Koordinierte zyklische Kontext-Aktualisierungen in Datenströmen

Dennis Geesen[1], André Bolles[1], Marco Grawunder[1], Jonas Jacobi[1], Daniela Nicklas[2], H.-Jürgen Appelrath[1]

[1]Informationssysteme und [2]Datenbank- und Internettechnologien
Department für Informatik
Universität Oldenburg
D-26121 Oldenburg

Abstract: Kontextsensitive Anwendungen benötigen ein möglichst exaktes Modell der Umgebung. Zur Ermittlung und regelmäßigen Aktualisierung dieses Kontextmodells werden typischerweise Sensordaten verwendet. Datenstrommanagementsysteme (DSMS) bilden die ideale Basis, um mit den durch die Sensoren generierten, potentiell unendlichen Datenströmen umzugehen. Leider bieten bisherige DSMS keine native Unterstützung für dynamische Kontextmodelle. Insbesondere die bei der Aktualisierung entstehenden Zyklen im Anfrageplan bedürfen einer besonderen Koordination, um Aktualität und Konsistenz des Kontextmodells zu gewährleisten. Diese Arbeit präsentiert eine Lösung, die einen Broker zur Koordination der verschiedenen Zugriffe auf das Kontextmodell als neuen Operator im DSMS einführt. Wir zeigen dazu eine semantische Beschreibung und eine abstrakte Implementierung des Brokers.

1 Einleitung

Zur effektiven Verwendung von Sensoren in kontextsensitiven Anwendungen ist die Verwaltung eines Kontextmodells notwendig. Im Bereich der Fahrerassistenzsysteme (FAS) beispielsweise spiegelt das Kontextmodell die derzeitige Umgebung des Fahrzeugs wider. Sensoren erzeugen dabei in einer hohen Rate eine große Menge an Daten, die zeitnah verarbeitet werden muss. Zur zeitnahen Verarbeitung solcher Daten können Datenstrommanagementsysteme (DSMS) eingesetzt werden. Diese adaptieren Anfrageverarbeitungsmechanismen, wie sie aus Datenbankmanagementsystemen (DBMS) bekannt sind und ermöglichen so eine flexible Verarbeitung potentiell unendlicher Datenströme [See04].

Bei der Umsetzung kontextsensitiver Anwendungen wie FAS muss jedoch das Kontextmodell verwaltet werden. Insbesondere die koordinierte Aktualisierung durch mehrere Sensoren spielt hier eine entscheidende Rolle. Hierbei entstehen mehrere Zyklen, indem das aktuelle Kontextmodell gelesen, durch einen Sensormesswert aktualisiert und dann wieder gespeichert wird. Diese zyklischen Kontext-Aktualisierungen müssen zeitlich korrekt ausgeführt werden, sodass das Kontextmodell fortlaufend und nicht beliebig aktualisiert wird und damit Aktualität und Konsistenz des Kontextmodells gewährleistet werden. Dabei ist jedoch nicht die zeitliche Reihenfolge des ersten Zugriffs eines Zyklus auf das Kontextmo-

dell wichtig, sondern die zeitliche Reihenfolge, in der die Sensormesswerte erfasst worden sind. Da Zyklen jedoch parallel verarbeitet werden und verschiedene Latenzen besitzen können, ist die Reihenfolge der Zugriffe auf das Kontextmodell nicht zwingend synchron mit der Reihenfolge der Messwerterfassung. Obwohl im ersten Ansatz DBMS und deren Transaktionskontrolle für die Koordinierung sinnvoll erscheinen, beruht die Transaktionskontrolle auf der zeitlichen Reihenfolge des ersten Zugriffs auf das Kontextmodells und nicht wie gewünscht auf die zeitliche Reihenfolge der Messwerterfassung, also der Daten selbst. Aus diesem Grund sind sowohl die – ohnehin nicht für kontinuierliche Daten adäquaten – klassischen DBMS als auch die schnelleren Hauptspeicher-DBMS für dieses Problem nicht geeignet.

Alle lesenden und schreibenden Zugriffe müssen anhand der Daten koordiniert werden. Insbesondere stellen zyklische Aktualisierungen eine besondere Herausforderung dar, da ein Zyklus komplett beendet sein muss, bevor ein neuer Zugriff gestattet werden darf. Die hochdynamische Verwaltung eines Kontextmodells in flexiblen DSMS wurde bisher nicht betrachtet. Aus diesem Grund stellen wir in dieser Arbeit einen Broker vor, der als Datenstrom-Operator das Speichern, sowie den koordinierten und zeitlich korrekten Zugriff auf das Kontextmodell erlaubt. Dazu stellt Kapitel 2 zunächst verwandte Arbeiten vor. Kapitel 3 führt das verwendete Modell und System des Datenstrommanagements ein. Im Anschluss werden in Kapitel 4 die Motivation und Anforderungen an eine Umsetzung erläutert. Darauf aufbauend beschreibt Kapitel 5 die Realisierung des Broker-Operators, indem zum einen ein logischer Operator für die Semantik und zum anderen ein physischer Operator für eine abstrakte Implementierung vorgestellt wird. Wir betrachten den Ansatz in Kapitel 6 in einem Experiment und geben in Kapitel 7 eine Übersicht und ein abschließendes Fazit.

2 Verwandte Arbeiten

Die Verarbeitung von Datenströmen auf Basis von Anfrageverarbeitungsmechanismen, die ähnlich zu denen in DBMS sind, wurde bisher unter anderem in Prototypen wie Aurora [ACc+03], Borealis [AAB+05], STREAM [ABB+03], PIPES [KS05] oder auch inzwischen in kommerziellen Produkten wie RTM [Rea10] oder SPADE [GAW+08] umgesetzt. Auch für die Verwaltung von Kontextmodellen in kontextsensitiven Anwendungen existieren Middlewares, wie z.B. Gaia [RHC+02], MAIS [CCMP06] oder Nexus [CEB+09]. Dabei integriert Nexus auch Datenstrommanagement (DSM)-Konzepte, auch wenn hier aufgrund der geringen Datenrate ein korrekter Zugriff auf das Kontextmodell auch dann gewährleistet werden kann, wenn dies in einem DBMS gespeichert wird. Das hier vorgestellte Konzept betrachtet jedoch eine hohe Datenrate und verwaltet das Kontextmodell daher direkt im DSMS.

Die hier als Motivation dienende Objektverfolgung wurde im Projekt STREAM erprobt, indem mehrere Datenstrom-Elemente den Bewegungsverlauf eines Objektes beschreiben [PS04]. Weitere Arbeiten beschäftigen sich ebenfalls mit kontextbezogenen Daten, wie beispielsweise [HJ04] mit ortsbasierten Anfragen oder [MA08] mit spatio-temporalen Daten. Hierbei wurden jedoch keine Kontextmodelle verwaltet. Die Betrachtung von Zyklen

in Datenströmen wurde unter anderem von [WRML08], [DS00] oder [GADI08] behandelt. Hier wurden aber lediglich rekursive Vereinigung, rekursiv verschachtelte XML-Fragmente oder Kleenesche Hüllen betrachtet, bei denen keine koordinierte Verarbeitung verschiedener Zyklen notwendig war. Des Weiteren führt [CGM09] einen neuen Operator ein, um zu erkennen wann eine transitive Hülle bei einer rekursiven Vereinigung abgeschlossen ist. Allerdings wird auch hier keine Koordinierung der zeitlichen Reihenfolge der Daten durchgeführt. [GBz06] beschäftigen sich mit Synchronisation und Konfliktserialisierbarkeit beim Einfügen neuer Elemente innerhalb eines Fensters, während Daten aus dem Fenster noch gelesen werden und zeigen hierzu eine entsprechende Scheduling-Strategie. Hierbei werden bereits geordnete lesende und geordnete schreibende Zugriffe betrachtet. In dieser Lösung werden jedoch zum einen unsortierte Zugriffe und zum anderen Zyklen berücksichtigt. Abschließend spielen hier auch Konzepte der Transaktionskontrolle in DBMS eine Rolle (s. [EN09] für eine Einführung).

