Priorisierte Verarbeitung von Datenstromelementen

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

[1]Informationssysteme
Department für Informatik
Universität Oldenburg
D-26121 Oldenburg

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

[jonas.jacobi|andre.bolles|marco.grawunder|daniela.nicklas|appelrath]
@uni-oldenburg.de

Abstract: Datenstrommanagementsysteme, die effizient und skalierbar viele Anfragen gleichzeitig auf kontinuierlichen Daten auswerten können, sind eine vielversprechende Technologie für zukünftige Überwachungs- und Steuerungsanwendungen, z. B. bei dezentralen Energieanlagen. Dafür ist es jedoch notwendig, dass bestimmte Systemzustände (z. B. Warnungen oder Alarme) gegenüber dem Normalbetrieb bevorzugt behandelt werden. Im Gegensatz zu anderen Ansätzen, die solche Zustände durch eigene, vom Laufzeitsystem priorisierte Anfragen beschreiben, betrachten wir in diesem Beitrag priorisierte Datenstromelemente. Wir schaffen auf Basis einer logischen und physischen Operatoralgebra die formale Grundlage für die Priorisierung einzelner Datenstromelemente. Priorisierte Elemente profitieren darin doppelt. Zum einen können sie andere Elemente bspw. in Puffern „überholen“ und zum anderen können priorisierte Ergebnisse zustandsbehafteter Operatoren früher erzeugt werden, als bisherige Ansätze dies erlauben. Die Anfragesemantik bleibt dabei unverändert. Wir haben unseren Ansatz im Datenstrom-Framework ODYSSEUS implementiert und zeigen durch formale Abschätzungen und umfangreiche Messungen, wie durch die bevorzugte Behandlung priorisierter Elemente sowohl innerhalb von Anfrageoperatoren als auch beim Transport zwischen Anfrageoperatoren eine deutlich geringere Latenz erreicht werden kann. Davon können dann alle Anfragen profitieren, die solche priorisierten Elemente verarbeiten.

1 Einleitung

Wenn in Anwendungssystemen kontinuierlich große Datenmengen auftreten, ist es häufig aus Performanz- und Speicherplatzgründen nicht möglich, diese vor der Verarbeitung z. B. in einem Datenbankmanagementsystem (DBMS) zu speichern. Stattdessen müssen die Daten strombasiert verarbeitet werden, um unter Umständen erst später gegebenenfalls aggregierte oder ausgewählte Elemente zu archivieren. Auf eine solche Verarbeitung angewiesene Anwendungen sind zahlreich und nehmen in betriebswirtschaftlichen, technisch-ingenieurs-wissenschaftlichen und naturwissenschaftlichen Kontexten kontinuierlich zu. Charakteristisch ist, dass dynamische Systeme kontrolliert oder gesteuert werden sollen,

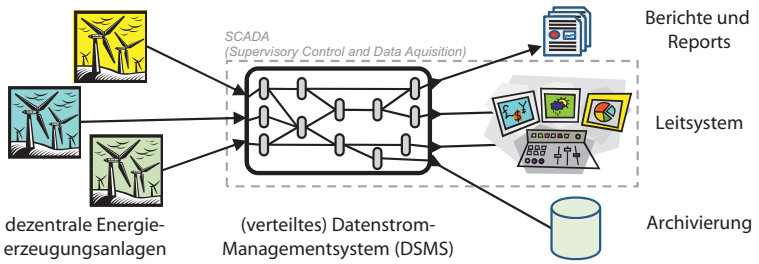

Abbildung 1: Szenario: Einsatz von Datenstrommanagementsystemen in Leitsystemen

die entweder durch Sensoren (wie z. B. technische Anlagen) oder Transaktionen (wie z. B. im Finanzsektor) beobachtet werden. Bisher werden solche Probleme in der Praxis meist mit Hard- oder Softwaresystemen gelöst, in welche die Verarbeitungslogik fest „einprogrammiert“ ist. Mit zunehmender Komplexität der Systeme und höherer Änderungsrate in den Anforderungen an die Systeme (welche eine bessere Wartbarkeit erfordert) werden diese Lösungen jedoch zunehmend unbrauchbar.

Seit einigen Jahren wird deswegen auf dem Gebiet der Datenstrommanagementsysteme (DSMS) geforscht, so dass mittlerweile erste kommerzielle Systeme verfügbar sind und eingesetzt werden. Analog zu einem DBMS ist die Kernidee dabei, das Informationsbedürfnis nicht durch ein Programm zu kodieren, sondern in einer Anfrage zu deklarieren, die dann von dem DSMS übersetzt, optimiert und ausgeführt wird. Im Unterschied zu einem DBMS sind die Anfragen jedoch meist kontinuierlich, das heißt sie werden registriert und über einen längeren Zeitraum hinweg auf den Eingangsdatenströmen verarbeitet. Das Ergebnis einer Anfrage auf Datenströmen ist in der Regel auch wieder ein Datenstrom.

Trotz der steigenden Bedeutung dieser Technologie sind viele Aspekte bisher ungelöst, was einem direkten kommerziellen Einsatz in einigen vielversprechenden Anwendungsdomänen, wie der Energiebranche, entgegen steht. In dieser Branche müssen vor allem durch die zunehmende Verbreitung regenerativer Energieerzeugung geographisch und organisatorisch immer mehr und weiter verteilte Anlagen und Stromnetze überwacht, gesteuert und reguliert werden. Der Einsatz von DSMS kann hier die Skalierbarkeit, Wartbarkeit und Flexibilität deutlich verbessern. Dazu muss zum einen ein performanter und hoch-ausfallsicherer Normalbetrieb sichergestellt werden, zum anderen wird für kritische Ausnahmesituationen eine zeitnahe (priorisierte) Verarbeitung benötigt.

Grundsätzlich gibt es zwei Ansätze für eine priorisierte Verarbeitung in Datenströmen: Zum einen können einzelne Anfragen im Datenstromsystem priorisiert werden, die dann vom System (bspw. durch einen Scheduler) bevorzugt bearbeitet werden, zum anderen können einzelne Elemente im Datenstrom mit bestimmten Eigenschaften (z. B. der Überschreitung eines Grenzwerts) priorisiert werden, wovon dann alle Anfragen profitieren, die diese Elemente verarbeiten. Unserem Wissen nach gibt es für letzteres bisher keine Ansätze im Datenstromkontext.

Ein typischer Anwendungsfall für eine elementbasierte Priorisierung findet sich z. B. bei

der Überwachung dezentraler Energieerzeugungsanlagen in einem Leitsystem (SCADA, Supervisory Control and Data Aquisition). Dabei müssen Statusmeldungen von Anlagen für die Darstellung in der Leitzentrale beispielsweise mit stromnetztopologischen Informationen angereichert werden. Außerdem werden auf den Meldungen – je nach Anlagentyp – weitere Analysen durchgeführt. Trifft nun eine Alarmmeldung ein, muss diese zeitnah zum Auftreten verarbeitet werden, und eine schnelle Reaktion auf Fehlerzustände ermöglichen. Da der Alarm prinzipiell in den gleichen Anfragen wie die restlichen Statusmeldungen verarbeitet wird, lässt sich keine Anfragepriorisierung durchführen, die die benötigten Ergebnisse schneller erzeugt. Insbesondere kann sich eine kritische Situation auch anhand von Grenzwertüberschreitungen eines Sensors ohne Alarmmeldungen identifizieren lassen.

Daher stellen wir in diesem Beitrag einen Ansatz zur Priorisierung einzelner Datenstromelemente vor, der es zum Ziel hat, die Latenz von Datenelementen zu verringern, die aufgrund ihres Inhaltes (Alarm, Warnung, Schwellenwert ...) besonders relevant sind. Wir beschreiben das Problem und formalisieren unseren Ansatz zur Priorisierung von Datenstromelementen in Kapitel 2. Kapitel 3 beschreibt eine physische Operatoralgebra, welche die priorisierte Verarbeitung von Datenstromelementen erlaubt. Kapitel 4 detailliert die von uns entwickelten operatorunabhängigen Techniken, mit denen priorisierte Elemente andere Elemente überholen können und die so eine optimierte Verarbeitung realisieren. Wir evaluieren den Ansatz in Kapitel 5, geben in Kapitel 6 eine Übersicht über verwandte Arbeiten und schließen in Kapitel 7 mit einem Fazit.

