ausgeführt werden, reicht diese Art von AC jedoch nicht aus: Es kann zur Konfliktlösung nicht mehr auf die Terminierung einer Anfrage gewartet werden. Zudem ist die Verarbeitung datengetrieben, d. h. Umfang und Inhalt der Daten können stark variieren.

Diese Arbeit stellt ein Konzept vor, wie eine flexible und anpassbare AC-Komponente für kontinuierliche Anfragen auf Basis eines Event-Condition-Action-Modells (ECA) umgesetzt werden kann. Zur Regeldefinition wird die einfache Sprache CADL vorgestellt, die eine hohe Flexibilität und Anpassbarkeit der kontinuierlichen AC an konkrete Systeme und Hardware erlaubt. Die Evaluation zeigt mittels Odysseus, einem Framework für Datenstrommanagementsysteme, dass das Konzept effizient funktioniert und effektiv zur Lastkontrolle und Ergreifung von Maßnahmen zur Lastreduktion eingesetzt werden kann.

1 Einleitung

Für Datenbanksysteme ist es essentiell, jederzeit eine reibungslose Verarbeitung der Daten sicherzustellen. Überlastungen können Antwortzeiten vergrößern, wirtschaftliche Kosten verursachen und im Extremfall zu Ausfällen führen. Solche Situationen können entstehen, wenn das Datenbanksystem eine große Menge an Anfragen verarbeiten muss. Daher ist es wichtig, Überlastsituationen generell zu vermeiden oder schnell aufzulösen. Dementsprechend müssen Datenbanksysteme in der Lage sein, ihre aktuelle Systemauslastung zu überwachen und in kritischen Situationen Maßnahmen zu ergreifen. Dieser Vorgang der Lastüberwachung und -kontrolle wird i. Allg. als *Admission Control* (AC) bezeichnet.

Für klassische Datenbankmanagementsysteme (DBMS) existieren bereits ausgereifte AC-Konzepte [HSH07]. Ist die vermutete zusätzliche Systemlast einer neuen Anfrage zu hoch (z. B. fehlender freier Arbeitsspeicher), kann die Anfrage abgelehnt, zurückgestellt, ausgelagert oder nur unter eingeschränkten Rahmenbedingungen ausgeführt werden. Diese Möglichkeiten sind für einmalig ausgeführte Anfragen ausreichend. Es existieren jedoch Anwendungsszenarien, die eine kontinuierliche und zeitnahe Verarbeitung erfordern. Dort

senden *aktive Datenquellen* autonom theoretisch unbegrenzte Mengen an Daten, sogenannte *Datenströme* [Krä09]. Diese können vom System ausschließlich sequenziell gelesen werden [BBD+02]. Anfragen, welche solche Datenquellen verarbeiten, werden i. Allg. *kontinuierliche Anfragen* genannt und können in DBMS bspw. durch Trigger realisiert werden und stehen in spezialisierten Datenstrommanagementsystemen (DSMS) unmittelbar zur Verfügung [BBD+04]. Eine kontinuierliche Anfrage kann – ähnlich zu Planoperatoren zu DBMS – als ein gerichteter, azyklischer Operatorgraph interpretiert werden. Die Knoten beschreiben Operatoren, welche einzelne Verarbeitungsschritte repräsentieren. Die Kanten stehen für die Datenströme zwischen den Operatoren.

Für kontinuierliche Anfragen reichen die etablierten Konzepte zur Lastkontrolle jedoch nicht mehr aus, da permanent laufende Anfragen nicht nur zu Beginn der Verarbeitung überprüft werden müssen. Änderungen im Datenlieferungsverhalten von Quellen können zu kritischen Lastsituationen führen – auch ohne dass der Nutzer neue Anfragen hinzufügt. Zudem sollte für jedes Verarbeitungssystem die Lastkontrolle aufgrund unterschiedlicher Anwendungskontexte und Hardware individuell konfiguriert werden können.

In dieser Arbeit wird ein Konzept für eine flexible und erweiterbare AC-Komponente für kontinuierliche Anfragen vorgestellt, die sogenannte *kontinuierliche AC*. Grundlage ist ein dynamisches Regelwerk in Form des Event-Condition-Action (ECA)-Modells. *Events* (Ereignisse) sind in diesem Fall Änderungen im Systemzustand. *Actions* (Aktionen) sind die Maßnahmen, die die AC zur Lastkontrolle und -steuerung ausführen kann. Mit dieser Vorgehensweise ist es möglich, eine AC-Komponente zu entwickeln, die den besonderen Rahmenbedingungen der kontinuierlichen Anfragen gewachsen ist. Ziel dieser Arbeit ist es somit, die Möglichkeit eines ECA-Modells zur Umsetzung einer flexiblen, kontinuierlichen Admission Control aufzuzeigen. Die entwickelte Sprache CADL (Countinuous Admission Control Language) erlaubt die Definition komplexer Ereignisse und Regeln, um die Lastkontrolle individuell zu konfigurieren. Es ist nicht Ziel dieser Arbeit, eine konkrete AC für eine konkrete Anwendungsdomäne vorzustellen.