3 Datenmodell und Anfrageverarbeitung

Ein Datenstrom ist eine potentiell unendliche Folge von Daten, die in der Regel als relationale Tupel repräsentiert werden. Ein Datenstrommanagementsystem (DSMS) erlaubt eine flexible und effiziente Verarbeitung solcher kontinuierlich auftretenden Daten, indem ähnlich zu Datenbankmanagementsystemen (DBMS) deklarative Anfragen verwendet werden. Zurzeit gibt es diverse Umsetzungen von DSMS, die meist in der Forschung als Prototyp eingesetzt werden oder aus einem solchen hervorgegangen sind (vgl. Abschnitt 2). Neben den genannten Systemen gibt es mit Odysseus [BGJ+09] ein Framework für DSMS, welches leicht erweiterbar und anpassbar ist. Dies erlaubt unter anderem eine einfache und flexible Evaluation neuer Forschungskonzepte. Die Architektur von Odysseus besteht, wie Abbildung 1 zeigt, aus mehreren Komponenten, die jeweils feste Funktionen (Fixpunkt) besitzen, die um sogenannte Variationspunkte erweitert werden [BGJ+09].

Die Anfrageverarbeitung kann in vier Phasen betrachtet werden. Zunächst sorgt die Übersetzungskomponente (Translate) dafür, dass eine deklarative Anfrage in einen Anfrageplan übersetzt wird. Hierzu bedient sich die Übersetzungskomponente einer logischen Algebra, die Semantik und Aufbau des Anfrageplans definiert. Der daraus entstandene logische Anfrageplan wird im nächsten Schritt durch die Restrukturierungskomponente (Rewrite) unter Verwendung von Rewrite-Regeln optimiert. Dieser Plan wird in einem weiteren Schritt der Transformationskomponente (Transform) übergeben. Diese übersetzt den Plan auf Grundlage von Transform-Regeln und einer physischen Algebra in einen physischen Anfrageplan. Der physische Anfrageplan wird dann der Ausführung (Execute) übergeben und gegebenenfalls mit vorhandenen Anfrageplänen verbunden. Die linke Seite in Abbildung 1 zeigt einen gesamten Anfrageplan, wobei die Operatoren als *Pipe* dargestellt werden. Die zuvor genannten Variationspunkte sind beispielsweise durch die Möglichkeit gegeben, dass die Übersetzungskomponente um neue Sprachen erweitert werden kann. Des Weiteren können Übersetzung und Transformation um neue Algebra-Operatoren erweitert werden. Analog können auch die Restrukturierungs- und Transformationskomponente

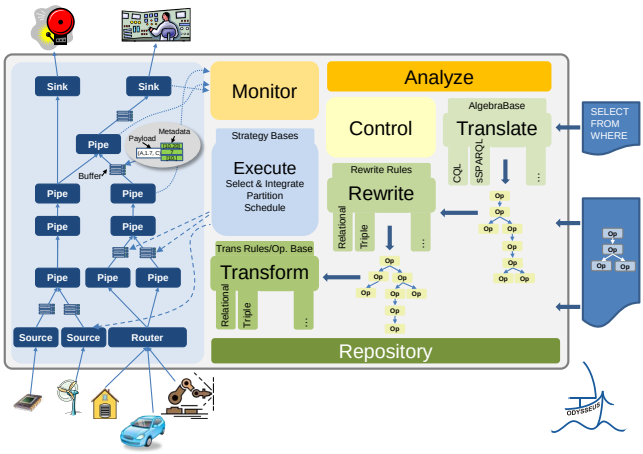

Abbildung 1: Architektur von Odysseus [BGJ+09]

um neue Regeln ergänzt werden. Außerdem lassen sich zum Beispiel neue Datenmodelle, Metadaten oder auch neue Scheduling-Strategien integrieren.

Der genannte Anfrageplan basiert auf einer Verknüpfung von Operatoren, die auf der logischen bzw. physischen Algebra basieren und daher entsprechend einen logischen bzw. physischen Anfrageplan darstellen. Hierbei dient die logische Algebra i. A. lediglich dazu, die genaue Semantik der einzelnen Operatoren festzulegen und auf Grundlage der erweiterten relationalen Algebra Optimierungen zu ermöglichen. Dabei betrachtet die logische Algebra alle Datenströme als temporale Multimengen und Operatoren stellen Abbildungen zwischen diesen Multimengen dar. Da hierbei die gesamte Menge bekannt sein muss, würden Operatoren entsprechend unendlich warten und blockieren. Daher hat die physische Algebra eine andere Sicht auf die Datenströme. Hier unterscheidet man zwischen nicht-blockierenden und blockierenden Operatoren. Nicht-blockierende Operatoren müssen ihre Eingabe nicht komplett konsumieren, um eine Ausgabe zu erzeugen. So kann z.B. eine Selektion direkt auf ein Element angewendet werden, indem es entweder weitergeleitet oder verworfen wird. Blockierende Operatoren hingegen sind meist zustandsbehaftete Operatoren, die zunächst die ganze Eingabe benötigen, um ein Ergebnis produzieren zu können. Beispielsweise kann eine Aggregation erst berechnet werden, wenn alle nötigen Elemente vorliegen. Um dieses blockierende Verhalten zu lösen, gibt es unter anderem den hier verwendeten Fenster-Ansatz, bei dem lediglich endliche Ausschnitte – in der Regel die aktuellsten Elemente – des Datenstroms betrachtet werden. Hierbei bekommt jedes Datenstromelement eine Gültigkeit, die entweder durch das Anheften eines Gültigkeitsintervall [KS05] oder durch Markierung mit einem positiven bzw. negativen Marker [GAE06] umgesetzt wird. Blockierungen können nun dadurch aufgelöst werden, dass in den einzelnen Operatoren nur noch Elemente gemeinsam betrachtet werden, die gleichzeitig gültig sind, sich also im selben Fenster befinden.

4 Motivation und Anforderungen

Kontextsensitive Anwendungen treffen ihre Entscheidung typischerweise auf Grundlage eines Kontextmodells, das die derzeitige Umgebung widerspiegelt. Dazu ist es notwendig, dass die aktuell gültige Instanz des Kontextmodells gespeichert und durch neue Messdaten fortlaufend aktualisiert und der Anwendung zu Verfügung gestellt wird. Hierzu zeigt Abbildung 2 eine beispielhafte Architektur, in der mit zwei Sensoren kontinuierlich Objekte in der Umgebung erfasst werden. Angenommen ein Fahrzeug sei mit einem Radarsensor

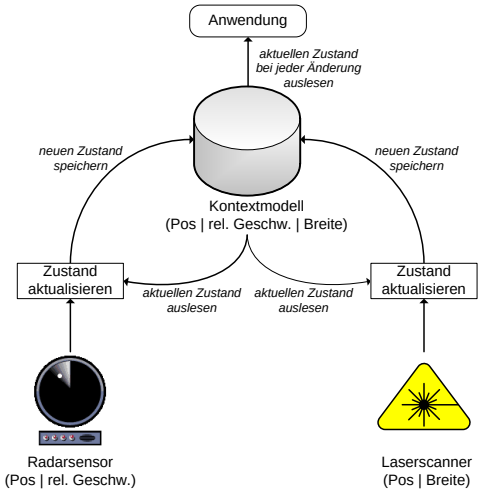

Abbildung 2: Beispielarchitektur

und einem Laserscanner ausgestattet. Der Radarsensor liefert die Position sowie die relative Geschwindigkeit detektierter Objekte. Der Laserscanner dagegen liefert die Position sowie die Breite detektierter Objekte. Anhand dieser Daten wird ein Kontextmodell mit allen drei Eigenschaften, Position, relative Geschwindigkeit und Breite aufgebaut. Wenn neue Objektdetektionen im System eintreffen, wird das aktuelle Kontextmodell aus dem Speicher gelesen. Daraufhin wird das Kontextmodell anhand der neuen Objektdetektionen im entsprechenden Teil des Anfrageplans (links Radar oder rechts Laserscanner) aktualisiert und anschließend wieder zurück in den Speicher geschrieben. Dabei ist die zeitliche Reihenfolge der Aktualisierungen an der zeitlichen Reihenfolge der erfassten Daten auszurichten. Werden bspw. Objekte vom Radarsensor zum Zeitpunkt t_1 und vom Laserscanner zum Zeitpunkt t_2 mit $t_1 < t_2$ erfasst, so ist das Kontextmodell zunächst mit den Objekten aus dem Radarsensor und anschließend mit denen aus dem Laserscaner zu aktualisieren. Andernfalls können Informationen verloren gehen oder bspw. in Vorhersagefunktionen falsch berechnet werden. Ist es in einer Anwendung möglich, dass Objekte von mehreren Sensoren gleichzeitig erkannt werden, so muss das Kontextmodell in diesem Fall durch einen speziellen Aktualisierungs-Operator, der mehrere Messwerte gleichzeitig verarbeiten kann, in einem eigenen Zyklus aktualisiert werden. Bei unsynchronisierten Sensoren tritt dieser Fall jedoch i. d. R. nicht auf, so dass er im Folgenden nicht näher betrachtet

wird. Da Sensoren nicht zwangsläufig synchronisiert sind, kann über die Reihenfolge des Erfassens der Daten im Vorfeld keine Aussage getroffen werden. Damit benötigt man für die zeitlich koordinierte Aktualisierung des Kontextmodells eine Art Transaktionskontrolle, die dynamisch anhand der eintreffenden Daten entscheiden kann, welcher Zyklus zur Aktualisierung des Kontextmodells genutzt wird. Die Transaktionskontrolle eines klassischen oder eines Hauptspeicher-DBMS kann hierzu, wie bereits erwähnt, nicht eingesetzt werden, da das entsprechende DBMS keine Kenntnis über die zeitliche Reihenfolge der Daten erhalten würde. Man benötigt also eine native Transaktionskontrolle für das Kontextmodell innerhalb des eingesetzten DSMS. Das in Abbildung 2 gezeigte Beispiel kann dabei als Anfrageplan in einem DSMS umgesetzt werden, wie Abbildung 3 zeigt. Dabei