2 Priorisierte Datenstromelemente

Das Ziel dieser Arbeit ist es, besonders relevante Anfrageergebnisse mit möglichst geringer Latenz zu berechnen. Im Gegensatz zu bisherigen Arbeiten wird die Relevanz eines Ergebnisses nicht durch die Anfrage definiert (Anfragepriorisierung), sondern durch die Datenelemente, die zur Berechnung des Ergebnisses benötigt werden (Datenpriorisierung). Es gibt eine kleine Menge an Datenstromelementen innerhalb eines Datenstroms, die priorisiert verarbeitet werden muss. Ergebnisse, für die diese Elemente eine Rolle spielen, müssen also mit möglichst geringer Latenz berechnet werden. Jedes Element eines Datenstroms wird zu einem Zeitpunkt t_{in} im DSMS registriert. Die Latenz eines Ergebnisses einer Operation ist die Dauer vom Zeitpunkt t_{in} des ältesten Elements bzw. des Elements mit der höchsten Priorität, das zum Ergebnis beigetragen hat, bis zur Erzeugung des Ergebnisses. Bei einem Verbund zweier Alarmmeldungen gleicher Priorität wäre dies z. B. das ältere der beiden Eingabeelemente. Bei einem Verbund einer Alarmmeldung mit niedrig priorisierten Informationen ist für die Latenz nur das Eintreffen der Alarmmeldung relevant, da diese schnell verarbeitet werden soll. Natürlich sind auch andere Definitionen der Latenz eines Elementes (wie z. B. in [SCLP08]) denkbar und in anderen Anwendungskontexten sinnvoll.

Die Priorisierung ist nur für die Geschwindigkeit der Verarbeitung relevant und darf die Semantik von Anfragen nicht verändern. Zur Definition der Anfragesemantik stützen wir uns auf die logische Operatoralgebra aus [Krä07], und erweitern die ebenfalls darin definier-

te physische Algebra derart, dass ohne Veränderung der Anfragesemantik Priorisierung ausgedrückt werden kann und priorisierte Elemente andere Elemente bei der Verarbeitung überholen können.

Die logische Operatoralgebra aus [Krä07] definiert die Semantik der verschiedenen Operatoren über logischen Datenströmen. Wir definieren in Anlehnung einen logischen Datenstrom $S_T^l \in \mathbb{S}^l$ als Menge von Elementen (e, t, n), wobei $e \in \Omega_T$ ein Tupel vom Typ T ist und $n \in \mathbb{N}, n > 0$ die Anzahl des Tupels zum Zeitpunkt $t \in \mathbb{T}$ im Datenstrom definiert. Ein logischer Datenstrom $\{(a, 1, 2), (b, 2, 1)\}$ enthält also zum Zeitpunkt 1 zwei Kopien des Elements a und zum Zeitpunkt 2 einmal das Element b. Die Definition ist unabhängig von einer Ordnung der Elemente im Datenstrom. Die Semantik eines logischen Operators wird über die vom Operator erzeugte Ausgabemenge definiert, je nach Operator in Abhängigkeit von einem bzw. zwei logischen Eingabeströmen. Als Beispiel sei die Definition des kartesischen Produkts $\times : \mathbb{S}_{T_1}^l \times \mathbb{S}_{T_2}^l \rightarrow S_{T_3}^l$ aus [Krä07] aufgeführt, wobei die Funktion $\circ : \Omega_{T_1} \times \Omega_{T_2} \rightarrow \Omega_{T_3}$ ein Ausgabetupel durch Konkatenation zweier Eingabetupel erzeugt und deren Vorkommen multipliziert:

$$\times(S_1, S_2) \coloneqq \{(\circ(e_1, e_2), t, n_1 \cdot n_2) | (e_1, t, n_1) \in S_1 \wedge (e_2, t, n_2) \in S_2\}$$

Physische Datenströme ($\mathbb{S}^p$) erlauben eine kompaktere Repräsentation der logischen Datenströme durch die Zusammenfassung zusammenhängender Zeitpunkte zu Intervallen und definieren eine Multimenge von Elementen $(e, [t_s, t_e))$, wobei $[t_s, t_e)$ die Gültigkeit des Elements im Datenstrom vom Startzeitpunkt t_s bis zum Endzeitpunk t_e repräsentiert. Damit unterscheidet sich diese Datenstromrepräsentation von der sonst üblichen Verwendung positiver und negativer Tupel zum Markieren der Gültigkeit (z. B. in [ABW06]). Die Elemente sind monoton steigend nach dem Startzeitstempel geordnet. Physische Datenströme lassen sich eindeutig auf logische Datenströme abbilden.

Zum Ausdrücken von Prioritäten führen wir eine spezielle Variante physischer Datenströme ein: Priorisierte physische Datenströme ($\mathbb{S}_{pr}^p$), oder kurz: priorisierte Datenströme.

Wir definieren schwach und stark priorisierte Datenströme um die Striktheit, mit welcher eine Ordnung zwischen Elementen innerhalb eines Datenstroms definiert wird, unterscheiden zu können. Ein stark priorisierter physischer Datenstrom $(S_{pr}^p, \leq)$ mit $S_{pr}^p \in \mathbb{S}_{pr}^p$ des Typs T – im Folgenden kurz S_{pr}^p – ist eine potentiell unendliche, geordnete Multimenge von Elementen $(e, [t_s, t_e), p)$, wobei wie gehabt $e \in \Omega_T$ ein Tupel vom Typ T, $[t_s, t_e)$ ein rechtsoffenes Zeitintervall in der diskreten Zeitdomäne $\mathbb{T}$ und $p \in \mathbb{N}_0$ zusätzlich die Priorität des Elements repräsentiert. Je wichtiger das Element, desto höher ist seine Priorität. Ein nicht priorisiertes Element wird durch $p = 0$ gekennzeichnet. Die Elemente eines solchen Datenstroms sind durch die Ordnungsrelation $\leq$ geordnet, die kompatibel zu der folgenden partiellen Ordnung sein muss:

$$<_{S_{pr}^p}^{strong} \coloneqq \{(a, b) \in S_{pr}^p \times S_{pr}^p | a.t_s < b.t_s \wedge a.p \geq b.p\} \quad (1)$$

Das bedeutet, dass Elemente der gleichen Priorität monoton steigend nach ihren Startzeitstempeln geordnet auftreten, jedoch Elemente höherer Priorität trotz eines jüngeren Zeitstempels vor Elementen mit geringerer Priorität eintreffen *können*. Wir weisen darauf hin, dass unter Elementen mit gleichen Startzeitstempeln keine Ordnung definiert wird, auch

nicht für Elemente mit unterschiedlichen Prioritäten. Dies liegt daran, dass Elemente trotz gleichen Zeitstempels sequentiell verarbeitet werden. Ansonsten müsste für alle Elemente eines Startzeitpunktes auf das Eintreffen eines jüngeren Elementes gewartet werden, um sicherzustellen, dass kein weiteres Element mit gleichem Startzeitstempel eintrifft, das in der Ordnung vor den bisherigen Elementen liegt. Das stünde im Widerspruch zu unserem Ziel, die Latenz zu optimieren.

Dass Elemente gleicher Priorität zeitlich geordnet sind, scheint einem intuitiven Verständnis von priorisierten Datenströmen zu entsprechen. Das Ziel dieser Arbeit ist aber nicht, die Ordnung der Elemente vorteilhaft für priorisierte Elemente zu gestalten, sondern ihre Latenz zu verbessern. Wir zeigen in Kapitel 3.1, dass beides nicht zusammenhängen muss, beziehungsweise sich unter bestimmten Umständen sogar widerspricht, und führen daher eine weitere Definition schwach priorisierter Datenströme ein.

Schwach priorisierte physische Datenströme garantieren keine Ordnung innerhalb priorisierter Elemente. Sie unterscheiden sich von einem stark priorisierten einzig dadurch, dass ihre Ordnung zum Folgenden kompatibel sein muss:

$$<_{S_{pr}^p}^{weak} := \{(a,b) \in S_{pr}^p \times S_{pr}^p | a.t_s < b.t_s \wedge b.p = 0\} \tag{2}$$

Ein physischer Strom kann mittels der Funktion $\varphi^{nonp \to p} : \mathbb{S}^p \to \mathbb{S}_{pr}^p$ in einen priorisierten physischen Strom gleichen Typs transformiert werden. Die zeitliche Ordnung wird beibehalten.

$$\varphi^{nonpr \to pr}(S_T^p) := \{(e,[t_s,t_e), priority(e)) \in \Omega_T \times (\mathbb{T} \times \mathbb{T}) \times \mathbb{N}_0 | (e,[t_s,t_e)) \in S_T^p\}$$

Die Funktion $priority : S_T^p \to N_0$ bestimmt dabei die Priorität eines Elements anhand seines Dateninhalts.

Die umgekehrte Transformation eines priorisierten in einen „normalen" physischen Datenstrom (S^p) erfolgt durch Abbilden der priorisierten Elemente auf ihr unpriorisiertes Gegenstück

$$(e,[t_s,t_e),p) \mapsto (e,[t_s,t_e))$$

und Herstellen der zeitlichen Ordnung auf dem Datenstrom nach

$$<_{S^p} := \{(a,b) \in S^p \times S^p | a.t_s < b.t_s\}.$$

Über diesen Zwischenschritt lässt sich auch ein priorisierter physischer Datenstrom in einen logischen Strom transformieren, über den die Anfragesemantiken eindeutig definiert sind.