Die vorliegende Arbeit ist wie folgt gegliedert: Im Folgenden Abschnitt 2 wird das Konzept der kontinuierlichen AC genauer erklärt. In Abschnitt 3 wird die Evaluation des Konzepts erläutert. Anschließend werden in Abschnitt 4 die verwandten Arbeiten aufgelistet und beschrieben. Den Abschluss bildet Abschnitt 5 mit einer Zusammenfassung dieser Arbeit.

2 Konzept einer kontinuierlichen Admission Control

Kontinuierliche Anfragen können im Laufe ihrer Ausführung unterschiedliche Zustände einnehmen, welche sich je nach Verarbeitungssystem unterscheiden. Je nach Zustand erzeugt eine kontinuierliche Anfragen unterschiedliche Mengen an Systemlast. Abbildung 1 zeigt ein Beispiel für übliche Zustände und deren Übergänge. Im Allgemeinen ist jede neue kontinuierliche Anfrage zunächst *inaktiv*, d. h., sie ist installiert, verarbeitet jedoch keine Daten und erzeugt keine Systemlast. Eine solche Anfrage kann durch *starten* in den Zustand *aktiv* übergehen: Daten werden empfangen, verarbeitet und die Ergebnisse

ausgegeben. Dadurch steigt auch die Auslastung des Systems. Durch *stoppen* kann die Anfrage wieder auf *inaktiv* gesetzt werden. In den meisten Systemen können Anfragen ohne Berücksichtigung des Zustands jederzeit komplett entfernt werden.

Einige Systeme erlauben es, Anfragen zu pausieren, d. h., die Daten werden weiterhin von den Quellen empfangen, zwischengespeichert, aber nicht verarbeitet. Dadurch sinkt die Systemlast, jedoch wir mit zunehmender Dauer mehr Arbeitsspeicher zur Zwischenspeicherung benötigt. Alternativ kann die Last durch das Verwerfen von Daten reduziert werden (*partielle* Ausführung, sog. *Load Shedding* [TcZ+03]). Gegebenenfalls sind hier Kombinationen denkbar (z. B. *partiell pausiert*), d. h., es werden nicht alle Daten zwischengespeichert.

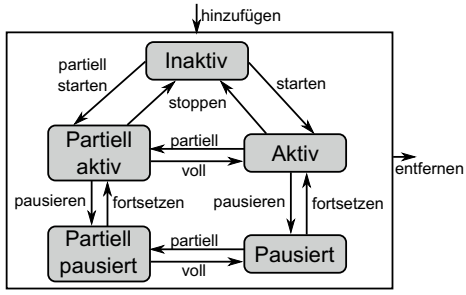

Abbildung 1: Mögliche Zustände für eine kontinuierliche Anfrage.

Die Systemlast für solche kontinuierliche Anfragen setzt sich i. d. R. aus den Systemressourcen Prozessor, Arbeitsspeicher sowie Netzwerkdatenrate zusammen. Die ersten beiden ergeben sich direkt aus der Tatsache, dass die Daten lediglich im Arbeitsspeicher zwischengespeichert und sofort verarbeitet werden. Aus diesem Grund wird hier externer Speicher vernachlässigt. Die Netzwerkbandbreite ist bei im Netzwerk verteilten Anfragen entscheident. Wird von einer Systemressource eine größere Menge benötigt, als von der Maschine zur Verfügung gestellt wird, tritt eine Überlastsituation ein. Eine bevorstehende Überlastung kann beispielsweise dadurch ankündigt werden, dass eine kontinuierliche Anfrage ihren Zustand wechselt (bspw. gestartet wird) oder sich die Datenrate einer Datenquelle ändert. Weiterhin können sich die Datenwerte ändern, sodass sich die Selektivität einiger Operatoren verändert und mehr Elemente als zuvor verarbeiten werden müssen. Werden umgekehrt Systemressourcen wieder frei, so können bspw. zuvor gestoppte kontinuierliche Anfragen wieder gestartet werden oder aproximierende Algorithmen durch exaktere erstetzt werden.