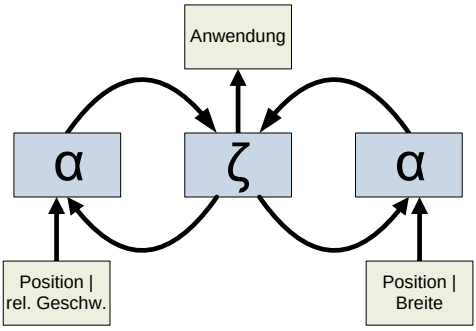

Abbildung 3: Umsetzung des Beispiels als Anfrageplan

wird das Kontextmodell durch einen Broker-Operator (ζ) verwaltet und jeweils durch Aktualisierungs-Operatoren (α) aktualisiert. Die Aktualisierungen werden dabei nicht von dem Broker selbst übernommen sondern in zusätzlichen Aktualisierungs-Operatoren ausgelagert, da für jeden Sensor gegebenenfalls verschiedene Aktualisierungs-Algorithmen von einfachen Berechnungen bis zur Verwendung von Prädiktionsfunktionen existieren. Die Verwendung von Aktualisierungs-Operatoren schafft somit eine höhere Flexibilität und erlaubt es u.a. auch, dass zwischen Broker-Operator und Aktualisierungs-Operator ggf. weitere Operatoren existieren, die das übertragene Kontextmodell u.U. modifizieren. Aus diesen Gründen sei an dieser Stelle von der konkreten Funktionalität von α abstrahiert, um lediglich auf die genaue Funktionsweise des Brokers einzugehen.

Anforderungen und Konzepte

Wird die Verwaltung eines Kontextmodells wie oben beschrieben durch ein DSMS umgesetzt, so ergeben sich dadurch Anforderungen an den Broker. Zum einen muss der Broker Mechanismen bereitstellen, die einen lesenden und schreibenden Zugriff auf das Kontextmodell erlauben. Zum anderen muss der Broker Eigenschaften eines DSMS, insbesondere die zeitliche Ordnung und die push-basierte Verarbeitung, berücksichtigen. Im Folgenden werden daher die Anforderungen und die jeweiligen Konzepte anhand der verschiedenen Zugriffsmöglichkeiten beschrieben und entsprechende Konzepte vorgestellt, die eine Inte-

gration in ein DSMS erlauben.

Schreibender Zugriff Ein schreibender Zugriff ist über eingehende Datenströme, den sogenannten *Update Streams*, mit dem Broker Operator möglich. Ein Update Stream liefert Datenelemente, die im Kontextmodell gespeichert werden sollen. Solche Elemente können zum einen Aktualisierungen des Kontextmodells zum anderen aber auch neue Elemente sein, die noch nicht im Kontextmodell vorhanden sind. Beim Eintreffen neuer Daten über einen Update Stream wird der aktuelle Zeitfortschritt festgestellt und alte, nicht mehr gültige Daten aus dem Kontextmodell entfernt. Neue Elemente werden einfach hinzugefügt. Aktualisierungen ersetzen jeweils ihre Vorgängerversion.

Lesender Zugriff Der lesende Zugriff auf den Broker kann in zwei unterschiedliche Zugriffsarten unterteilt werden. Zum einen gibt es den kontinuierlich lesenden Zugriff, der über sogenannte *Observation Streams* realisiert wird. Jede Änderung des Kontextmodells wird über diese Observation Streams bspw. an nachfolgende Anwendungen weitergeleitet. Zum anderen gibt es aber auch den einmalig lesenden Zugriff, der bei Vorliegen neuer Messwerte am Aktualisierungs-Operator das jeweils aktuelle Kontextmodell zurückliefern soll. Da dieses Kontextmodell jedoch am Aktualisierungs-Operator vorliegt, kann der Broker nicht wissen, wann das aktuelle Kontextmodell ausgeliefert werden soll. Daher werden für diese Art des Zugriffs zwei Arten von Datenströmen eingeführt. Über sogenannte *Request Streams* kann bei Vorliegen neuer Messwerte aus den Sensoren das jeweils aktuelle Kontextmodell beim Broker angefragt werden. Dieses wird dann über sogenannte *Response Streams* an den Aktualisierungs-Operator weitergeleitet.

Für einen Aktualisierungszyklus erhält man damit die in Abbildung 4 gezeigten Datenströme. Trifft hier ein neuer Messwert bei dem Aktualisierungs-Operator ein, so kann die-

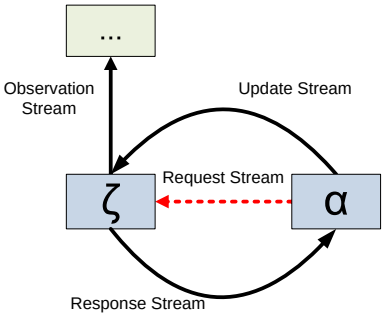

Abbildung 4: Stromtypen

ser einen Request an den Broker schicken, indem er ein entsprechendes Element über den Request Stream übergibt. Dann kann der Broker diesen Request bearbeiten und schickt entsprechend das aktuelle Kontextmodell über den zugehörigen Response Stream an den Aktualisierungs-Operator. Dieser kann nun das Kontextmodell anhand des Messwertes

und des aktuellen Kontextmodells aktualisieren. Danach wird das Kontextmodell zurück an den Broker übergeben. Da sich hier nun noch andere, gegebenenfalls nicht mehr gültige, Elemente im Broker befinden, wird das gespeicherte Kontextmodell im Broker zunächst anhand der Aktualisierung bereinigt. Danach wird dann die Aktualisierung dem Kontextmodell hinzugefügt. Der Broker beinhaltet danach entsprechend wieder das gültige Kontextmodell. Erst jetzt kann ggf. ein anderer Zyklus vom Broker bedient werden.

Ein hier verwendeter Request über den Request Stream besteht nur aus einem Zeitstempel, da nur der Zeitpunkt des zugehörigen Messwertes im Aktualisierungs-Operator und nicht die Daten des Requests selbst für eine zeitliche Sortierung der Requests notwendig sind. Hierbei wäre es ebenso denkbar, dass neben dem Zeitstempel zusätzliche Daten übertragen werden, um bspw. nur einen Teil des Kontextmodells auszuwählen, der vom Broker als Antwort an den Aktualisierungs-Operator geschickt wird. Da dies jedoch stark von der Anwendung und den genutzten Aktualisierungs-Algorithmen abhängt und hier davon abstrahiert wird, wird das Kontextmodell als ein Gesamtes betrachtet.

5 Realisierung

Die Realisierung des Broker-Operators, der die im vorherigen Abschnitt genannten Anforderungen und Konzepte umsetzt, wird in zwei Ebenen betrachtet. Die Umsetzung als logischer Operator erlaubt eine semantisch korrekte Formulierung des Brokers und erlaubt dadurch u.a. Optimierungen auf Grundlage einer Algebra. Das logische Konzept kennt jedoch alle Daten im Voraus, was jedoch nicht effizient implementiert werden könnte, so dass der physische Broker-Operator eine abstrakte Implementierung auf Grundlage eines erweiterten Konzeptes darstellt.