3 Physische Algebra

In diesem Kapitel stellen wir einen Auszug aus der von uns entwickelten physischen Operatoralgebra vor, die eine bevorzugte Verarbeitung priorisierter Datenstromelemente

erlaubt. Dazu führen wir einen neuen physischen Operator ein, der die Elemente eines „normalen" physischen Datenstroms priorisiert und dazu – wie im vorigen Kapitel bei der Umwandlung in priorisierte Datenströme beschrieben – jedem Element eine Priorität zuordnet, die sich anhand einer zu übergebenen Funktion aus den Daten des Elements berechnet.

In der Definition priorisierter physischer Datenströme können Elemente „out-of-order" – d.h. in nicht zeitlicher Ordnung – eintreffen. Die physischen Operatoren müssen teilweise an dieses Verhalten angepasst werden, damit korrekte Anfrageergebnisse garantiert werden können.

Die zustandslosen, satzorientierten Operatoren (Selektion, Projektion, ...) funktionieren ohne Änderung auch auf priorisierten Datenströmen, da die Prioritäten einfach ignoriert werden können.

Anders hingegen verhält es sich mit den zustandsbehafteten, mengenorientierten Operatoren. Diese lassen sich in zwei Klassen unterscheiden: In solche, die sinnvoll mit priorisierten Elementen umgehen können und solchen, die für eine Verarbeitung die ursprüngliche Stromreihenfolge benötigen, für die eine Priorisierung einzelner Elemente also irrelevant ist. Zu letzteren gehören beispielsweise der Aggregations- sowie der Sortieroperator.

Die notwendigen Anpassungen zur Sicherstellung der korrekten Semantik der anderen Operatoren beziehen sich in erster Linie auf das Aufräumen des internen Zustandsspeichers, da dies bei den mengenorientierten Operatoren anhand der Zeitstempel auf Grundlage der zeitlichen Ordnung des Datenstroms geschieht, die bei einem priorisierten Strom nicht mehr garantiert ist. Ein Element im Zustandsspeicher eines mengenorientierten Operators kann genau dann verworfen werden, wenn garantiert ist, dass kein zukünftiges Element mehr ein überlappendes Zeitintervall besitzt. Dies wird aufgrund der Ordnung des Eingabestromes bisher daran erkannt, dass der Startzeitstempel eines eingehenden Elements größer als der Endzeitstempel des betreffenden Elements im Zustandsspeicher ist.

Neben den notwendigen Änderungen der Operatoren zur Sicherstellung der semantischen Korrektheit gibt es weitere, die zur Latenzverbesserung bei der Verarbeitung priorisierter Elemente beitragen.

Unsere mengenorientierten Operatoren benutzen sogenannte SweepAreas [DSTW02] als Zustandsspeicher. Der abstrakte Datentyp SweepArea wird mit drei Eingabeparametern parametrisiert: eine Ordnungsrelation $\leq$ sowie zwei binäre Prädikate p_{query} und p_{remove}. Er stellt unter anderen folgende Methoden zur Verfügung, deren Verhalten von den Eingabeparametern beeinflusst wird:

query(element $s, j \in \{0, 1\}$**)** Liefert einen Iterator mit der Ordnung $\leq$ über alle Elemente $\hat{s}$ der SweepArea, für die das Prädikat p_{query} mit s und $\hat{s}$ als Parametern erfüllt ist. Der Parameter j bestimmt dabei, ob an s als erster und $\hat{s}$ als zweiter Parameter an p_{query} übergeben wird, oder ob $\hat{s}$ der erste und s der zweite Parameter sein soll.

purgeElements(element $s, j \in \{0, 1\}$**)** Entfernt alle Elemente $\hat{s}$ aus der SweepArea für die p_{remove} mit s und $\hat{s}$ als Parametern erfüllt ist. Der Parameter j bestimmt wie bei **query** die Reihenfolge, in der s und $\hat{s}$ übergeben werden.

Die folgende Definition eines Theta-Join-Operators benutzt diese Datenstruktur als internen Zustandsspeicher.

3.1 Kartesisches Produkt/Theta-Join-Operator

Beispielhaft wird im Folgenden ein Kartesisches Produkt/Theta-Join-Operator definiert, der korrekt mit priorisierten Datenströmen umgeht und bei der Erzeugung von Verbundelementen solche mit höherer Priorität unabhängig von den Startzeitstempeln anderer Elemente als erstes herausschreibt, so dass diese andere Elemente überholen.

Die möglichen Verbundpartner eines Elements werden mittels der `query`-Methode aus einer SweepArea ermittelt. Dafür wird folgendes Query-Prädikat verwandt, welches neben der Gültigkeit des Verbundprädikates $theta$ noch den Schnitt der Gültigkeitsintervalle einbezieht:

$$p^{\theta}_{query}(s, \hat{s}) \coloneqq \begin{cases} wahr & \text{wenn } \theta(e, \hat{e}) \wedge [t_s, t_e) \cap [\hat{t_s}, \hat{t_e}) \neq \varnothing, \\ falsch & \text{sonst.} \end{cases}$$

Elemente werden aus der SweepArea entfernt, wenn aufgrund der Ordnung der Datenströme in Zukunft keine Verbundpartner mehr eintreffen können. Um dies zu erkennen, können nur unpriorisierte Elemente ($p = 0$) dafür verwendet werden, da bei höher priorisierten Elementen nicht sichergestellt ist, dass diese nicht andere Elemente überholt haben und damit später weitere Verbundpartner eintreffen können.

$$p_{remove}(s, \hat{s}) \coloneqq \begin{cases} wahr & \text{wenn } t_s > \hat{t_e} \wedge p = 0 \\ falsch & \text{sonst.} \end{cases}$$

Die Ergebniselemente des Joins werden in einer nach der Ordnung $<^{p,t_s}$ aufsteigend sortierte Warteschlange eingefügt, wobei $<^{p,t_s}$ wie folgt definiert ist:

$$<^{p,t_s} \coloneqq \{(a, b) | a.p > b.p \vee (a.p = b.p \wedge a.t_s < b.t_s)\}$$

Damit werden die Ergebnisse nach absteigender Priorität ausgegeben und höher priorisierte Elemente können die weniger wichtigen in der Warteschlange überholen.

Die Priorität eines Ergebnisses der Verbundoperation berechnet sich aus den Eingabeelementen anhand einer zu übergebenen Funktion. Davon ausgehend, dass wichtige Informationen auch nach dem Verbund mit anderen Informationen wichtig sind, verwenden wir in unserem Anwendungskontext dafür das Maximum der Prioritäten der beiden Eingabeelemente.

Wir haben den in [Krä07] vorgestellten Algorithmus, der die Ripple-Join-Technik [HH99] auf push-basierte Verarbeitung überträgt, für die Verarbeitung priorisierter Datenströme angepasst.

Der in Algorithmus 1 definierte Join-Algorithmus verwendet eine Warteschlange zur Zwischenspeicherung für Verbundergebnisse und eine SweepArea für jeden Eingabedaten-

Algorithmus 1 : Kartesisches Produkt ($\times$)/Theta-Join ($\bowtie$)

Eingabe : Priorisierte physische Datenströme S_{in_0}, S_{in_1};
Funktion zur Berechnung von Prioritäten von Verbundelementen `newPriority`;

Ausgabe : Priorisierter physischer Datenstrom $S_{out} \leftarrow \varnothing$;

Daten : Seien SA_0, SA_1 leere SweepAreas($\leq^{t_e}, p^{\theta}_{query}, p_{remove}$);
$ts[0], ts[1], min_{ts} \in T \cup \perp; ts[0] \leftarrow \perp, ts[1] \leftarrow \perp, min_{ts} \leftarrow \perp$;
Sei Q eine nach $<_Q^{p,t_s}$ aufsteigend sortierte leere Prioritätswarteschlange;
Datenelement e';
$j, k \in \{0, 1\}$;

`/* j indiziert in jeder Iteration den Strom, aus dem in Zeile 1 das Element s eingetroffen ist. k referenziert den jeweils anderen Eingabestrom. */`

1 **foreach** $s \coloneqq (e, [t_s, t_e), p) \hookleftarrow S_{in_j}$ **do**
2 SA_k.purgeElements(*s,j*);
3 SA_j.insert(*s*);
4 Iterator $qualifies \leftarrow SA_k$.query(*s,j*);
5 **while** $qualifies$.hasNext() **do**
6 Element$(\hat{e}, [\hat{t_s}, \hat{t_e}), \hat{p}) \leftarrow qualifies$.next();
7 **if** $j = 0$ **then**
8 $e' = e \circ \hat{e}$
9 **else**
10 $e' = \hat{e} \circ e$
11 **end**
12 Q.insert($e', [t_s, t_e) \cap [\hat{t_s}, \hat{t_e})$, newPriority($(e, [t_s, t_e), p), (\hat{e}, [\hat{t_s}, \hat{t_e}), \hat{p})$));
13 **end**
14 $ts[j] \leftarrow t_s$;
15 **if** *p = 0* **then**
16 $min_{ts} \leftarrow \min(ts[0], ts[1])$;
17 **end**
18 TRANSFER(Q, min_{ts}, S_{out});
19 **end**
20 **while** $\neg Q$.isEmpty() **do**
21 Q.extractMin() $\hookrightarrow S_{out}$;
22 **end**

strom. Jedes eingehende Element eines Datenstroms wird zunächst in die eigene SweepArea geschrieben und, falls es die Priorität 0 hat, dafür verwendet den Zustand der SweepArea des anderen Datenstroms aufzuräumen (Zeilen 3 und 4). Danach werden mittels der `query`-Methode alle passenden Verbundpartner aus der entsprechenden SweepArea ermittelt und für jeden Partner ein Verbundresultat berechnet, das zunächst in die Ausgabewarteschlange eingefügt wird (Zeile 12). Insbesondere wird die Priorität des neuen Elementes aus den beiden Verbundpartnern berechnet.

