Seminar Datastreams
Thema: Operatoren auf Datastreams

Bearbeiter: Marc Fleming
Betreuer: Boris Stumm

Technische Universität Kaiserslautern

Lehrgebiet Datenverwaltungssysteme

Wintersemester 04/05

Inhaltsverzeichnis

1 Einleitung

2 Anfrageverarbeitungsarchitektur im Datenstrommodell

3 Zeitstempel

4 Windowed Join

4.1 Der Multi-Way-Join-Algorithmus

4.2 Problem bei Zeitstempeln im Zusammenhang mit dem Ergebnis von Joins

5 Block-Operatoren

5.1 Online-Aggregation

5.2 Der REORDER-Operator

5.3 Die Blockierungsproblematik

5.4 Punctuations

5.5 Anwendungsbeispiel

6 Kombination von Datenströmen mit statischen Relationen

6.1 Die Anfragesprache CQL

7 Fazit

1. Einleitung

Erst kürzlich ist eine neue Klasse von Anwendungen entstanden, die sehr datenintensiv sind. Dies sind z.B. Anwendungen aus dem Finanzbereich, dem Telekommunikationsbereich, aus dem Bereich der Sensornetzwerke usw. In solchen Anwendungen stehen die Daten, auf denen gearbeitet wird, nicht auf Platte oder im Hauptspeicher zur Verfügung, sondern kommen als Datenströme im System an. Damit entsteht ein neues Datenmodellierungskonzept, das Datenstrommodell.

Das Datenstrommodell hat folgende Eigenschaften [BBD+02]:

- Die Daten werden sofort bei ihrer Ankunft im System verarbeitet.

- Das System hat keine Kontrolle über die Reihenfolge, in der die Datenelemente ankommen.

- Datenströme haben potentiell unbegrenzte Länge.

- Sobald ein Datenelement verarbeitet worden ist, wird es entweder verworfen oder für eine begrenzte Zeit gespeichert.

Es wird im folgenden untersucht werden, mit welchen Operatoren man auf Datenströmen arbeiten kann. In Abschnitt 2 wird allgemein vorgestellt, wie eine Anfrageverarbeitungsstruktur auf Datenströmen aussehen kann. In Abschnitt 3 wird das Zeitstempelkonzept eingeführt und darauf aufbauend dann zeitbasierte Fenster definiert. In Abschnitt 4 wird ein Join-Algorithmus vorgestellt, der auf diesen Fenstern arbeitet, sowie die Problematik der Zeitstempelzuordnung für das Join-Ergebnis besprochen. In Abschnitt 5 werden Block-Operatoren besprochen. In Abschnitt 6 werden wir sehen, wie man Datenströme und statische Relationen kombinieren kann. Als Abschluss gibt es dann ein Fazit.

2. Anfrageverarbeitungsarchitektur im Datenstrommodell

Es wird jetzt eine mögliche Ausführungsarchitektur nach [BBD+02] für Anfragen auf Datenströmen dargestellt. Die Verarbeitungsarchitektur besteht aus Operatoren, die durch Warteschlangen miteinander verbunden werden. In diesen Warteschlangen werden die Datenstromelemente abgelegt und dann so schnell wie möglich von den Operatoren verarbeitet. In dieser Verarbeitungsarchitektur gibt es auch Strukturen, „synopsis“ genannt, in denen die aktuellen Arbeitsdaten abgelegt werden. Bei Join sind dies z.B. Fenster. Auf diese wird bei der Vorstellung von „Windowed Join“ genauer eingegangen.