5.1 Logischer Operator

Zur Definition der Semantik des Brokers wird ein logischer Broker-Operator auf Basis der logischen Algebra aus [Krä07] formal beschrieben. Sei dazu $\mathbb{S}^l$ die Menge aller logischen Datenströme. Dann ist ein Datenstrom $S^l_\tau \in \mathbb{S}^l$ als eine Multimenge von Elementen (e, t, n) definiert, wobei $e \in \Omega_\tau$ ein Tupel vom Typ τ ist. Des Weiteren ist $n \in \mathbb{N}$ mit $n > 0$ die Anzahl des Tupels e zum Zeitpunkt $t \in T$. Hierbei ist T die Menge aller Zeitstempel in einem diskreten und total geordneten Wertebereich $\mathbb{T} = (T, \leq)$. Somit gibt beispielsweise ein Element $(a, 4, 3)$ an, dass das Tupel a zum Zeitpunkt 4 dreimal vorkommt.

Sei der Broker $\zeta : (\mathbb{S}^l_\tau)^k \times \mathbb{F}_{next} \rightarrow (\mathbb{S}^l_\tau)^m$ mit $k, m \in \mathbb{N}, k, m \geq 1$ als eine Abbildung von k logischen Datenströmen und einer Auswahlfunktion auf m logische Datenströme definiert. Entsprechend wird ein Update Stream oder ein Request Stream mit $S^l_{in} \in (\mathbb{S}^l_\tau)^k$ und analog ein Observation oder Response Stream mit $S^l_{out} \in (\mathbb{S}^l_\tau)^m$ bezeichnet. Des Weiteren beschreibt eine Auswahlfunktion $f_{next} \in \mathbb{F}_{next}$ mit $f_{next} : T \rightarrow \mathcal{P}(S^l_{out})$ eine Abbildung von einem Zeitstempel auf eine Menge von ausgehenden Datenströmen. Sie entspricht der Umsetzung der Request Streams, indem sie zu einem Zeitpunkt t die Menge

von Response Streams oder Observation Streams liefert, die zu diesem Zeitpunkt bedient werden müssen. Hierzu sei $S^l_{req} \subseteq S^l_{in}$ die Menge der Request Streams aus allen eingehenden Datenströmen. Dann ist $R : S^l_{req} \rightarrow S^l_{out}$ eine Abbildung, die jedem eingehenden Request Stream einen Ausgangsdatenstrom zuordnet. Dann wäre

$$f_{next}(t) := \{\hat{S} \in S^l_{out} | \forall S \in R^{-1}(\hat{S}) : \exists (e,t,n) \in S\} \tag{1}$$

eine Auswahlfunktion, die eine Menge von Response Streams $\hat{S}$ liefert, zu denen im zugehörigen Request Stream $R^{-1}(\hat{S})$ ein Tupel (e,t,n) und damit ein Request vorliegt.

Seien S_1 bis S_k logische eingehende Datenströme und $\hat{S}_1$ bis $\hat{S}_m$ logische ausgehende Datenströme. Dann ist der Broker definiert durch

$$\begin{aligned}
\zeta_{f_{next}}(S_1,...,S_k) :=& (\hat{S}_1,...,\hat{S}_m) \text{ mit} \\
\hat{S}_j :=& \{(e,\hat{t},\hat{n}) | \exists X \subseteq S. \\
& X \neq \emptyset \quad \wedge \\
& X = \{(e,t,n) \in S | n(t) = n(\hat{t})\} \quad \wedge \\
& \hat{n} = \sum_{(e,t,n) \in X} n \quad \wedge \\
& S := \{(e,\hat{t},\overline{n}) | ((e,\hat{t},n_1) \in S_1 \wedge S_1 \notin S^l_{req} \vee n_1 = 0) \wedge ... \wedge \\
& \qquad ((e,\hat{t},n_k) \in S_k \wedge S_k \notin S^l_{req} \vee n_k = 0) \quad \wedge \\
& \qquad \overline{n} > 0 \quad \wedge \\
& \qquad \overline{n} = \sum_{i=1}^{k} n_i\} \quad \wedge \\
& \hat{S}_j \in f_{next}(\hat{t})\}
\end{aligned} \tag{2}$$

wobei $n(t)$ die Anzahl der Elemente des Datenstromes bis zum Zeitpunkt t ist, welche durch

$$n(t) = |\{(e,\widetilde{t},n) \in S | \widetilde{t} \leq t\}| \tag{3}$$

definiert ist.

Zu einem Zeitpunkt $\hat{t}$ werden alle eingehenden Datenströme $S_1,...,S_k$ zu einem Datenstrom S zusammengefasst, sofern es kein Tupel $(S_i, \widetilde{S}) \in R$ gibt, sodass es sich bei S_i nicht um einen Request Stream handelt. $\overline{n}$ gibt an, dass gleiche Tupel in unterschiedlichen Eingangsströmen aufsummiert werden. Aus diesem zusammengeführten Datenstrom S nimmt der Broker das Element (e,t,n), das mit $n(t)$ dieselbe Anzahl an Vorgängern hat, wie es zum aktuellen Zeitpunkt mit $n(\hat{t})$ gibt. Hiermit wird zeitlich betrachtet das letzte Element genommen, das vor $\hat{t}$ gültig ist. Dadurch erhält man in Bezug auf das Kontextmodell den letzten gültigen Zustand zum Zeitpunkt $\hat{t}$. Das so ausgewählte Tupel wird an alle ausgehenden Datenströme $\hat{S}_j$ übergeben, die von der Auswahlfunktion f_{next} bestimmt worden sind. Somit bekommen zum Zeitpunkt $\hat{t}$ alle Datenströme $\hat{S}_j$ die Tupel, die zuletzt gültig waren.

Beispiel

Im Folgenden hat der Broker drei eingehende Datenströme S_1, S_2, S_3 und S_4, sowie drei ausgehende Datenströme $\hat{S}_1$, $\hat{S}_2$ und $\hat{S}_3$. Seien die eingehenden Datenströme wie folgt definiert:

$$
\begin{array}{lll}
S_1 := & \{(c,1,1),(a,2,3),(a,3,3), & S_3 := \quad \{(b,2,2),(b,3,2),(a,4,1), \\
& (b,3,1),(a,4,3),(b,4,1), & \qquad (b,4,1),(c,4,1),(a,5,2), \\
& (c,4,1),(b,5,2),(b,6,2)\} & \qquad (b,5,1),(a,6,1),(c,6,2)\} \\
S_2 := & \{(c,1,1),(a,2,1),(a,5,1), & S_4 := \quad \{(b,3,1),(b,6,1)\} \\
& (b,6,1)\} &
\end{array}
$$

Dann zeigt Tabelle 1 den jeweiligen Zustand der vier Eingangsdatenströme S_1, S_2, S_3 und S_4 zu einem Zeitpunkt t.

t	$\mathbf{S_1}$	$\mathbf{S_2}$	$\mathbf{S_3}$	$\mathbf{S_4}$
1	$< c >$	$< c >$	$<>$	$<>$
2	$< a, a, a >$	$< a >$	$< b, b >$	$<>$
3	$< a, a, a, b >$	$<>$	$< b, b >$	$< b >$
4	$< a, a, a, b, c >$	$<>$	$< a, b, c >$	$<>$
5	$< b, b >$	$< a >$	$< a, a, b >$	$<>$
6	$< b, b >$	$< b >$	$< a, c, c >$	$< b >$

Tabelle 1: Beispiele für vier logische Datenströme

Seien weiter S_2 und S_4 Request Streams für die Ausgangsdatenströme $\hat{S}_1$ und $\hat{S}_2$, sodass nachfolgende Zuordnungen existieren: $R(S_2) = \hat{S}_1$ und $R(S_4) = \hat{S}_2$.