Die `TRANSFER`-Funktion (siehe Algorithmus 2) sorgt dafür, dass die Elemente der Warteschlange, für die es möglich ist, in den Ausgabestrom geschrieben werden. Dazu wird über den Zeitstempel min_{t_s} bestimmt, ob noch Elemente auftauchen können, die aufgrund der zeitlichen Ordnung des Datenstroms vor vorhandenen Elementen herausgeschrieben werden müssen. Die von uns definierte schwache Ordnung priorisierter Datenströme (siehe Gleichung 2) verlangt, dass zur Berechnung von min_{t_s} nur nicht priorisierte Elemente verwendet werden, aber erlaubt es priorisierte Ergebnisse schon früher als andere Elemente herauszuschreiben.

Wenn die Eingabeströme erschöpft sind, müssen noch alle übriggebliebenen Elemente aus der Warteschlange herausgeschrieben werden (Algorithmus 1, Zeilen $20 - 22$).

Algorithmus 2 : TRANSFER-Funktion für schwach priorisierte Datenströme

```
   Eingabe :  Min-Prioritäts-Warteschlange Q;
              Zeitstempel min_ts; Ausgabestrom S_out;
 1 Iterator it ← Q.iterator();
 2 while it.hasNext() do
 3 |   Element (e, [t_s, t_e), p) ← it.next();
 4 |   if p > 0 ∨ t_e ≤ min_ts then
 5 |   |   it.remove();
 6 |   |   S_out ↩ (e, [t_s, t_e), p);
 7 |   else
 8 |   |   break;
 9 |   end
10 end
```

Wir haben uns bei der Realisierung für die Verwendung der schwachen Ordnung entschieden, da die starke Ordnung einige eklatante Nachteile hätte.

Die `TRANSFER`-Funktion für stark priorisierte Datenströme müsste sicherstellen, dass alle priorisierten Elemente gegenüber gleich oder höher priorisierten Elementen zeitlich geordnet sind. Das bedeutet, priorisierte Elemente würden erst dann herausgeschrieben, wenn ein nicht priorisiertes Element mit einem höheren Startzeitstempel in die Warteschlange eingefügt wird. Dies geschieht zu einem Zeitpunkt, zu dem das Element auch ohne jegliche Priorisierung herausgeschrieben worden wäre. Es bliebe einzig der Gewinn durch das Überholen in der Warteschlange selbst. Überholen von Elementen vor dem Operator hätte keinen Einfluss mehr auf die Latenz, da wieder auf das Eintreffen der überholten Elemente gewartet werden müsste. Damit hat ein priorisiertes Element dann zwar andere Elemente

überholt (evtl. mehr als in einem schwach priorisierten Datenstrom), die Latenz, mit der das Ergebnis bereitsteht, steigt aber, weil es erst viel später in den Ausgabestrom herausgeschrieben wird. Das heißt, in einem Plan würde nur noch der oberste Verbund- bzw. binäre, zustandsbehaftete Operator zu einem Gewinn bei der Latenz beitragen können, der zudem (je nach Verarbeitungskosten der darauf folgenden Operatoren) eher marginal ist. Aus diesem Grund haben wir uns zur Verwendung der schwachen Ordnung $<^{weak}_{S^p_{pr}}$ für priorisierte Datenströme entschieden, die diesen Nachteil nicht besitzt.

Im Folgenden soll nun skizzenhaft gezeigt werden, dass der von uns vorgestellte Join-Algorithmus auch die gewünschten Ergebnisse erzeugt und insbesondere semantisch äquivalent zum Algebraoperator in [Krä07] ist.

3.1.1 Semantische Korrektheit des Verbundoperators

Wenn ein Element eingelesen wird, gibt es zwei Möglichkeiten. Wenn Verbundpartner bereits eingelesen wurden, finden sie sich in der komplementären SweepArea und werden über die `query`-Methode ermittelt. Wenn Verbundpartner erst später im anderen Datenstrom eintreffen, ist das Element seinerseits aber in der SweepArea und wird selbst über die `query`-Methode zurückgeliefert. Dass die `query`-Methode nur korrekte Verbundpartner zurückgibt, lässt sich leicht über das Prädikat p_{query} nachvollziehen.

Damit der Algorithmus also korrekte Ergebnisse liefert, muss nur noch sichergestellt sein, dass kein Element aus einer SweepArea entfernt wird, für das später noch weitere Verbundpartner eintreffen können. Zu diesem Zweck soll das Remove-Prädikat genauer untersucht werden. Seien dazu in_j das zuletzt gelesene Element eines Eingabestroms S_{in_j} und SA_i mit $i \neq j$ die Menge der Elemente der SweepArea des anderen Eingabestroms. Sollte die Priorität von in_j größer als 0 sein, gilt automatisch $\forall e \in SA_i : \neg p_{remove}(in_j, e)$.

Es bleibt also zu zeigen:

$$in.p = 0 \Rightarrow \nexists e \in SA_i : p_{remove}(in_j, e) \wedge \exists s \in S_{in_j} : \neg(s <^{weak}_{S_{in_j}} in_j) \wedge p_{query}(s, e).$$

Angenommen es gäbe solch ein Element e, dann ist $e.t_e < in_j.t_s$ (wegen $p_{remove}(in_j, e)$). Außerdem muss es ein Element $s \in S_{in_j}$ mit $\neg(s <^{weak}_{S_{in_j}} in_j) \wedge p_{query}(s, e)$ geben. Das bedeutet

$$\begin{aligned} & \exists s \in S_{in_j} : s.t_s \geq in_j.t_s \wedge p_{query}(s, e) \\ \Rightarrow \quad & \exists s \in S_{in_j} : s.t_s \geq in_j.t_s \wedge [s.t_s, s.t_e) \cap [e.t_s, e.t_e) \neq \varnothing \\ \Rightarrow \quad & \exists s \in S_{in_j} : s.t_s \geq in_j.t_s \wedge s.t_s < e.t_e. \end{aligned}$$

Dies steht im Widerspruch zu $e.t_e < in_j.t_s$, womit gezeigt wurde, dass kein Element aus einer SweepArea entfernt wird, wenn später noch weitere Verbundpartner eintreffen können.

Es bleibt noch zu zeigen, dass der Ausgabedatenstrom auch der korrekten Ordnung unterliegt.

3.1.2 Korrektheit der Ordnung des Ausgabedatenstroms

Der von einem Operator erzeugte Ausgabedatenstrom S_{out} mit der Ordnung $<$ muss bzgl. $<^{weak}_{S_{out}}$ sortiert sein. Im Folgenden zeigen wir, dass der von Algorithmus 1 erzeugte Datenstrom diese Bedingung erfüllt.

Es sei $t_{s,min,k} = min(t_{s,1}, t_{s,2})$ mit $t_{s,j} = max(\{t_s | (e, [t_s, t_e), 0) \in S_{in_j,k}\})$ für $j \in \{0, 1\}$ der minimale Startzeitstempel der bis zum k-ten Durchlauf eingelesenen nicht-priorisierten Elemente der Eingabedatenströme des Joins. Sei weiterhin $S_{out,k}$ die Menge der nach dem k-ten Durchlauf durch die Funktion `TRANSFER` in den Ausgabedatenstrom geschriebenen Elemente.

Es gilt $<^{weak}_{S_{out,k}} \subseteq <$, denn:

$$\begin{array}{rll} & \forall (a,b) \in <^{weak}_{S_{out,k}}: & a.t_s < b.t_s \wedge b.p = 0 \\ \overset{\text{wgn. } b.p=0}{\Longrightarrow} & \forall (a,b) \in <^{weak}_{S_{out,k}}: & a.p > b.p \vee (a.p = b.p \wedge a.t_s < b.t_s) \\ \Longrightarrow & \forall (a,b) \in <^{weak}_{S_{out,k}}: & (a,b) \in <^{p,t_s}_{S_{out,k}} \end{array} \tag{3}$$

Es bleibt zu zeigen, dass $<^{weak}_{S_{out}}$ für $S_{out} = \bigcup_{k=1}^{\infty} S_{out,k}$ gilt, also nicht nur die im k-ten Durchlauf erzeugte Ausgabe der Ordnung unterliegt, sondern der gesamte Ausgabestrom. Auch diese Bedingung ist erfüllt, da die Funktion `TRANSFER` im k-ten Durchlauf nur jeweils alle Elemente s_Q bzgl. $<^{p,t_s}_{Q}$ sortiert aus der Warteschlange Q in den Ausgabedatenstrom S_{out} schreibt, für die $s_Q.t_s < t_{s,min} \vee s_Q.p > 0$. Damit ist $max(\{t_s | (e, [t_s, t_e), 0) \in S_{out,k}\}) \leq t_{s,min,k}$. Das bedeutet, dass danach keine weiteren Elemente mehr in die Queue eingefügt werden können, die nach $<^{weak}_{S_{out}}$ vor einem der herausgeschriebenen Elemente liegen sollten.