Die Idee ist, Indikatoren der Überlastung und geringere Systemlasten als Ereignisse aufzufassen. Diese geben bekannt, wenn eine kontinuierliche Anfrage den Zustand ändert, wenn größere Fluktuationen in den Datenraten auftreten oder wenn bestimmte Systemauslastungen über- oder unterschritten werden. Immer wiederkehrende Ereignisse können genutzt werden, um kontinuierliche Anfragen regelmäßig neu zu überprüfen.

Grundlage des hier vorgestellten Konzeptes für die kontinuierliche AC ist es, diese Ereignisse zur Laufzeit zu erfassen, sie in Abhängigkeit zum aktuellen Systemzustand auszuwerten, (potenzielle) Chancen und Risiken zu erkennen und sie ggfs. mittels gezielter (individueller) Maßnahmen auszunutzen bzw. aufzulösen. Um solche Ereignisse effektiv in der kontinuierlichen AC verarbeiten zu können und gleichzeitig Flexibilität zu gewährleisten, wird in dieser Arbeit ein ECA-Modell eingesetzt. Das Modell erlaubt die Definitionen der Reaktionen und Maßnahmen anhand von *Regeln*, welche auftretende Ereignisse auswerten. Sie werden zu sogenannten *Regelsätzen* zusammengefasst.

2.1 CADL als Sprache zur Regeldefinition

Zur einfachen Definition der Regeln wird die Beschreibungssprache CADL vorgeschlagen, die speziell auf die Lastkontrolle ausgelegt ist. Dort werden Ereignisse, welche vom System gesendet werden, als *atomare Ereignisse* bezeichnet. Jedes Ereignis hat einen Typ mit unterschiedlichen Attributen. Beispielsweise könnte ein Ereignis `QueryActivatedEvent` im Attribut `query` die Anfrage beinhalten, die gerade aktiviert wurde (und das Ereignis ausgelöst hat). CADL nutzt die Punktnotation, um auf Attribute eines Ereignisses zuzugreifen. Eine Regel lässt sich in CADL wie folgt definieren:

```
ON EVENT <ETyp> <EVariable> IF <Bedingung> THEN <Aktionen>
```

Tritt das Ereignis ein und die Bedingung ist erfüllt, werden die Aktionen ausgeführt. In den Bedingungen kann mittels der Ereignisvariable EVariable auf Attribute des konkreten Ereignisses zugegriffen werden. Bedingungen sind i. d. R. herkömmliche logische Ausdrücke, die wiederum aus mehreren Ausdrücken bestehen können, die mit `OR` oder `AND` verknüpft werden. Logische Ausdrücke beinhalten die typischen Bestandteile wie Funktionen, Konstanten, mathematische Operatoren und Vergleichsoperationen. Weitere Ereignisse können basierend auf anderen Ereignissen ausgelöst werden. Diese *komplexen Ereignisse* werden in CADL wie folgt definiert:

```
DEFINE COMPLEX EVENT <Ereignisname>:  <Kontinuierliche Anfrage>.
```

Damit kann `<Ereignisname>` in den Regeln ausgewertet werden. Die Idee ist, kontinuierliche Anfragen zur Identifikation der komplexen Ereignisse einzusetzen. Diese können die Ereignisse als Datenstrom empfangen und auswerten. Das Resultat ist eine Menge an komplexen Ereignissen, welche unter `<Ereignisname>` ansprechbar sind. Der Vorteil ist, dass der Nutzer auf der einen Seite den vollen Sprachumfang des Verarbeitungssystems nutzen kann, um komplexe Ereignisse zu definieren, auf der anderen Seite die kontinuierliche AC bereits etablierte Komponenten des Systems wiederverwenden kann, wie bspw. den Optimierer. Dabei ist jedoch eine strikte Trennung von Anfragen von CADL und normalen Anfragen notwendig, um Manipulation der Anfragen der AC zu vermeiden.

Mit CADL ist es möglich, einfach und flexibel komplexe Ereignisse und Regelsätze zu definieren. Der Anwender kann eigene atomare Ereignisse, Bedingungen und Aktionen implementieren und in CADL einsetzen. Damit ist es möglich, das ECA-Modell für besondere Anwendungen zu spezialisieren. Es unterliegt jedoch der Verantwortung des Nutzers, für die Situation sinnvolle komplexe Ereignisse und Regeln zu definieren. Wie bei jedem ECA-System muss der Regelersteller auf konsistente Regeln achten. Dies soll hier jedoch nicht weiter betrachtet werden.