Dann kann unter Verwendung der Auswahlfunktion f_{next} der Broker-Operator auf die Eingangsdatenströme S_1, S_2, S_3 und S_4 und entsprechend auf die drei Ausgangsdatenströme $\hat{S}_1$, $\hat{S}_2$ und $\hat{S}_3$ angewendet werden. Tabelle 2 zeigt entsprechend zu den Eingaben das Ergebnis von $\zeta_{f_{next}}(S_1, S_2, S_3, S_4)$.

t	$\hat{\mathbf{S}}_\mathbf{1}$	$\hat{\mathbf{S}}_\mathbf{2}$	$\hat{\mathbf{S}}_\mathbf{3}$
1	$< c >$	$<>$	$< c >$
2	$< a, a, a, b, b >$	$<>$	$< a, a, a, b, b >$
3	$<>$	$< a, a, a, b, b, b >$	$< a, a, a, b, b, b >$
4	$<>$	$<>$	$< a, a, a, a, b, b, c, c >$
5	$< a, a, b, b, b >$	$<>$	$< a, a, b, b, b >$
6	$< a, b, b, c, c >$	$< a, b, b, c, c >$	$< a, b, b, c, c >$

Tabelle 2: Broker-Operation über die logischen Datenströme S_1, S_2, S_3, S_4

Wie zu sehen ist, vereinigt der Broker alle Elemente, die nicht aus einem Request Stream kommen. Zum Zeitpunkt $t = 2$ werden daher nur die Elemente $< a, a, a >$ aus dem Datenstrom S_1 und die Elemente $< b, b >$ aus dem Datenstrom S_3 zusammengeführt. Da zu diesem Zeitpunkt eine Anforderung $< a >$ von dem Request Stream S_2 vorliegt,

muss das Ergebnis $< a, a, a, b, b >$ auch an den Ausgangsdatenstrom $R(S_2) = \hat{S}_1$ geliefert werden. Daher ist die Auswahlfunktion zu diesem Zeitpunkt wie folgt definiert: $f_{next}(2) = \{\hat{S}_1, \hat{S}_3\}$. Hierbei kann man unter anderem erkennen, dass die Nutzdaten von Elementen aus Request Streams ignoriert werden, da lediglich die Zeitstempel von Interesse sind. Wie bereits in Abschnitt 4 genannt, wird dann entsprechend auch das ganze Kontextmodell wiedergegeben, indem keine Auswahl für einen Teil des Kontextmodells als Nutzdaten mitgegeben wird.

5.2 Physische Algebra

Eine direkte Implementierung des logischen Algebra-Operators ist nicht praktikabel, da sonst für jeden Zeitpunkt ein eigenes Element im Datenstrom verarbeitet werden müsste. Die physische Algebra (vgl. [Krä07]) betrachtet daher eine kompaktere Sichtweise auf Datenströme. Sei dazu $\mathbb{S}^p$ die Menge aller physischen Datenströme und $S^p_\tau \in \mathbb{S}^p$ ein physischer Datenstrom. Dann ist S^p_τ eine Menge von Elementen $(e, [t_s, t_e))$, wobei $e \in \Omega_\tau$ ein Element vom Typ τ ist und $[t_s, t_e)$ ein rechts-halboffenes Zeitintervall mit $t_s, t_e \in T$ ist. Hierbei ist t_s der Startszeitstempel und gibt den Zeitpunkt eines atomar auftretenden Messwertes an. Ferner ist t_e der Endzeitstempel, der das Gültigkeitsende eines Messwertes angibt, welches i.d.R. durch ein Fenster zugewiesen wurde. Hierbei ist T die Menge aller Zeitstempel mit $\mathbb{T} = (T, \leq)$.

5.2.1 Kontextmodell-Speicher

Der Kontextmodell-Speicher stellt im Prinzip nur einen Ausschnitt der schreibenden Datenströme dar. Da ein ähnliches Verhalten auch bei anderen Operatoren zu finden ist, wurde durch [DSTW02] eine *SweepArea* definiert, die es erlaubt Ausschnitte eines Datenstroms effizient zu speichern, zu durchsuchen und zu aktualisieren. Die *SweepArea* bietet dazu unter anderem die Methoden `insert` zum Hinzufügen, `iterator` zum sortierten Durchlaufen (bzgl. $\leq_{t_s}$) und `purgeElements` zum Bereinigen an. `purgeElements` bereinigt den Inhalt der *SweepArea* anhand eines Prädikats p_{remove}, indem die Methode nur solche Elemente entfernt, die sich über das Prädikat qualifiziert haben. Damit die *SweepArea* zum Löschen und zum Aktualisieren verwendet werden kann, wird folgendes Prädikat verwendet:

$$p^{\zeta}_{remove}(s, \hat{s}) := \begin{cases} true & \text{wenn } (t_s \geq \hat{t}_s \wedge p_{equal}(s, \hat{s})) \vee t_s \geq \hat{t}_e \\ false & \text{sonst} \end{cases}$$

Jedes neue Element s wird mit jedem Element $\hat{s}$ aus der *SweepArea* geprüft, ob sie zusammen das Prädikat erfüllen. Dabei wird geprüft, ob es sich bei s um eine neuere Version handelt. Das ist der Fall, wenn es einen größeren Startzeitstempel als $\hat{s}$ besitzt und sie gemeinsam das Gleichheitsprädikat p_{equal} erfüllen. Ist das nicht der Fall, wird geprüft, ob das Element $\hat{s}$ nicht mehr gültig ist. Dies ist der Fall wenn der Startzeitstempel von s größer als der Endzeitstempel von $\hat{s}$ ist. Da das Gleichheitsprädikat p_{equal} abhängig vom Kontextmodell ist, wird es hier nicht explizit aufgeführt.

5.2.2 Auswahlfunktion

Um die Auswahlfunktion umzusetzen, wie sie im vorigen Abschnitt als f_{next} beschrieben ist, wird zunächst die Abbildung R benötigt. Hierzu sei ein abstrakter Datentyp (ADT) `Metadatenverzeichnis` definiert, der folgende Methoden bereitstellt:

`getObservationStreams`() Liefert alle Observation Streams zurück.

`getResponseStream`(`Datenstrom` S) Liefert den Response Stream $\hat{S} \in S^l_{out}$ zu dem Request Stream $S \in S^l_{req}$.

`isRequestStream`(`Datenstrom` S) Liefert wahr zurück, wenn es sich bei S um einen Request Stream handelt.

`getRequests`() Liefert eine Prioritätsqueue bzgl. eines Zeitstempels, die alle Request Streams enthält, zu denen noch Anfragen ausstehen.

Die Methode `getResponseStream` setzt dabei die Abbildung R und `isRequest-Stream` den Term $S \in S^l_{req}$ um. Ferner liefert `getRequests` eine Prioritätsqueue, die alle Requests liefert. Hierbei sind alle Requests aus den Request Streams anhand ihres Startzeitstempels einsortiert. Durch die Prioritätsqueue wird unter anderem die in Definition 2 verwendete Bedingung $n(t) = n(\hat{t})$ umgesetzt, indem nur das letzte Element ausgegeben wird, welches zum aktuellen Zeitpunkt t an der Reihe ist. Des Weiteren beinhaltet `getRequests` bereits nur noch Request Streams, so dass statt der Abbildung R^{-1} direkt die Abbildung R verwendet werden kann. Unter Verwendung dieses Metadatenverzeichnisses kann die Auswahlfunktion wie folgt umgesetzt werden:

Algorithm 1 getNext(Metadatenverzeichnis M, Zeitstempel t)

Require: Metadatenverzeichnis M, Zeitstempel t
Ensure: Eine Menge O aus physischen Datenströmen S^p

```
1:  O ← M.getObservationStreams()
2:  loop
3:      r := (Ŝ, t̂) ← M.getRequests().peek()
4:      if t̂ ≤ t then
5:          M.getRequests().poll()
6:          S = M.getResponseStream(Ŝ)
7:          O.insert(S)
8:      else
9:          break
10:     end if
11: end loop
12: return O
```

Algorithmus 1 zeigt eine abstrakte Implementierung der Auswahlfunktion. Hierzu benötigt die Methode das genannte Metadatenverzeichnis, sowie den Parameter t, wie er auch in Definition 1 angegeben ist.