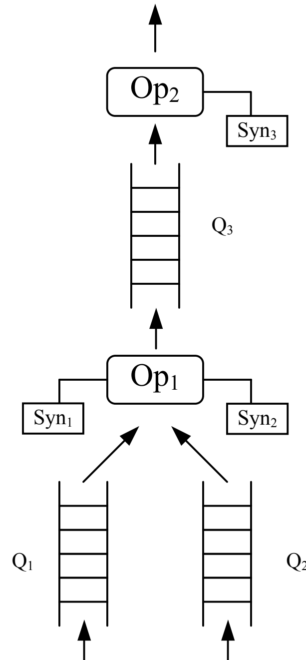

Abbildung 1: Anfrageplan

Ein Ausschnitt aus einem Anfragegraphen wird in Abbildung 1 gezeigt. Hier ist Op_1 ein Join-Operator, der zwei Fenster Syn_1 und Syn_2 verwaltet. Der Hauptspeicher wird dynamisch zwischen den Fenstern und den Warteschlangen sowie einem Cache für Daten die von Platte kommen aufgeteilt. Während der Verarbeitung der Anfrage liest ein Operator immer wieder ein Datenelement aus seiner Eingabewarteschlange, aktualisiert sein Fenster, falls er eines besitzt und schreibt das Resultat in seine Ausgabewarteschlange. Die Teilresultate einer Anfrage werden so im Anfragegraphen bis zum obersten Operator weitergereicht. Die Dauer des Ausführungszyklus für einen Operator wird durch eine zentrale Schedulereinheit festgelegt. Dabei kann die Dauer des Ausführungszyklus vom Scheduler entweder durch eine feste Zeit festgelegt werden oder durch eine feste Anzahl an Datentupel, die der Operator konsumieren oder ausgeben darf, gegeben sein.

3. Zeitstempel

Da, wie bereits in der Einleitung angesprochen, Datenströme eine potentiell unbegrenzte Länge haben, kann im Allgemeinen nicht der gesamte Datenbestand betrachtet werden, um eine Anfrage auf Datenströmen zu beantworten. Also geht man dazu über, nur noch einen bestimmten Anteil des Gesamtbestandes der Daten zu betrachten. Dieser Anteil wird durch die Menge der Daten, die das System während eines Zeitintervalls der Grösse T erreicht, festgelegt und wird als Fenster bezeichnet. Jedes Datentupel ist Träger eines Zeitstempels, und die Definition eines zeitbasierten Fensters muss unter Berücksichtigung dieses Zeitstempels erfolgen.

Im folgenden werden zwei Methoden besprochen, um Zeitstempel zu vergeben [BBD+02]:

implizites Timestamping: Das System fügt jedem Tupel aus dem Datenstrom ein spezielles Zeitfeld hinzu. Das Zeitfeld repräsentiert den Ankunftszeitpunkt des Datentupels im System.

explizites Timestamping: Die Datenelemente aus dem Datenstrom werden als Datentupel betrachtet. Beim expliziten Timestamping wird ein Datenattribut des ankommmenden Datentupels als Zeitstempel gewählt. Explizites Timestamping kann genutzt werden, wenn die Bedeutung eines Datentupels an ein Ereignis in der realen Welt geknüpft ist. Zum Beispiel ein Temperatursensor in einem Gewächshaus, der sowohl die gemessene Temperatur als auch die Zeit, zu der diese Temperatur gemessen wurde, an ein System schickt. Hier wird das Attribut Zeit als Zeitstempel gewählt. Dieser Zeitstempel wird nicht vom System erzeugt und zur Temperatur hinzugefügt (dies wäre implizites Timestamping), sondern vom Temperatursensor als Teil des Datentupels an das System geliefert.

Implizites Timestamping kann genutzt werden, wenn die von der Datenquelle gelieferten Daten selbst keine Zeitinformation enthalten oder wenn der genaue Zeitpunkt, an dem ein bestimmtes Datenelement entstanden ist, keine Bedeutung hat und man nur wissen will, ob es sich um ein älteres oder jüngeres Datenelement handelt.

Problem bei expliziten Zeitstempeln [BBD+02]:

Explizite Zeitstempel haben den Nachteil, dass Datentupel möglicherweise nicht in der gleichen Reihenfolge wie ihre Zeitstempel im System ankommen; d.h. dass Datentupel mit späterem Zeitstempel im System eher ankommen können als Datentupel mit früherem Zeitstempel.