2.2 Architektur der kontinuierlichen AC

Der Kern des Konzeptes der kontinuierlichen AC besteht darin, einen mit CADL definierten Regelsatz zu nutzen, um auf atomare und komplexe Ereignisse zu reagieren und unter den gegebenen Bedingungen Aktionen zur Lastkontrolle durchzuführen. Die Architektur des Konzepts ist in Abbildung 2 zu sehen.

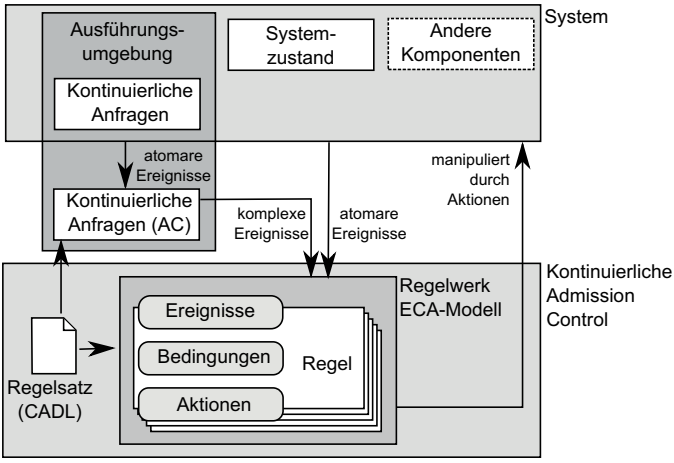

Abbildung 2: Konzeptionelle Übersicht der kontinuierlichen AC.

In der Ausführungsumgebung werden die (kontinuierlichen) Anfragen organisiert. Die Umgebung steuert und koordiniert die Ausführung der kontinuierlichen Anfragen und verwaltet ihre Zustände. Sie sendet i. d. R. atomare Ereignisse, wenn kontinuierliche Anfragen hinzugefügt werden, ihren Zustand ändern oder vom System entfernt werden. Im Regelsatz definierte kontinuierliche Anfragen werden gesondert behandelt: sie liefern keine atomaren Ereignisse, sondern verarbeiten diese zu komplexen Ereignissen, die bspw. bestimmte Zustandsverläufe erkennen und melden können. Zusätzlich senden kontinuierliche Anfragen selbst atomare Ereignisse, um Veränderungen in der Datenverarbeitung mitteilen zu können: Fluktuationen in der Datenrate können zu änderungen in der Systemlast führen, sodass solche atomare Ereignisse wichtig werden.

Es ist notwendig, dass das System Möglichkeiten anbietet, den eigenen Systemzustand (und insbesondere die Systemlast) zu beobachten. Atomare Ereignisse können hier bspw. überschreitungen von Lastschwellwerten sein. Es ist dann für die kontinuierliche AC möglich, auf Schwellwerte zu reagieren, auch wenn keine neue kontinuierliche Anfrage gestartet wurde. Schließlich können andere Komponenten des Systems atomare Ereignisse generieren (bspw. der Nutzer oder basierend auf Netzwerksignalen).

Alle Ereignisse werden im ECA-Modell des kontinuierlichen AC ausgewertet. Dazu werden im Regelsatz u. a. Bedingungen spezifiziert. Diese können neben typischen Ausdrücken, Konstanten und Funktionen auch spezielle Ausdrücke bzgl. der Lastüberwachung beinhalten (bspw. Abfrage der aktuellen Systemlast).

Das Resultat der Auswertung ist eine Menge von Aktionen. Diese Aktionen werden von der AC ausgeführt, um das System (und damit die Systemlast) zu manipulieren. Ein typischer Ansatz zur Senkung der Systemlast besteht in der Zustandsänderung der Anfragen, indem die kontinuierliche AC Anfragen stoppt, pausiert, partiell ausführt oder ganz entfernt (analog zu den Zuständen in Abbildung 1). Im Netzwerk könnte die AC einzelne Anfragen an andere Maschinen abgeben. Es sollte jedoch berücksichtigt werden, die Ausführung so wenig wie möglich zu beschneiden. Deswegen sollten leichte Aktionen wie bspw. die partielle Ausführung bevorzugt werden. Nur in kritischen Situationen soll-

ten Anfragen gestoppt oder sogar ganz entfernt werden. Später können Regeln erkennen, ob Leistungsreserven wieder verfügbar sind. Je nach System können kontinuierliche Anfragen wieder gestartet oder – falls zuvor partiell ausgeführt – wieder alle Daten verarbeiten. Eine gestaffelte Vorgehensweise zur Reaktivierung der kontinuierlichen Anfragen kann hier ebenfalls nützlich sein. Anstatt eine gestoppte Anfrage wieder vollständig auszuführen, kann sie zunächst partiell ausgeführt werden. Dabei können weitere Rahmenbedingungen wie bspw. die Genauigkeit der Ergebnisse berücksichtigt werden.