5.2.3 Physischer Operator

Um die Semantik des Brokers umzusetzen, wie sie in Definition (2) als logischer Broker-Operator beschrieben ist, muss bei der physischen Umsetzung noch der zeitliche Verlauf berücksichtigt werden. Ein physischer Operator konsumiert alle Elemente nur nacheinander. Daraus resultiert, dass auch der Broker zu einem Zeitpunkt t nicht wissen kann, ob noch Elemente folgen, dessen Zeitstempel kleiner als t sind. Im Falle der Request-Datenströme bedeutet dies, dass der Broker erst eine Anforderung bedienen darf, wenn er sicher gehen kann, dass nicht noch eine Anforderung eintrifft, die davor liegt. Um dieses blockierende Verhalten aufzulösen, bedient man sich einer Grundvoraussetzung von Datenströmen, bei der alle Elemente eines Datenstroms zeitlich anhand des Startzeitstempels sortiert sind, da dieser den eigentlichen atomaren Zeitpunkt eines Datenstromelements festlegt. Kommt demnach ein Element $(e, [t_s, t_e))$ am Operator an, so kann dieser davon ausgehen, dass aus demselben Eingang nur noch Elemente folgen werden, deren Startzeitstempel größer oder gleich t_s sind. Für ein Element muss der Broker also auf alle Eingänge entsprechend warten, bis er sichergehen kann, dass keine jüngeren Elemente mehr ankommen. Hierzu verwendet der Broker einen Zeitstempel min_{t_s}, bei dem der Broker sichergehen kann, dass aus keinem Eingang mehr Elemente kleiner als min_{t_s} folgen. Dieses Minimum kann dadurch gebildet werden, dass der Broker sich zu jedem Eingang i merkt, welchen Startzeitstempel t_{in_i} das letzte Element hatte. Das Minimum aller t_{in_i} ergibt entsprechend das globale Minimum , sodass $min_{t_s} := min(t_{in_1}, ..., t_{in_k})$, wobei k der Anzahl der Eingangsdatenströme entspricht. Durch Berücksichtigung dieses Minimums, der zuvor beschriebenen *SweepArea* und der Auswahlfunktion `getNext` kann der Broker, wie im folgenden Algorithmus umgesetzt werden:

Algorithm 2 Broker-Operator

Require: Physische Datenströme $S_{in_1}, ..., S_{in_k}$
Ensure: Physische Datenströme $S_{out_1}, ..., S_{out_m}$

1: Sei M ein Metadatenverzeichnis mit den Datenströmen S_{in} und S_{out}, sowie einer Zuordnung von Request-Datenströmen zu Ausgangsdatenströmen
2: $t_{in_1}, ..., t_{in_k}, min_{t_s} \in T \cup \{\bot\}$;$t_{in_i} \leftarrow \bot$ mit $1 \leq i \leq k$; $min_{t_s} \leftarrow \bot$
3: Sei *SA* eine leere *SweepArea*($\leq_{t_s}$, p^{ζ}_{remove})
4: Sei *buf* eine Prioritätsqueue für Elemente $(e, [t_s, t_e))$ mit Ordnungsrelation $\leq_{t_s}$
5: **for** $s := (e, [t_s, t_e)) \hookleftarrow S_{in_j}$ **do**
6: $t_{in_j} \leftarrow t_s$
7: $min_{t_s} \leftarrow min(t_{in_1}, ..., t_{in_k})$
8: **if** *M.isRequestStream*(S_{in_j}) **then**
9: *M.getRequests().offer*((S_{in_j}, t_s))
10: **else**
11: *buf.offer(s)*
12: **end if**
13: **if** min_{t_s} **not** $\bot$ **then**
14: **while not** *buf.isEmpty()* **do**
15: $\hat{s} := (\hat{e}, [\hat{t_s}, \hat{t_e})) \leftarrow buf.peek()$
16: **if** $\hat{t_s} \leq min_{t_s}$ **then**
17: $\hat{s} \leftarrow buf.poll()$

```
18:            SA.purgeElements(ŝ)
19:            SA.insert(ŝ)
20:          else
21:            break
22:          end if
23:        end while
24:        Iterator it = getNext(M, min_{t_s})
25:        while not it.hasNext() do
26:          n := (S_o, t_o) ← it.next()
27:          for all s̃ ↩ SA.iterator() do
28:            s̃ ↪ S_o
29:          end for
30:        end while
31:      end if
32:    end for
```

Der Algorithmus wird für jedes ankommende Element s ausgeführt. Dabei wird zunächst das Minimum für den zugehörigen Eingsangsdatenstrom gesetzt und danach das globale Minimum berechnet (Zeile 6–7). Darauf folgend wird geprüft, ob es sich um einen Request Stream handelt. Wenn dem so ist, wird dem Metadatenverzeichnis eine neue Anforderung übergeben. Falls nicht, wird das ankommende Element zunächst in einem Puffer abgelegt (Zeile 8–12). Ist in einem weiteren Schritt dann $min_{t_s} \neq \perp$, sodass mindestens aus jedem Eingangsdatenstrom ein Zeitstempel t_{in} vorliegt, dann wird zunächst der Puffer behandelt. Hierzu werden nacheinander die Elemente $\hat{s}$ aus dem Puffer geholt. Diese werden dann mit `purgeElements` benutzt, um die *SweepArea SA* zu bereinigen. Hierbei werden, wie oben beschrieben, alle Elemente aus der *SweepArea* entfernt, die laut p^{ζ}_{remove} nicht mehr gültig sind. Anschließend wird das Element $\hat{s}$ der *SweepArea* hinzugefügt. Der Inhalt der *SweepArea* entspricht somit dem aktuellen Zustand des Kontextmodells (Zeile 13–18). Ist jedoch ein Element $\hat{s}$ zeitlich vor min_{t_s}, dann darf es noch nicht aus dem Puffer geholt werden, da eventuell erst andere Eingaben und Requests abgearbeitet werden müssen (Zeile 16 bzw. 21). Somit dient dies auch der zeitlich korrekten Aktualisierung des Kontextmodells. Wenn alle aktuell gültigen Daten aus dem Puffer geholt wurden, dann werden mit `getNext` alle Ausgangsdatenströme bestimmt, die zum Zeitpunkt min_{t_s} ausgeführt werden müssen. Anschließend wird der gesamte Inhalt der *SweepArea*, also das gesamte aktuell gültige Kontextmodell, an die vorher ausgewählten Ausgangsdatenströme geschickt (Zeile 24–30).

5.2.4 Optimierungen

Aus der relationalen Algebra sind Optimierungsregeln bekannt, die es erlauben einen Anfrageplan für eine effizientere Ausführung umzustellen, ohne dabei die Ergebnisse zu verändern. Ähnliche Regeln existieren auch für die relationale Algebra auf Datenströmen. Bei diesen Optimierungsregeln werden jedoch nur azyklische Anfragepläne betrachtet. Mit der Einführung des Broker-Operators werden jedoch Zyklen in Anfragepläne integriert, die sich in gewissen Anwendungsszenarien nicht vermeiden lassen. Dennoch lassen

sich auch hier Teile entsprechender Anfragepläne unter gewissen Umständen optimieren. Hierzu muss ein Anfrageplan mit Zyklen jeweils am Broker geteilt werden. Die dabei entstehenden Teilpläne sind azyklisch und lassen sich mit den gleichen Optimierungsregeln umstrukturieren, wie es auch in [Krä07] der Fall ist. Selbst eine Optimierung über den Broker hinweg ist möglich. So kann bspw. eine Selektion in einem Teilplan, der einen Observation Stream darstellt, vor den Broker gesetzt werden, wenn die Selektion nur Objekte herausfiltert, die nicht zur Aktualisierung anderer Objekte des Kontextmodells benötigt werden. Ein Beispiel könnte wie folgt sein. Sensoren am Fahrzeug detektieren auf einer Autobahn alle Objekte vor dem eigenen Fahrzeug inklusive der Fahrspur, auf der sich die Objekte befinden. Wenn eine Anwendung jedoch nur die Objekte auf der eigenen Fahrspur benötigt, würde sie einen Observation Stream, also eine Anfrage am Broker registrieren, in der eine Selektion auf die entsprechende Fahrspur enthalten ist. Wenn keine anderen Anwendungen existieren, die auch Fahrzeuge auf anderen Fahrspuren benötigen, dann können die Fahrzeuge, die sich nicht auf der eigenen Fahrspur befinden, bereits herausgefiltert werden, bevor sie im Kontextmodell abgespeichert werden. Damit reduziert sich dann der Aufwand für die Aktualisierung des Kontextmodells.