Damit wurde gezeigt, dass der Ausgabedatenstrom, welcher von Algorithmus 1 erzeugt wird, bzgl. $<^{weak}_{S_{out}}$ sortiert ist.

3.1.3 Abschätzung zur Verbesserung der Latenzen

Um eine Abschätzung darüber vornehmen zu können, wie sich der Algorithmus 1 auf die Verbesserung der Latenzen auswirken kann, müssen zunächst einige Randbedingungen festgelegt werden. In einem allgemeinen Datenstromanfrageplan spielen sehr viele Parameter, wie Scheduling-Strategien, Zusammensetzung des Anfrageplans oder auch die zeitliche Reihenfolge der Elemente im Datenstrom eine Rolle. Wir betrachten bei dieser Abschätzung daher einen Queryplan mit nur einem „priorisierten Join“, der sich vorteilhaft auf die Latenzen der Elemente auswirkt und die Selektivität σ hat. Weiterhin seien w_j die Fensterbreiten auf Datenströmen S_j sowie f_j die Frequenzen, mit denen Elemente in den Datenströmen S_j auftauchen, wobei $j \in \{0, 1\}$ gilt. Wir gehen davon aus, dass die Fensterbreiten vor dem Eintreffen der Elemente im hier betrachteten Join nicht verändert werden, bspw. durch einen vorhergehenden Join. Weiterhin sei t_{plan} die nach dem Join verbleibende Verarbeitungszeit eines einzelnen Elementes im Anfrageplan. Der Einfachheit halber gehen wir hier von einer Single-Thread-Scheduling-Strategie aus, so dass die Verarbeitungszeit des verbleibenden Anfrageplans für alle Elemente proportional zu t_{plan} ist. Es

ist leicht einzusehen, dass sich für die am höchsten priorisierten Elemente im schlechtesten Fall kein Vorteil durch das Überholen nicht priorisierter Elemente ergibt, da in diesem Fall ein hoch priorisiertes Element zwar eher im Eingabedatenstrom eines Joins auftaucht, jedoch noch keine Verbundpartner vorhanden sind und alle bereits überholten Elemente vorher ihre Verbundpartner vorfinden. Für den besten Fall, dass bereits alle Verbundpartner des überholenden Elementes vorhanden sind, lässt sich jedoch eine Abschätzung vornehmen, wie groß der Gewinn ist, der durch das Überholen im Join erreicht werden kann.

Hierzu sei o. B. d. A. angenommen, dass ein priorisiertes Element e_p, welches bereits x niedriger priorisierte Elemente überholt hat, im linken Eingabedatenstrom des Joins auftaucht. Wenn bereits alle Verbundpartner für dieses Element im rechten Eingabedatenstrom vorhanden sind, dann ist die Ausgabewarteschlange des Join fast leer. In ihm befinden sich noch genau die Verbundelemente, die durch das letzte Element des linken Eingabedatenstrom erzielt wurden. Dies sind im Durchschnitt $\sigma \cdot w_0 \cdot f_1$ Verbundelemente. Diese Ergebnisse können nun durch das neue priorisierte Element überholt werden.

Weiterhin können die Verbundelemente von e_p sofort herausgeschrieben werden, auch wenn ihr Startzeitstempel größer als min_{t_s} ist. Sie müssen nicht darauf warten, dass Elemente eintreffen, die min_{t_s} über ihren Startzeitstempel hinaus erhöhen. Damit überholt jedes Verbundelemente von e_p im Schnitt zusätzliche $\sigma \cdot \sum_{n=1}^{w_0 \cdot f_0} (w_1 + \frac{n}{f_0}) \cdot f_1 \cdot \frac{\cdot w_0}{w_0 + w_1}$ Elemente. Außerdem muss nicht auf das Eintreffen der entsprechenden anderen Elemente nach e_p gewartet werden, so dass im Schnitt noch $\frac{w_0^2}{2 \cdot (w_0 + w_1)}$ gewonnen wird (hierbei wird der Verständlichkeit halber davon ausgegangen, dass die Fenster in Systemzeit und nicht in einer davon unabhängigen Applikationszeit definiert sind).

Geht man jetzt davon aus, dass jedes dieser Element noch eine Verarbeitungszeit t_{plan} im verbleibenden Anfrageplan verursacht und die Verarbeitungszeit eines Elementes im Join durchschnittlich t_{join} ist, dann ergibt sich ein durchschnittlicher Gewinn für die Latenzen von Verbundelementen von e_p zu

$$t_{win} = x \cdot t_{join} + \frac{w_0^2}{2 \cdot (w_0 + w_1)} + \sigma \cdot (\sum_{n=1}^{w_0 \cdot f_0} (w_1 + \frac{n}{f_0}) \cdot \frac{w_0}{w_0 + w_1} + w_0) \cdot f_1 \cdot t_{plan}$$

4 Überholen von Elementen im Datenstrom

Mit Hilfe der Operatoren lässt sich bereits eine Verbesserung der Latenz eines priorisierten Datenstromelements erreichen. Neben diesem intraoperatorbasierten Ansatz bietet es sich ebenfalls an, ein Überholen auch zwischen den Operatoren (Inter-Operator) zu ermöglichen, was zum einen weiteres Optimierungspotential bietet und zum anderen auch Pläne ohne spezielle (binäre) Operatoren vom Überholen profitieren lässt.

Wir gehen in unserer Arbeit von einem push-basierten Ansatz aus, d.h. Operatoren (Quellen) geben ihre produzierten Daten an Nachfolger (allgemein Senken) weiter. Für die Quelle ist es prinzipiell irrelevant, was die Senke mit den Daten macht. Insbesondere spielt es keine Rolle, ob der Nachfolger tatsächlich ein Operator oder ein zwischenspeichernder

Puffer ist. In der Abbildung 2 ist beispielhaft der Aufbau eines Operatorplans dargestellt.

Am unteren Ende befinden sich die Quellen, die Daten für die Senken produzieren. Zwischen Quelle und Senke kann dabei ein Puffer eingefügt werden. Ein Scheduler sorgt dafür, dass die untersten Quellen eines Planes und die Puffer ihre Daten weiterleiten. Wenn zwischen zwei Operatoren kein Puffer eingefügt ist, blockiert der unterste Operator so lange, bis der oberste Operator seine Daten geschrieben hat. Erst durch die Einführung von Puffern kann ein effektives Scheduling stattfinden. Im markierten Pfad in Abbildung 2 würde ein Tupel immer den kompletten Pfad nach dem Puffer durchlaufen, da es nicht zwischengespeichert werden kann, so dass keine Operatoren im Pfad parallel ausgeführt werden können. Für das Überholen im Datenstrom spielen also nur die Stellen eine Rolle, an denen sich Puffer zwischen zwei Operatoren befinden.

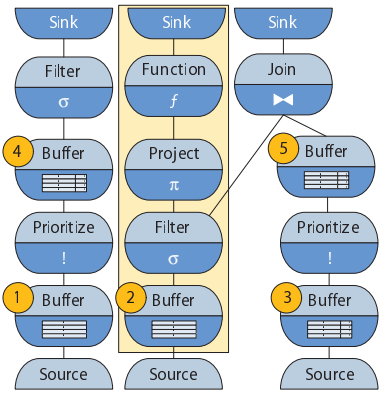

Abbildung 2: Aufbau eines Operatorplans mit Quellen und Senken

Grundsätzlich sind die in der Abbildung 3 dargestellten Ansätze für das Überholen im Datenstrom zwischen Operatoren möglich. Zu sehen sind jeweils zwei Operatoren und ein dazwischen geschalteter Puffer. In den Puffern befinden sich Datenstromelemente, die jeweils mit einer Priorität versehen sind. Ein Element mit Priorität 0 ist nicht priorisiert.