3 Evaluation

Das vorgestellte Konzept wird anhand eines einfachen Szenarios evaluiert, mit dem gezeigt wird, wie mit beispielhaften Regeln eine Überlastung des Systems verhindern wird. Fokus dieser Arbeit ist es nicht, ein komplexes Regelwerk zu evaluieren, da diese von der Hardware, der Anwendungsdomäne sowie der eingesetzten Software abhängig ist.

Zur Umsetzung der kontinuierlichen AC wurde das Framework für DSMS Odysseus verwendet [AGG+12]. Seine komponentenbasierte Architektur erlaubt die Implementierung zusätzlicher Funktionen, ohne die Basiskomponenten zur Verarbeitung anpassen zu müssen. Die kontinuierliche AC mitsamt dem ECA-Modell und der Sprache zur Regeldefinition wurde dort als zusätzliche Komponente umgesetzt. Zur Laufzeit sendet Odysseus Ereignisse bzgl. der Ausführung der kontinuierlichen Anfragen und der Systemlast.

Die genutzte Maschine ist ein Notebook mit Windows 7 als Betriebssystem, hat 4 GB RAM und einen Intel Core 2 Duo CPU T9500 Prozessor mit 2,60 GHz. Neben Odysseus laufen auf der Maschine keine weiteren Applikationen. Es wurde bewusst eine relativ leistungsschwache Maschine ausgewählt, um die Effekte der Lastkontrolle besser hervorheben zu können.

Während der Ausführung von Odysseus wird alle zehn Sekunden eine neue kontinuierliche Anfrage gestellt und ausgeführt (ohne die alten kontinuierlichen Anfragen zu stoppen). Dies geschieht bis zu einer Menge von 20 kontinuierlichen Anfragen. Damit wird die Lastgrenze der genannten Maschine erreicht. Für die Reproduzierbarkeit der Evaluation ist jede Anfrage gleich, Optimierungen sind deaktiviert. Die Anfrage besteht hier aus speziellen Benchmarkoperatoren und erfüllt lediglich den Zweck, reproduzierbar eine bestimmte, konstante Prozessorlast zu erzeugen. Durch das permanente Hinzufügen neuer Anfragen erhöht sich mit der Zeit die gesamte Systemlast.

Zur Laufzeit wird die Prozessorauslastung alle drei Sekunden mittels systeminterner Methoden gemessen. Ein Wert von 100% entspricht dabei einer vollen Auslastung. Da ausreichend Arbeitsspeicher vorhanden ist und um die Evaluationsergebnisse einfach zu halten, wird auf eine Messung der Belegung des Arbeitsspeichers verzichtet. Sie wäre jedoch im Rahmen des vorgestellten Konzeptes problemlos möglich. Weiterhin werden im Laufe der Experimente Latenz und Datenrate gemessen. *Latenz* beschreibt die Zeit, die benötigt wird, um mit Hilfe eines eingehenden Datenelements ein Ergebnis zu produzieren. Das bedeutet für dieses Szenario, wie lange ein Element im System verbleibt. *Datenrate* beschreibt die Menge an verarbeiteten Datenelementen pro Sekunde.

3.1 Ergebnisse

Das erste Experiment soll zeigen, dass durch das ständige Hinzufügen neuer kontinuierlicher Anfragen die Maschine (und damit Odysseus) überlastet werden kann, wenn die kontinuierliche AC nicht eingreift. Das Resultat der regelmäßigen Messung der Prozessorauslastung ist in Abbildung 3 als gestrichelte Linie zu sehen.