6 Experimente

Die Funktionsweise des Brokers wurde durch Experimente geprüft, da keine vergleichbaren Algorithmen bzw. Operatoren in DSMS für eine sinnvolle Evaluation existieren. Ferner erfolgte eine Implementierung in das Datenstrom Management Framework Odysseus [BGJ+09]. Hierbei wurde der im vorigen Abschnitt beschriebene physische Operator umgesetzt und integriert. Aufbauend auf diesem Operator wurde ein beispielhafter Anfrageplan in Anlehnung an die in Abschnitt 4 Motivation im System installiert, wie er in Abbildung 5 gezeigt wird. Dieser Plan beinhaltet zwei Datenquellen (A und B), die je-

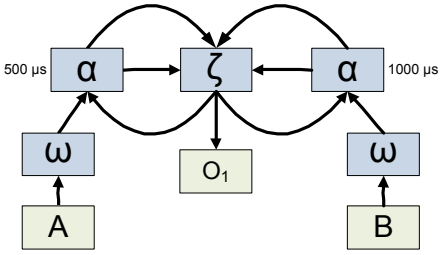

Abbildung 5: Anfrageplan der Experimente

weils einen Sensor simulieren sollen. Diese haben beide eine feste Frequenz und schicken immer abwechselnd, ebenfalls in einer festen Frequenz, neue Elemente in aufsteigender Reihenfolge bzgl. der Startzeitstempel an Odysseus. Die Elemente gelangen zunächst in einen Fenster-Operator ω, das jedem Element eine Gültigkeitsdauer zuweist, sodass Elemente später auch vom Broker entfernt werden können. Anschließend gelangen die Daten in die Aktualisierungs-Operatoren α. Diese fordern daraufhin mit einem Request die Daten

beim Broker an. Wenn der Broker auf Grundlage seiner Semantik den Request bedienen kann, liefert er das aktuell gültige Kontextmodell an den Aktualisierungs-Operator. Diese aktualisieren anhand des Elements und des übergebenen Kontextmodells den aktuellen Zustand. Um hierbei die Verarbeitungsdauer einer solchen Aktualisierung zu simulieren, wird auf der linken Seite 500 μs und auf der rechten Seite 1000 μs gewartet, bis das Ergebnis wieder dem Broker übergeben wird. Damit ist entsprechend ein Aktualisierungsschritt im Zyklus abgeschlossen. Ferner wird dazu die linke Aktualisierung als Zyklus A und die rechte als Zyklus B bezeichnet. Dies bedeutet, dass sowohl ein Zyklus als auch ein Sensor feste Frequenzen haben. Aus diesem Grund wurde in der Evaluation zwischen zwei Fällen unterschieden. Zum einen ist es möglich, dass die Sensorfrequenz niedriger als die Frequenz der Zyklen ist und zum anderen ist es möglich, dass die Frequenz des Sensors höher als die der Zyklen ist. Als Messpunkt wurde hierbei die Zeit gewählt, die ein neues Element warten muss, bis es das Kontextmodell bekommt und aktualisieren kann. Diese ergibt sich jeweils pro Aktualisierungs-Operator von dem Abschicken eines Requests bis zur Antwort durch den Broker anhand des Kontextmodells.

Die Tests fanden auf einem Intel Core 2 Duo mit 2 CPUs bei 2.20 GHz und 4 GB Arbeitsspeicher statt. Als Betriebssystem wurde Windows 7 mit 64 Bit verwendet.

6.1 Niedrigere Sensorfrequenz

Wie zuvor beschrieben beträgt die Verarbeitungszeit von Zyklus A 500 μs und von Zyklus B 1000 μs. Dementsprechend wurde für die Sensoren eine kleinere Frequenz genommen, sodass die Sensoren jede 2000 μs ein neues Element erzeugen. Die Ergebnisse des Experiments zeigt Abbildung 6. In der Abbildung ist zu sehen, dass die durchschnittliche

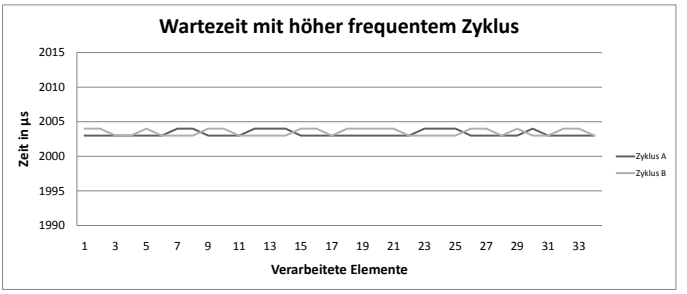

Abbildung 6: Wartezeit bei einem niedriger frequenten Sensor

Wartezeit eines Elements bei etwas über 2000 μs liegt. Die Verarbeitungsgeschwindigkeit wird demnach von dem Sensor, also der kleineren Frequenz, vorgegeben. Da die Zyklen mit einer größeren Frequenz arbeiten, ist die Aktualisierung bereits beendet bevor ein neues Element das System erreicht. Der Broker muss bei einem Request von Zyklus A erst warten, bis ein Request von Zyklus B eingegangen ist. Dies ist nötig, damit der Broker weiß, dass nicht eventuell der Request von Zyklus B vor dem von Zyklus A abgearbeitet werden müsste. Entsprechend richtet sich die Wartezeit eines Elements nach der höchsten

Frequenz der Sensoren. Da hier die Frequenz der Sensoren konstant ist, ist auch die Wartezeit aller Elemente konstant. Diese schwankt lediglich ein wenig auf Grund des Schedulings und der momentanen Auslastung des Gesamtsystems. Um hierbei nicht von der Frequenz einer Datenquelle abhängig zu sein, kann ein Heartbeat-Mechanismus in Form von Punctuations [TMSF03] verwendet werden. Punctuations sind in der Regel einfache Zeitstempel, die den zeitlichen Fortschritt in einem Datenstrom angeben. Ein Operator kann dann davon ausgehen, dass nach einer Punctuation nur noch Elemente folgen, die einen größeren Zeitstempel als den Zeitstempel der letzten Punctuation haben. Da dies der Semantik des Minimum-Zeitstempels entspricht, wie er in Abschnitt 5.2 verwendet wird, ist es mit einer Punctuation möglich, den Minimum-Zeitstempel bereits vor einem neuen Messwert zu berechnen und somit die Wartezeit für einen anderen wartenden Messwert zu verkürzen.

6.2 Höhere Sensorfrequenz

Im Gegensatz zum vorherigen Test, wurde des Weiteren der Fall betrachtet, in dem die Sensorfrequenz höher ist. Indem die Sensoren in einem Abstand von $100\mu s$ neue Elemente erzeugen, wurde eine größere Frequenz als die der Zyklen gewählt. Abbildung 7 zeigt die Messergebnisse. Hierbei ist zu erkennen, dass die Wartezeit einzelner Elemente

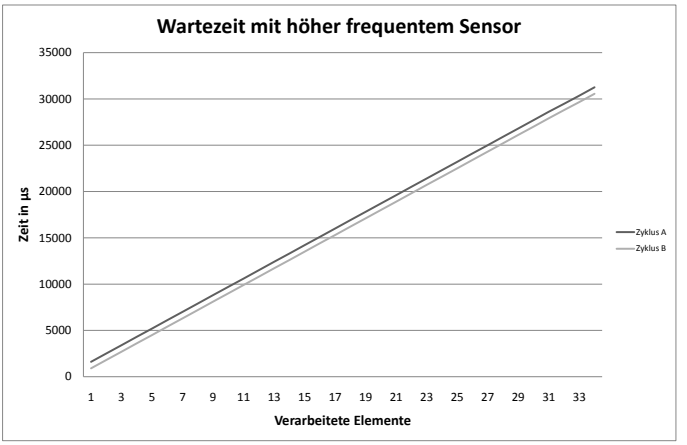

Abbildung 7: Wartezeit bei einem höher frequenten Sensor

bei jedem weiteren Element zunimmt. Dies begründet sich dadurch, dass neue Elemente wesentlich schneller beim System ankommen, als sie vom System verarbeitet werden können. Da die Frequenz der Sensoren konstant ist, nimmt auch die Wartezeit je Element mit einem konstanten Faktor zu, sodass die entsprechend konstante Steigung entsteht. Die ansteigende Wartezeit entspricht ebenfalls der Aktualität des Kontextmodells. Je länger ein Element auf die Aktualisierung warten muss, desto mehr weicht auch das aktuell gültige Kontextmodell von der tatsächlichen Welt ab. Um hierbei ein möglichst aktuelles Kon-

textmodell zu gewährleisten, ist es sinnvoll, dass für die Sensoren im Vergleich zu der Verarbeitungsfrequenz der Zyklen eine kleinere Frequenz gewählt wird.

Die Experimente zeigten demnach die erwarteten Ergebnisse, indem bei niedrigerer Frequenz eine bestimmte Zeit gewartet werden muss. Hierbei kann man u.a. beobachten, dass ein Messwert aus Sensor A erst bedient werden kann, wenn ein weiteres Element aus Sensor B vorhanden ist. Denn erst zu diesem Zeitpunkt kann der Broker sicher gehen, dass aus Sensor B nicht eventuell noch Messwerte folgen, die zuerst ausgeführt werden müssten. Demnach entstehen zusätzliche Latenzen, die auf Grund der Transaktionskontrolle des Brokers entstehen und bei einer nicht-transaktionssicheren Implementierung mit einem (Hauptspeicher-)DBMS entsprechend nicht vorhanden wären. Des Weiteren entsprechen die Ergebnisse bei einer höheren Sensorfrequenz ebenso den Erwartungen, indem es einen Systemüberlauf gibt, da wie beschrieben zusätzliche Latenzen entstehen.