Im ersten, *direct mode* genannten Ansatz werden in den Puffern nur Elemente ohne Priorität gespeichert und Elemente mit einer Priorität direkt an die nachfolgende Senke weitergeleitet. Priorisierte Elemente werden also direkt im Kontext des selben Threads weiterverarbeitet, womit in diesem Fall der untere Operator blockiert. Aus diesem Grund sind solche Puffer nie direkt über den untersten Operatoren eines Planes einsetzbar. Ansonsten könnte es passieren, dass der Thread für den Quellenoperator so lange mit der weiteren Verarbeitung priorisierter Elemente beschäftigt ist, dass dieser eingehende Elemente „verpasst“. Außerdem besteht ein erhöhter Synchronisationsbedarf in den Operatoren, da die Verarbeitungsmethode nun gleichzeitig mit zwei Elementen aus dem selben Eingabestrom aufgerufen werden kann. Wird bspw. für die Verarbeitung ein Plan vertikal geteilt und ein Thread führt die Operatoren des unteren Teils aus und ein anderer die des oberen, kann es passieren, dass der untere Thread – durch die direct-Puffer – Elemente in obere Operatoren pusht, während diese gerade vom oberen Thread ausgeführt werden.

Der zweite Ansatz (*weak order mode*) fügt alle nicht priorisierten Elemente wie gehabt an das Ende des Puffers ein. Priorisierte Elemente werden ohne Berücksichtigung der Priorität an den Anfang des Puffer geschrieben. Dies hat den Vorteil, dass beim Einfügen eine Sortierung priorisierter Elemente vermieden werden kann, hat aber den Nachteil, dass unterschiedliche Prioritäten nicht unterschiedlich gewichtet werden.

Der letzte Ansatz (*strong order mode*) sorgt schließlich dafür, dass wenn mehrere priorisierte Elemente im Puffer vorhanden sind, diejenigen mit der höchsten Priorität auch am weitesten nach oben geschrieben werden, allerdings auf Kosten teurerer Einfügeoperatio-

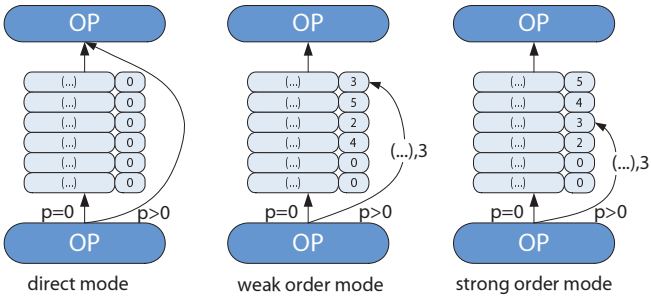

Abbildung 3: Puffermodi zur Bevorzugung priorisierter Datenstromelemente

nen.

5 Evaluation

Wir haben den Ansatz zur prioritätsbasierten Verarbeitung von Datenstromelementen im Rahmen unseres Java-basierten ODYSSEUS-Frameworks [JG08] für Datenstrommanagementsysteme implementiert und evaluiert. Unsere Messungen haben sich zunächst darauf beschränkt, zu untersuchen, in wie weit sich die Latenzen für einzelne sich im Datenstrom befindende Elemente mit und ohne Prioritäten verändern. Dabei haben wir es uns zu Nutze gemacht, dass wir in ODYSSEUS an das eigentliche Datenobjekt nahezu beliebige Metadaten anhängen können. In unserem Fall haben wir die Latenz durch die Vergabe von Zeitstempeln messen können: Beim Eintritt in den Plan wurde ein Element mit einem Startzeitstempel versehen und beim Erreichen des obersten Operators mit dem passenden Endzeitstempel, die Differenz ist die Latenz des Elements. Um nicht von Datenverteilungen in Eingabedatenströmen abhängig zu sein, haben wir statt echter Operatoren nur simulierte Benchmark-Operatoren im Anfrageplan verwendet, die jeweils mit Ausführungszeiten und Selektivitäten konfiguriert werden konnten. Die Benchmark-Operatoren verändern die Zeitstempel dabei nicht. So können wir sicher sein, dass wir nur die Effekte messen, die sich durch das Überholen in den Puffern ergeben und die nicht durch spezielle Datenverteilungen bedingt sind. Es bedeutet aber auch, dass nicht das volle Optimierungspotential unseres Ansatzes genutzt wurde, da so keine Vorteile aus den neuen Operatorimplementierungen gezogen werden konnte. Der einzige reguläre Operator ist der Operator zum Setzen der Prioritäten eines Elements. Genauso wie die Latenzinformation wird die Priorität eines Elements auch mit Hilfe von angehängten Metadaten realisiert.

Wir haben für unsere Tests einen komplexen Anfrageplan mit einer maximalen Tiefe von neun bestehend aus ca. 60 Operatoren (plus Puffer) verwendet. Die Tests fanden auf einem Intel Core 2 Duo mit 2,4 Ghz und 2 CPUs, mit 3,2 GByte Hauptspeicher und einem CentOS 5.2 Linux statt.

In den Tests wurden die Anfragen mit verschiedenen, teilweise aus der Literatur bekannten, Scheduling-Strategien und mit den unterschiedlichen Pufferkonzepten ausgeführt. Im

Einzelnen haben wir die folgenden Strategien für das Scheduling verwendet:

Round Robin: Die Operatoren werden einfach in der Reihenfolge eingeplant, wie sie im Plan vorkommen. Wenn ein Operator keine Daten zum Verarbeiten vorliegen hat, wird der nächste Operator der Liste betrachtet. Zwei spezielle Ansätze des Round Robin sind die beiden folgenden, die eine spezielle Reihenfolge der Operatoren festlegen.

Aurora Min-Cost: Die aus dem Aurora-System stammende Strategie [CcR+03] führt eine topologische Sortierung des Anfrageplans durch, wodurch sicher gestellt wird, dass kein Operator zur Verarbeitung angestoßen wird, dessen Eingabeoperatoren zu einem früheren Zeitpunkt hätten Daten produzieren müssen. Die Verarbeitung wird also insgesamt von unten nach oben und mit möglichst geringem Scheduling-Overhead durchgeführt. Eine mögliche Scheduling-Reihenfolge für den Anfrageplan in Abbildung 2 wäre also $1, 2, 3, 4, 5$.

Aurora Min-Latency: Bei dieser ebenfalls aus dem Aurora-System stammenden Strategie wird versucht, die Latenz für ein einzelnes Element im Strom zu reduzieren, in dem immer komplette Pfade von den Operatoren zu den Senken eingeplant werden. Die Reihenfolge der Pfade ist dabei aufsteigend nach der Latenz, die sie für ein Element verursachen, festgelegt. Eine mögliche Scheduling-Reihenfolge für den Anfrageplan in Abbildung 2 wäre also $4, 1, 4, 5, 3, 5, 2$. Auf Grund fehlender Kostenmodellinformationen konnten wir nicht die Originalversion der Strategie verwenden, sondern verzichten auf eine Sortierung der Pfade nach der größten Produktivität. Dies sollte sich eigentlich auch nur zu Beginn der Verarbeitung für die ersten Tupel auswirken und ist aus diesem Grund zu verkraften.

Biggest-Queue: Dies ist eine dynamische Strategie, die bei jedem Scheduling-Aufruf immer die Operatoren ausführt, deren Eingabepuffer die meisten Daten enthält.

Highest-Priority-Queue: Diese Strategie ist schließlich die einzige, die die Priorität der Daten berücksichtigt. Es werden immer die Operatoren angestoßen, deren vorgelagerter Puffer das Element mit der höchsten Priorität enthält.

In der Standardimplementierung verarbeitet jede Strategie in jedem Durchlauf genau ein Element und wählt dann anschließend einen neuen Operator. Zusätzlich können alle Strategien noch in einem speziellen Modus bei jedem Durchlauf eine Menge oder gar alle Elemente des Puffers verarbeiten (in [ACc+03] Train-Scheduling genannt). Zur Unterscheidung haben wir diesen Modus durch ein angehängtes '+'-Zeichen gekennzeichnet.

Insgesamt haben wir die Scheduling-Ansätze mit fünf verschiedenen Pufferrealisierungen getestet.

normal: Dieser Ansatz stellt die Vergleichsbasis dar und unterscheidet nicht zwischen unterschiedlichen Prioritäten. Die Elemente werden einfach in einer verketten Liste gespeichert und nach dem FIFO-Prinzip verarbeitet. Um vergleichbare Laufzeiten zwischen verschieden priorisierten Elementen zu bekommen, wird aber auch in diesem Plan der Operator zum Setzen der Priorität verwendet. Dieser trägt aber mit

Verarbeitungszeiten im Mikrosekundenbereich nicht zu einer relevanten Verschlechterung der Latenz bei.

direct: Wir bereits oben beschrieben, werden hier priorisierte Elemente immer sofort an den Nachfolge-Operator geschoben. Die anderen Operatoren werden in eine Liste eingefügt und dann nach dem FIFO-Prinzip verarbeitet.

weak order: Dies ist der Ansatz, bei dem alle Elemente in den Puffer geschrieben werden, priorisierte Element aber nach dem LIFO-Prinzip verarbeitet werden. Sie werden also an den Kopf der Liste gesetzt.

strong order: In diesem Ansatz werden die priorisierten Elemente an den Anfang der Liste geschrieben, allerdings bezüglich ihrer Priorität einsortiert. In der Liste befinden sich also gleichzeitig priorisierte und nicht priorisierte Elemente.

strong order2: In Analogie zum *strong order*-Puffer werden auch hier die priorisierten Elemente nach ihrer Priorität sortiert, jedoch gibt es für priorisierte und für nicht priorisierte Elemente unterschiedliche Listen.