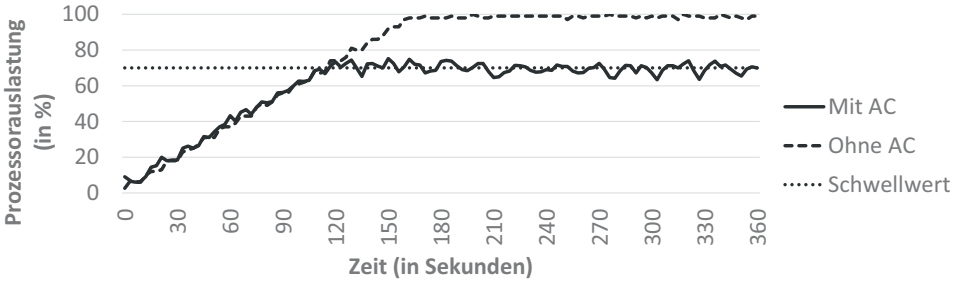

Abbildung 3: Prozessorauslastungen ohne Regeln (gestrichelte Linie) und mit (durchgezogene Linie)

Die Last steigt mit zunehmender Menge an aktiven kontinuierlichen Anfragen, bis schließlich nach ungefähr 150 Sekunden das Maximum an Prozessorlast erreicht wurde und Odysseus als überlastet gilt.

In einem zweiten Experiment soll die kontinuierliche AC eingesetzt werden, um die Überlastsituation zu vermeiden. Dazu wurde mittels CADL ein Regelsatz bestehend aus mehreren Regeln und einem komplexen Ereignis definiert. Der Regelsatz stoppt neue kontinuierliche Anfragen, wenn die aktuelle Prozessorlast 70% übersteigt. 30% der Leistung soll als Reserve vorgehalten werden (bspw. für Lastspitzen). Zusätzlich wird zwischen wichtigen kontinuierlichen Anfragen (Priorität $>= 10$) und unwichtigen (Priorität < 10) unterschieden. Erst sollen niedrig priorisierte Anfragen gestoppt werden, bevor es hoch priorisierte Anfragen betrifft. Können Anfragen später wieder reaktiviert werden, sollen hoch priorisierte Anfragen bevorzugt werden. Die Regeldefinition beginnt wie folgt:

```
ON EVENT QueryActivatedEvent e
IF curCpuLoad() > 70 AND e.query.prio < 10
THEN stopQuery( e.query )
```

Damit werden niedrig priorisierte Anfragen sofort gestoppt, wenn die Prozessorlast aktuell 70% übersteigt. Bei neuen hoch priorisierten Anfragen muss mittels zwei weiteren Regeln anders vorgegangen werden:

```
ON EVENT QueryActivatedEvent e
IF curCpuLoad()>70 AND e.query.prio >= 10
   AND existsQuery(state=ACTIVE AND prio<10)
THEN stopQuery(selectRandomQuery(state=ACTIVE AND prio<10))

ON EVENT QueryActivatedEvent e
IF curCpuLoad()>70 AND e.query.prio>=10
   AND NOT existsQuery(state = ACTIVE AND prio < 10)
THEN stopQuery(e.query)
```

Bei neuen hoch priorisierten Anfragen wird anstelle der neuen Anfrage eine bereits aktive niedrig priorisierte Anfrage gestoppt (erste Regel). Ist eine solche Anfrage nicht vorhanden (oder alle sind bereits gestoppt), muss die hoch priorisierte Anfrage gestoppt werden

(zweite Regel). Die Funktion `existsQuery` überprüft, ob eine Anfrage im System existiert, die das genannte Prädikat erfüllt (hier: ob eine Anfrage aktiv ist und eine Priorität kleiner als zehn besitzt). Die Funktion `selectRandomQuery` funktioniert ähnlich, außer dass aus der Menge der kontinuierlichen Anfragen, welche das Prädikat erfüllen, genau eine zufällig ausgewählt und zurückgegeben wird.

Werden Anfragen gestoppt, sinkt die Systemlast, sodass es auch die Chance gibt, zuvor gestoppte Anfragen wieder zu starten. Sei dazu `TryRestartQueryEvent` ein Event, welches alle 10 Sekunden aktiviert wird:

```
ON EVENT TryRestartQueryEvent e
IF curCpuLoad()<70 AND existsQuery(state=INACTIVE AND prio>=10)
THEN startQuery(selectRandomQuery(state=INACTIVE AND prio>=10))

ON EVENT TryRestartQueryEvent e
IF curCpuLoad()<70 AND existsQuery(state=INACTIVE AND prio<10)
   AND NOT existsQuery(state=INACTIVE AND prio>=10)
THEN startQuery(selectRandomQuery(state=INACTIVE AND prio<10))
```

Das komplexe Ereignis `TryRestartQueryEvent` wird alle zehn Sekunden automatisch ausgelöst, sodass die beiden Regeln regelmäßig unabhängig vom Zustand des Systems geprüft werden. Die erste Regel besagt, dass bei einer niedrigen Prozessorlast (unter 70%) eine hoch priorisierte Anfrage gestartet werden kann, falls eine solche Anfrage gestoppt im System vorliegt. Ist dies nicht der Fall und es existieren gestoppte niedrig priorisierte Anfragen, wird eine davon gestartet (zweite Regel). Bei der Durchführung des Experiments sind die ersten 14 Anfragen niedrig priorisiert. Anschließend werden sechs hoch priorisierte Anfragen hinzugefügt. Das Resultat der Messungen der Prozessorlast im zweiten Experiment ist in Abbildung 3 als durchgezogene Linie dargestellt.