7 Fazit

Obwohl sich die Verwendung eines DSMS im Bereich kontextsensitiver Anwendungen anbietet, werden für die Verarbeitung von Sensordaten in der Regel feste Programme eingesetzt. Möchte man jedoch solche Konzepte in einem DSMS umsetzen, muss man auch die aktuelle Umgebung in einem Kontextmodell abbilden. Hierbei muss unter anderem eine möglichst effiziente Speicherung berücksichtigt werden. Bei den Aktualisierungen des Kontextmodells, die jeweils periodisch durch ein neu ankommendes Element angestoßen werden, müssen die verschiedenen lesenden und schreibenden Zugriffe auf das Kontextmodell berücksichtigt werden, damit Aktualität und Konsistenz des Kontextmodells sichergestellt werden. Wir haben dazu den Broker-Operator auf Grundlage einer festen Semantik eingeführt und eine abstrakte Implementierung gezeigt. Dieser Operator erlaubt es, das Kontextmodell transaktionssicher im Hauptspeicher zu verwalten, ohne dass dabei die temporale Ordnung der Daten vernachlässigt wird. Die Umsetzung des Brokers erfolgte in dem komponentenbasierten Datenstrom Management Framework Odysseus. Diese Umsetzung wurde ebenso für Experimente genutzt. Hierbei wurde das Verhalten bei verschiedenen Frequenzen von Datenquelle und Zyklen beobachtet. Bei einem langsameren Sensor erfolgen Aktualisierung innerhalb einer konstanten Zeit, die auf Grund der Transaktionkontrolle nicht nur vom eigenen Sensor sondern auch von allen Sensoren beeinflusst wird. Werden die Daten jedoch schneller erzeugt, als sie im Zyklus verarbeitet werden können, so entsteht ein Datenstau, wodurch Aktualität des Kontextmodells und die tatsächliche Umgebung mit der Zeit immer mehr auseinander laufen.

Zukünftig wird der Broker-Operator in einem DSMS-basierten Framework für FAS eingesetzt. Dabei werden neben der zeitlichen Koordination insbesondere auch anwendungsspezifische Löschstrategien für das Kontextmodell entwickelt, um ungültige Elemente aus dem Kontextmodell zu entfernen. Außerdem werden Anfragepläne mit zwei oder mehr Broker-Operatoren entwickelt, um neben dem eigentlichen Kontextmodell auch temporäre Kontextmodelle zuzulassen. Dies ist insbesondere deswegen wichtig, weil auf Grund möglicher Fehlmessungen nicht jedes Objekt direkt in das eigentliche Kontextmodell aufgenommen werden soll, sondern erst nachdem es mehrmalig erkannt worden ist.

Literatur

[AAB+05] Daniel Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Cetintemel, Mitch Cherniack, Jeong Hwang, Wolfgang Lindner, Anurag Maskey, Alexander Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing und Stan Zdonik. The Design of the Borealis Stream Processing Engine. In *Second Biennial Conference on Innovative Data Systems Research (CIDR 2005)*, 2005.

[ABB+03] Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Keith Ito, Itaru Nishizawa, Justin Rosenstein und Jennifer Widom. STREAM: the stanford stream data manager (demonstration description). In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, Seiten 665–665, San Diego, California, 2003. ACM.

[ACc+03] Daniel J. Abadi, Don Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul und Stan Zdonik. Aurora: a new model and architecture for data stream management. *The VLDB Journal*, 12(2):120–139, 2003.

[BGJ+09] A. Bolles, M. Grawunder, J. Jacobi, D. Nicklas und H.-J. Appelrath. Odysseus: Ein Framework für maßgeschneiderte Datenstrommanagementsysteme. In *39. GI Jahrestagung, Workshop: Verwaltung, Analyse und Bereitstellung kontextbasierter Informationen*, 2009.

[CCMP06] C Cappiello, M Comuzzi, E Mussi und B Pernici. Context Management for Adaptive Information Systems. *Electronic Notes in Theoretical Computer Science*, 146(1):69–84, 2006.

[CEB+09] Nazario Cipriani, Mike Eissele, Andreas Brodt, Matthias Grossmann und Bernhard Mitschang. NexusDS: a flexible and extensible middleware for distributed stream processing. In *IDEAS '09: Proceedings of the 2009 International Database Engineering & Applications Symposium*, Seiten 152–161, New York, NY, USA, 2009. ACM.

[CGM09] Badrish Chandramouli, Jonathan Goldstein und David Maier. On-the-fly Progress Detection in Iterative Stream Queries. *Proceedings oth the VLDB Endowment 2009*, 2(1):241–252, 2009.

[DS00] Guozhu Dong und Jianwen Su. Incremental maintenance of recursive views using relational calculus/SQL. *SIGMOD Rec.*, 29(1):44–51, 2000.

[DSTW02] J.P. Dittrich, Bernhard Seeger, D.S. Taylor und Peter Widmayer. Progressive merge join: A generic and non-blocking sort-based join algorithm. In *Proceedings of the 28th international conference on Very Large Data Bases*, Seite 310. VLDB Endowment, 2002.

[EN09] Ramez A. Elmasri und Shamkant B. Navathe. *Grundlagen von Datenbanksystemen*. Pearson Studium, 3. aktualisierte auflage. bachelorausgabe.. Auflage, 2009.

[GADI08] D. Gyllstrom, J. Agrawal, Y. Diao und N. Immerman. On supporting kleene closure over event streams. In *IEEE 24th International Conference on Data Engineering, 2008. ICDE 2008*, Seiten 1391–1393, 2008.

[GAE06] Thanaa M. Ghanem, Walid G. Aref und Ahmed K. Elmagarmid. Exploiting predicate-window semantics over data streams. *SIGMOD Rec.*, 35(1):3 – 8, 2006.

[GAW+08] Bugra Gedik, Henrique Andrade, Kun-Lung Wu, Philip S. Yu und Myungcheol Doo. SPADE: the system s declarative stream processing engine. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, Seiten 1123–1134, Vancouver, Canada, 2008. ACM.

[GBz06] Lukasz Golab, Kumar Bijay und M. Özsu. On Concurrency Control in Sliding Window Queries over Data Streams. In Yannis Ioannidis, Marc Scholl, Joachim Schmidt, Florian Matthes, Mike Hatzopoulos, Klemens Boehm, Alfons Kemper, Torsten Grust und Christian Boehm, Hrsg., *Advances in Database Technology - EDBT 2006*, Jgg. 3896 of *Lecture Notes in Computer Science*, Seiten 608–626. Springer Berlin / Heidelberg, 2006.

[HJ04] X. Huang und C.S. Jensen. Towards a streams-based framework for defining location-based queries. In *Proc. STDBM*, Seiten 78–85. Citeseer, 2004.

[Krä07] Jürgen Krämer. *Continuous Queries over Data Streams - Semantics and Implementation*. Dissertation, Philipps-Universität Marburg, 2007.

[KS05] Jürgen Krämer und Bernhard Seeger. A Temporal Foundation for Continuous Queries over Data Streams. In *Proceedings of the 11th International Conference on Management of Data (COMAD)*, Seiten 70–82, 2005.

[MA08] M.F. Mokbel und W.G. Aref. SOLE: scalable on-line execution of continuous queries on spatio-temporal data streams. *The VLDB JournalThe International Journal on Very Large Data Bases*, 17(5):995, 2008.

[PS04] K. Patroumpas und T. Sellis. Managing trajectories of moving objects as data streams. In *Proceedings of the STDBM*, Jgg. 4. Citeseer, 2004.

[Rea10] Realtime Monitoring GmbH. RTM Analyzer, 2010.

[RHC+02] Manuel Roman, Christopher Hess, Renato Cerqueira, Anand Ranganathan, Roy H. Campbell und Klara Nahrstedt. A Middleware Infrastructure for Active Spaces. *IEEE Pervasive Computing*, 1(4):74–83, 2002.

[See04] Bernhard Seeger. Datenströme. *Datenbank-Spektrum*, 4(9):30–33, 2004.

[TMSF03] P.A. Tucker, D. Maier, T. Sheard und L. Fegaras. Exploiting punctuation semantics in continuous data streams. *IEEE Transactions on Knowledge and Data Engineering*, 15:555–568, 2003.

[WRML08] Mingzhu Wei, Elke Rundensteiner, Murali Mani und Ming Li. Processing recursive XQuery over XML streams: The Raindrop approach. *Data Knowl. Eng.*, 65(2):243–265, 2008.