Die *strong order*-Umsetzung hat sich in ersten Tests als nicht verwendbar herausgestellt, da sie – aufgrund des erhöhten Verarbeitungsaufwands beim Einfügen – in allen Tests zu einer Vervielfachung der Latenz geführt hat. Aus diesem Grund haben wir diese Realisierung nicht weiter betrachtet. Das andere Extrem, für jede Priorität eine eigene Liste zu nutzen, ist bisher auch nicht weiter betrachtet worden und könnte noch einmal genauer untersucht werden.

Die Art, wie in dem Anfrageplan die Puffer platziert werden, ist noch statisch: Nach jeder Quelle, vor jedem zustandsbehafteten binären Operator und nach dem Prioritätsoperator wird entsprechend der Pufferrealisierung ein Puffer eingefügt. Eine Besonderheit ergibt sich, wenn der *direct*-Ansatz verwendet wird: In diesem Fall wird ein normaler Puffer und keine der genannten Spezialimplementierungen hinter die Quelle gesetzt um ein Blockieren des Quellen-Threads zu vermeiden. Auch bei der Platzierung von Puffern im Anfrageplan sind noch Untersuchungen von anderen Strategien denkbar und sinnvoll.

Getestet wurde das Verhalten mit einer konstanten Datenrate von 10000 Elementen pro Sekunde und einer Gesamtanzahl von 100000 Elementen, die jede der Quellen produziert. Insgesamt werden durch die Anfragen $1,4$ Millionen Elemente verarbeitet. Für jede Prioritätsstufe größer 0 werden dabei Elemente mit einer Wahrscheinlichkeit von 10% erzeugt, d.h. bspw. wenn es drei unterschiedliche Prioritäten $(0, 9, 10)$ gibt, dann sind etwa 20% aller Elemente priorisiert.

Ein Ergebnis welches relativ schnell offensichtlich wurde, ist dass sich der *strong order2*-Ansatz umso schlechter verhielt, je mehr unterschiedliche Prioritäten im Plan vorkommen, insbesondere in Kombination mit einer '+' Strategie. Dies ist auch nicht weiter verwunderlich, da das Einsortieren in die Liste mehr Zeit kostet, je mehr priorisierte Elemente bereits in der Liste enthalten sind.

Im Folgenden haben wir nur noch Anfragen mit vier unterschiedlichen Prioritäten, d.h. 0 für unpriorisierte Elemente und $8, 9$ und 10 als priorisierte Elemente, betrachtet. Hierbei wurde deutlich, dass trotz des hohen Anteils an priorisierten Elementen, sich keine

erkennbaren Unterschiede für die einzelnen Prioritätsstufen ausmachen ließen – *strong order2* also keinen Mehrwert brachte. Hier muss noch genauer untersucht werden, ob andere Prioritätsverteilungen im Strom Auswirkungen haben könnten. In der Abbildung 4 sind aus diesem Grund nur noch die beiden *direct*- und *weak order*-Pufferverfahren jeweils im Verhältnis zum *normal*-Pufferverfahren gegenüber gestellt. Ein Wert von $0,5$ bedeutet dabei, dass sich die durchschnittliche Latenz im Verhältnis zum Verfahren ohne Berücksichtigung der Prioritäten halbiert hat, ein Wert von 2 bedeutet, dass sie sich verdoppelt hat.

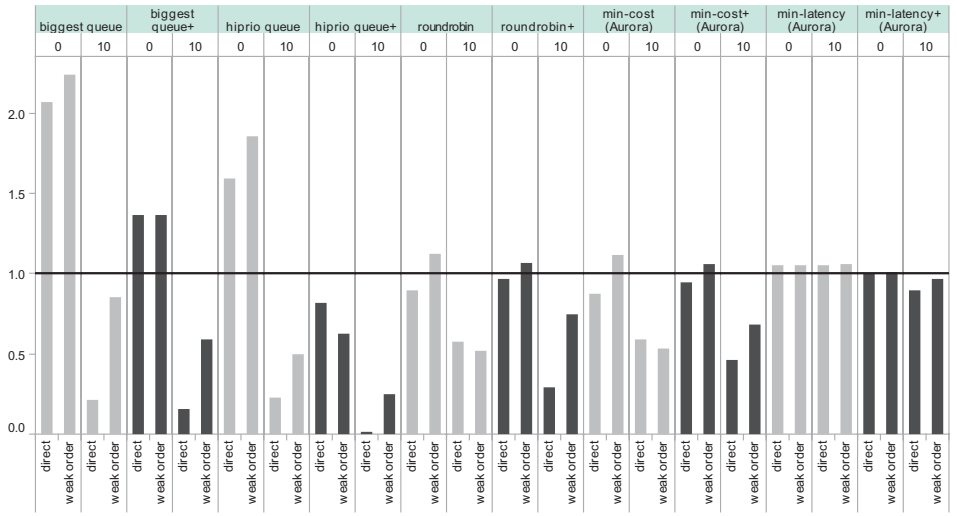

Abbildung 4: Änderung der Latenz gegenüber Normal-Puffermodus

Es fällt auf, dass in fast allen Verfahren eine Verbesserung der Latenz priorisierter Elemente zu verzeichnen ist. Die Verbesserung fällt dabei fast durchgehend sehr deutlich aus. Ebenso fällt auf, dass der *direct*-Ansatz fast immer zu einer größeren Verbesserung führt, als die *weak order*-Puffer. Dies war zu erwarten, da sich der Nachteil dieses Ansatzes (Blockierung des unteren Threads) erst bei sehr teuren Anfrageplänen (also welchen mit hohen Verarbeitungszeiten pro Element) bemerkbar machen sollte. Hier wäre also noch zu untersuchen, ab wann die Nachteile des *direct*-Ansatzes überwiegen. Bei der *Highest-Priority-Queue* lässt sich die größte Latenzverbesserung für priorisierte Elemente feststellen. Dies liegt daran, dass es die einzige Strategie ist, die das Scheduling an den priorisierten Elementen orientiert. Dadurch, dass zunächst Puffer ausgewählt werden, die priorisierte Elemente enthalten, lassen sich in Verbindung mit den *direct*-Puffern Verbesserungen von mehr als Faktor fünfzig beobachten. Grundsätzlich sind die Strategien, die mehrere Elemente auf einmal pro Puffer verarbeiten, durch den verringerten Scheduling-Overhead besser.

Die Laufzeiten der Anfragen und damit der Durchsatz scheinen nicht von den Puffermodi beeinflusst zu werden. Ausnahmen bilden die *Highest-Priority-Queue*- und die *Biggest-Queue*-Strategie, wenn diese nach jedem Element einen neuen Puffer für das Scheduling

auswählen (in Abbildung 4 ohne angehängtes '+' dargestellt). In den beiden Fällen steigen die Laufzeiten und wie in Abbildung 4 zu sehen auch die Latenzen für normale Elemente deutlich. Hierbei scheint im getesteten Szenario der Scheduling-Overhead so groß zu werden, dass es zu einer Art Überlast des Systems kommt und sich die Puffer nach den Quellenoperatoren zunächst stark füllen, bevor sie dann abgearbeitet werden.

Die beste durchschnittliche Latenz über alle Elemente liefert die *Aurora Min-Latency*-Strategie. Bei ihr ließen sicher aber leider so gut wie keine Gewinne durch die Priorisierung erzielen. Für die Zukunft scheint daher eine Verbindung der *Highest-Priority-Queue*-Strategie mit der *Min-Latency*-Strategie vielversprechend zu sein, wenn es auf die Optimierung der Gesamtlatenz unter zusätzlicher Berücksichtigung der Prioritäten ankommt.

6 Verwandte Arbeiten

Eine effiziente Verarbeitung von Datenströmen ist entscheidend auch von der Strategie abhängig, mit der die Operatoren eines Anfrageplans zur Ausführung gebracht werden [SCLP08]. Eine Reihe von Arbeiten hat sich bereits mit diesem Thema befasst. Stellvertretend sollen hier die Arbeiten zum Chain-Scheduling [BBDM03] und in Aurora [CcR+03], in NiagaraCQ [HFAE03] und PIPES [CHK+07] genannt werden. Alle uns bekannten Ansätze betrachten zum Priorisieren die Operatoren und optimieren ihre Strategien i.d.R. auf einen hohen Durchsatz, lediglich Aurora erlaubt die Zuweisung von Quality-of-Service Ansprüchen an Anfragen, wodurch eine gewisse Priorisierung auf Anfrageebene erfolgen kann. Ebenfalls nur auf Operatorebene betrachtet [UF01] die Priorisierung von Operatoren innerhalb einer Anfrage. Im Gegensatz dazu ist es mit unserem Ansatz möglich, unterschiedliche Prioritäten innerhalb des selben Pfades eines Anfrageplans zu betrachten.