Die gepunktete Linie beschreibt die 70%-Marke. Man erkennt, dass analog zum Vorversuch die Prozessorlast mit zunehmender Menge an kontinuierlichen Anfragen steigt, sich jedoch um den Wert 70% einpendelt. Dies hat den Grund, dass ab 70% durch die ersten Regeln neue Anfragen nicht mehr aktiv werden. Kommen später die hoch priorisierten Anfragen hinzu, werden andere niedrig priorisierte Anfragen gestoppt, um die Prozessorlast um 70% zu halten. Fällt die Last unter 70%, werden gestoppte Anfragen wieder ausgeführt, wobei hoch priorisierte Anfragen bevorzugt werden. Durch die zufällige Auswahl einer Anfrage wird über die Zeit jede Anfrage irgendwann ausgeführt.

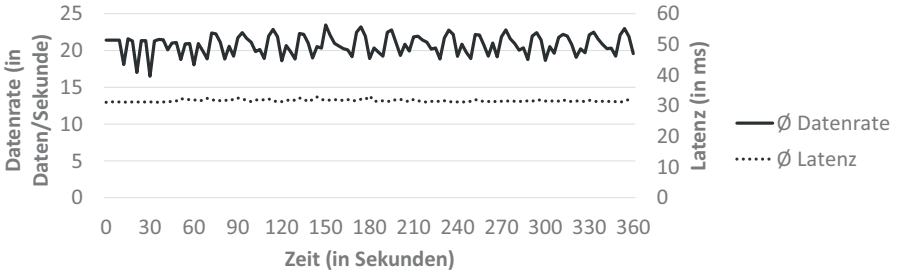

Abbildung 4: Verlauf der durchschnittlichen Datenrate und Latenz mit Regeldefinition.

Abbildung 4 zeigt die gemessenen Datenraten und Latenzen im zweiten Experiment. Die durchschnittliche Datenrate der Anfragen (durchgezogene Linie) bleibt auf gleichem Ni-

veau. Weiterhin ist die Latenz stabil. Das bedeutet, dass das System die Verarbeitung weiter effizient und effektiv fortsetzen kann. In realen Szenarien sollten jedoch andere Aktionen als das Stoppen gewählt werden (bspw. die Auslagerung im Netzwerk).

Die beiden Experimente zeigen, dass das hier vorgestellte Konzept seine Aufgabe der Lastkontrolle erfüllen kann: während noch im ersten Versuch ohne Einsatz des kontinuierlichen AC schnell eine Überlastsituation entstand, konnte diese im zweiten Versuch mittels einer relativ einfachen Regeldefinition abgewendet werden.

4 Verwandte Arbeiten

Wie bereits zu Beginn erwähnt, wird die AC in vielen konventionellen DBMS dann eingesetzt, wenn eine neue Anfrage ausgeführt werden soll [HSH07]. Diese Vorgehensweise der AC ist in vielen DBMS ausreichend, erlaubt jedoch nicht eine einfache und flexible Regeldefinition wie in dieser Arbeit vorgeschlagen.

Für DSMS werden in der Literatur hauptsächlich konkrete Ansätze zur Lastreduktion vorgestellt und diskutiert. Ziel ist es oft, eine maximale Genauigkeit mit minimalem Ressourcenverbrauch zu erreichen, wie es bspw. ausführlich in [ABB+04] beschrieben wird. Im Fokus der Literatur steht hauptsächlich das konkrete Problem des begrenzten Arbeitsspeichers. In [BBD+02] werden diverse Techniken beschrieben. Mittels des Fensteransatzes wird nur ein Teil des Datenstroms als gültig markiert (und verarbeitet). Mittels Sampling und Synopsen werden Datenströme verdichtet. Letztere werden bspw. in STREAM eingesetzt [MWA+03]. Load Shedding [TcZ+03] wird in vielen DSMS erfolgreich eingesetzt und wird bereits in dieser Arbeit als ein möglicher Zustand einer kontinuierlichen Anfrage betrachtet. Kalman-Filter werden genutzt, um zukünftige Systemlasten vorherzusagen um damit präventiv Maßnahmen ergreifen zu können [JCW04]. Mittels einer intelligenten Operatorreihenfolge lassen sich Systemlasten ebenfalls reduzieren [BBD+04].

5 Zusammenfassung

In Datenbanksystemen ist es wichtig, dass eine reibungslose Verarbeitung der Daten jederzeit sichergestellt ist. Überlastungen des Systems können zu Störungen und Ausfällen führen. Für DBMS existieren bereits Komponenten zur Lastkontrolle, die sogenannte Admission Control (AC). Jedoch existieren Anwendungsszenarien, die eine kontinuierliche und zeitnahe Verarbeitung erfordern. Das bedeutet, dass eine kontinuierliche Anfrage permanent ausgeführt werden muss, um sofort auf eintreffende Datenelemente einer aktiven Datenquelle reagieren zu können.

Für kontinuierliche Anfragen ist eine konventionelle AC nicht ausreichend: Sie muss nicht nur Anfragen zu Beginn, sondern auch während ihrer Ausführung prüfen, da durch die kontinuierliche Verarbeitung sich die Systemlast ständig ändern kann. Durch die Vari-

anz an Datenquellen, verschiedenen DBMS, DSMS und unterschiedlichen Hardwarekapazitäten ist eine Beschreibung für ein allgemeingültiges Verhalten nicht sinnvoll.

In dieser Arbeit wurde eine *kontinuierliche AC* auf Basis eines ECA-Modells vorgestellt, welche in Systemen mit kontinuierlichen Anfragen eingesetzt werden kann, um die dortige Systemlast permanent zu kontrollieren und ggfs. zu regulieren. Vom System gesendete, atomare Ereignisse können mittels Regeln ausgewertet werden und so zu gezielten Reaktion zur Lastkontrolle führen. Durch die vorgestellte Sprache CADL lassen sich flexibel komplexe Ereignisse und Regelsätze definieren, die an die speziellen Anforderungen und Rahmenbedingungen für kontinuierliche Anfragen angepasst sind. Die Evaluation zeigt, dass das Konzept praktikabel, umsetzbar und funktionsfähig ist.

Literatur

[ABB+04] Arvind Arasu, Brian Babcock, Shivnath Babu, Jon McAlister und Jennifer Widom. Characterizing Memory Requirements for Queries over Continuous Data Streams. *ACM Trans. Database Syst.*, 29(1):162–194, Marz 2004.

[AGG+12] H.-Jürgen Appelrath, Dennis Geesen, Marco Grawunder, Timo Michelsen und Daniela Nicklas. Odysseus: a highly customizable framework for creating efficient event stream management systems. DEBS '12, Seiten 367–368, New York, NY, USA, 2012. ACM.

[BBD+02] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani und Jennifer Widom. Models and issues in data stream systems. PODS '02, Seiten 1–16, New York, NY, USA, 2002. ACM.

[BBD+04] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani und Dilys Thomas. Operator Scheduling in Data Stream Systems. *The VLDB Journal*, 13(4):333–353, Dezember 2004.

[CKSV08] M. Cammert, J. Kramer, B. Seeger und S. Vaupel. A Cost-Based Approach to Adaptive Resource Management in Data Stream Systems. *Knowledge and Data Engineering, IEEE Transactions on*, 20(2):230–245, Feb 2008.

[HSH07] Joseph M. Hellerstein, Michael Stonebraker und James Hamilton. Architecture of a Database System. *Foundations and Trends in Databases*, 1(2):141–259, 2007.

[JCW04] Ankur Jain, Edward Y. Chang und Yuan-Fang Wang. Adaptive Stream Resource Management Using Kalman Filters. SIGMOD '04, Seiten 11–22, New York, NY, USA, 2004. ACM.

[Krä09] Jürgen Krämer. Continuous Queries over Data Streams - Semantics and Implementation. Jgg. 144 of *BTW '09*, Seiten 438–448. GI, 2009.

[MWA+03] Rajeev Motwani, Jennifer Widom, Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Gurmeet Manku, Chris Olston, Justin Rosenstein und Rohit Varma. Query processing, resource management, and approximation in a data stream management system. CIDR, 2003.

[TcZ+03] Nesime Tatbul, Uğur Çetintemel, Stan Zdonik, Mitch Cherniack und Michael Stonebraker. Load Shedding in a Data Stream Manager. VLDB '03, Seiten 309–320. VLDB Endowment, 2003.