Eine wesentliche Vorrausetzung für unseren Ansatz stellt die "out-of-order"-Verarbeitung von Datenstromelementen dar. In der Arbeit von [LTS+08] wurde gezeigt, dass diese Art der Verarbeitung im Kontext von Verzögerungen zu geringerem Speicherverbrauch, einer geringeren durchschnittlichen Latenz sowie zu einer kürzeren Laufzeit der Anfragen führen kann. Genau wie in [LTS+08] werden auch in [DR04] Punctuations genutzt, um ein Nicht-Blockieren von zustandsbehafteten Operatoren zu realisieren. In letzterer Arbeit werden sie nebem dem Einsatz von Zeitfenstern dazu genutzt, die Erzeugung von Ergebnissen in einem Join-Operator zu beschleunigen und damit ebenfalls eine geringere durchschnittliche Latenz zu erzielen. Beim Einsatz von Punctuations besteht jedoch die Gefahr, dass evtl. Tupel verworfen werden müssen, falls eine Punctuation an einem Eingabedatenstrom auf Grund fehlender oder fehlerhafter Fortschrittsinformationen über den Datenstrom zu früh erzeugt wird (vgl. [SW04]). Dieses Problem ergibt sich auch beim Schätzen der maximalen Verspätung eines Tupels (Slack), wie es beispielsweise in [ACc+03, LLD+07] angewandt wird. Wir stellen mit dieser Arbeit eine „out-of-order"-Verarbeitung einzelner Datenstromelemente vor, die ohne Punctuations oder Slack-Parameter auskommt und zusätzliche Optimierungsmöglichkeiten bei der Verarbeitung dieser Elemente bietet.

7 Fazit

Es gibt Anwendungsfälle für Datenstrommanagementsysteme (wie die Überwachung von Energieerzeugungsanlagen), in denen die Latenz bei der Verarbeitung besonderer Datenstromelementen (wie Alarme, Warnungen) wichtig ist. Bisherige Ansätze zur Anfragepriorisierung helfen bei diesem Problem nicht weiter, da ein entsprechendes Element im Extremfall in allen Anfragen gleichzeitig verarbeitet werden muss. Unsers Wissens nach sind wir die ersten, die ein Verfahren vorstellen, das es ermöglicht, wichtige Datenstromelemente bei der Verarbeitung zu bevorzugen. Wir haben in dieser Arbeit eine formale Grundlage für die Priorisierung einzelner Elemente in Datenströmen geschaffen. Der vorgestellte Ansatz bietet gleich zwei Vorteile bei der Verarbeitung priorisierter Elemente. Zum einen ermöglicht er es priorisierten Elementen andere Elemente im Datenstrom zu überholen und zum anderen können priorisierte Ergebnisse zustandsbehafteter Operatoren schneller erzeugt werden, als die normale Verarbeitung dies erlauben würde. Dabei wird die Anfragesemantik nicht geändert. Damit dies gewährleistet ist, haben wir die Algorithmen der zustandsbehafteten Operatoren angepasst. In dieser Arbeit wurde exemplarisch ein prioritätskompatibler Joinalgorithmus vorgestellt und nicht nur gezeigt, dass dieser korrekte Ergebnisse erzeugt, sondern auch eine Abschätzung des Gewinns bei der Latenz für priorisierte Elemente geliefert.

Wir haben verschiedene Strategien für Puffer in Anfrageplänen entwickelt, die priorisierte Elemente bevorzugt behandeln. In einer Evaluation mit diversen Schedulingstrategien und unseren neuen Puffermodi konnte ein meist sehr deutlicher Gewinn bei der durchschnittlichen Latenz festgestellt werden. Eine in dieser Arbeit neu entwickelte Scheduling-Strategie, die die Priorisierung von Elementen berücksichtigt, konnte besonders starke Reduktionen der Latenz erzielen.

Zukünftig wollen wir – neben ausführlicheren Tests weiterer Szenarien – vor allem die Verbindung von Schedulingstrategien mit elementbasierter Priorisierung untersuchen. Die Latenz bei Verbünden mit hoch priorisierten Elementen hängt auch vom rechtzeitigen Eintreffen der Verbundpartner ab. Daher wollen wir Techniken untersuchen, mit deren Hilfe diese schneller bereitgestellt werden können.

Danksagung: Wir danken Martin Hecker für seine wertvollen Hinweise.

Literatur

[ABW06] Arvind Arasu, Shivnath Babu und Jennifer Widom. The CQL continuous query language: semantic foundations and query execution. *The VLDB Journal*, 15(2), 2006.

[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), 2003.

[BBDM03] Brian Babcock, Shivnath Babu, Mayur Datar und Rajeev Motwani. Chain : Operator Scheduling for Memory Minimization in Data Stream Systems. In Alon Y. Halevy,

Zachary G. Ives und AnHai Doan, Hrsg., *SIGMOD 2003: Proc. of the 2003 ACM SIGMOD International Conference on Management of Data*. ACM, 2003.

[CcR+03] Donald Carney, Ugur Çetintemel, Alex Rasin, Stanley B. Zdonik, Mitch Cherniack und Michael Stonebraker. Operator Scheduling in a Data Stream Manager. In *VLDB 2003: Proc. of 29th International Conference on Very Large Data Bases*, 2003.

[CHK+07] Michael Cammert, Christoph Heinz, Jürgen Krämer, Bernhard Seeger, Sonny Vaupel und Udo Wolske. Flexible Multi-Threaded Scheduling for Continuous Queries over Data Streams. In *International Workshop on Scalable Stream Processing Systems*, 2007.

[DR04] Luping Ding und Elke A. Rundensteiner. Evaluating window joins over punctuated streams. In *CIKM '04: Proc. of the thirteenth ACM international conference on Information and knowledge management*, New York, NY, USA, 2004. ACM.

[DSTW02] Jens-Peter Dittrich, Bernhard Seeger, David Scot Taylor und Peter Widmayer. Progressive Merge Join: A Generic and Non-blocking Sort-based Join Algorithm. In *VLDB 2002, Proc. of 28th International Conference on Very Large Data Bases, August 20-23, 2002, Hong Kong, China*, 2002.

[HFAE03] Moustafa A. Hammad, Michael J. Franklin, Walid G. Aref und Ahmed K. Elmagarmid. Scheduling for shared window joins over data streams. In *VLDB 2003: Proc. of the 29th international conference on Very Large Data Bases*. VLDB Endowment, 2003.

[HH99] Peter J. Haas und Joseph M. Hellerstein. Ripple Joins for Online Aggregation. In Alex Delis, Christos Faloutsos und Shahram Ghandeharizadeh, Hrsg., *SIGMOD 1999: Proc. ACM SIGMOD International Conference on Management of Data*. ACM Press, 1999.

[JG08] Jonas Jacobi und Marco Grawunder. ODYSSEUS: Ein flexibles Framework zum Erstellen anwendungsspezifischer Datenstrommanagementsysteme. In Hagen Höpfner und Friederike Klan, Hrsg., *Grundlagen von Datenbanken*, Jgg. 01/2008 of *Technical Report*. School of Information Technology, International University in Germany, 2008.

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

[LLD+07] Ming Li, Mo Liu, Luping Ding, Elke A. Rundensteiner und Murali Mani. Event Stream Processing with Out-of-Order Data Arrival. In *ICDCSW '07: Proc. of the 27th International Conference on Distributed Computing Systems Workshops*, Washington, DC, USA, 2007. IEEE Computer Society.

[LTS+08] Jin Li, Kristin Tufte, Vladislav Shkapenyuk, Vassilis Papadimos, Theodore Johnson und David Maier. Out-of-Order Processing: A New Architecture for High-Performance Stream Systems. In *VLDB 2008: Proc. of 34th International Conference on Very Large Data Bases*, 2008.

[SCLP08] Mohamed A. Sharaf, Panos K. Chrysanthis, Alexandros Labrinidis und Kirk Pruhs. Algorithms and metrics for processing multiple heterogeneous continuous queries. *ACM Trans. Database Syst.*, 33(1), 2008.

[SW04] Utkarsh Srivastava und Jennifer Widom. Flexible time management in data stream systems. In *PODS 2004: Proc. of the twenty-third ACM SIGMOD-SIGACT-SIGART symposiumon Principles of database systems*, New York, NY, USA, 2004. ACM Press.

[UF01] Tolga Urhan und Michael J. Franklin. Dynamic Pipeline Scheduling for Improving Interactive Query Performance. In Peter M. G. Apers, Paolo Atzeni, Stefano Ceri, Stefano Paraboschi, Kotagiri Ramamohanarao und Richard T. Snodgrass, Hrsg., *VLDB 2001: Proc. of 27th International Conference on Very Large Data Bases*. Morgan Kaufmann, 2001.