Dieses Problem kann man jedoch, falls es sich nur um lokale Permutationen handelt, durch Zwischenpufferung lösen. Wenn bekannt ist, dass keine neuen Datentupel mit bestimmten Werten mehr kommen, kann dies markiert werden, und die Datentupel können zur Verarbeitung freigegeben werden. Markierungen werden in Abschnitt 5.4 vorgestellt.

4. Windowed Join

Betrachtet wird der Fall, dass man Daten aus verschiedenen Datenströmen erhält und dass man ein Join auf einem gemeinsamen Attribut ausführen möchte. Dies kann z.B. notwendig werden, wenn man Luftdruck- und Temperaturwerte haben möchte, die von unterschiedlichen Wetterstationen zur selben Zeit gemessen wurden, um Wettervorhersagen zu erstellen.

4.1 Der Multi-Way-Join-Algorithmus

Gegeben seien n verschiedene Datenströme und jedem Datenstrom sei ein Fenster zugeordnet. Die Datenelemente aus dem Datenstrom sind Tupel, bestehend aus einem Zeitstempel ts und attr, dem im System angekommenen Datenattribut. Die Fenster, welche den Datenströmen zugeordnet sind, passen vollständig in den Hauptspeicher. Somit besteht nicht die Gefahr eines Speicherüberlaufs. Hat ein Fenster die Grösse T, so enthält dieses Fenster bei Ankunft eines neuen Datenattributs mit Zeitstempel ts die Datenattribute aus dem Zeitintervall [ts-T, ts].

Zunächst wird eine erste Variante des Algorithmus [GoTa03] vorgestellt:

Wir zeigen die Vorgehensweise des Algorithmus im Fall, dass wir n = 3 Datenströme haben, mit den jeweils zugeordneten Fenstern, die mit S1, S2 und S3 bezeichnet werden. Sei der Anfragegraph . $S_1 \bowtie (S_2 \bowtie S_3)$

Der „Naive“ Multi-Way-Join-Algorithmus:

Bei Ankunft eines neuen S1-Datentupels werden verfallene S2 und S3-Tupel invalidiert, ein Join zwischen dem S2 und S3-Fenster wird ausgeführt, und anschliessend wird das Join-Prädikat für das neue S1-Tupel mit dieser Ergebnismenge geprüft. Kommt ein neues S2-Tupel im System an, so werden zunächst alle verfallenen S1 und S3-Tupel verworfen, anschließend wird für jedes Tupel in S1 der Join zwischen dem neuen S2-Tupel und dem S3-Fenster gebildet und die Ergebnismenge geprüft. Für neue S3-Tupel werden analog alle verfallenen S1 und S2-Tupel invalidiert, und für jedes S1-Tupel wird der Join zwischen dem neuen S3-Tupel und dem S2-Fenster gebildet und die Ergebnismenge geprüft. Es wird also immer erst $S_2 \bowtie S_3$ berechnet.

Bei dieser Variante des Algorithmus wird bei der Ankunft eines neuen S3-Datentupels für jedes Tupel in S1 der Join zwischen dem neuen S3-Tupel und dem S2-Fenster neu berechnet. Dies ergibt unötige Arbeit.

Es wird deshalb eine zweite Variante des Algorithmus vorgeschlagen, um neue S3-Tupel zu behandeln.

Der Eager Multi-Way-Join-Algorithmus (verbesserte Variante):

Der Algorithmus wählt zunächst nur die Tupel aus dem S1-Fenster, für die das Join-Prädikat mit dem neuen S3-Tupel erfüllt ist. Dies sei z.B. eine feste Anzahl von c Tupeln. Anschließend werden c Durchläufe von dem S2-Fenster durchgeführt. Bei der ersten Variante des Algorithmus ist die Anzahl der Durchläufe von dem S2-Fenster identisch mit der Größe des S1-Fensters. Ist nicht für alle Tupel in S1 das Join-Prädikat mit dem neuen S3-Tupel erfüllt gewesen, so ist c kleiner als die Anzahl der Tupel im S1-Fenster und man hat eine Verbesserung erzielt.

Wenn ein neues S1 bzw S2-Tupel im System ankommt, geht der Algorithmus ähnlich vor. Er wählt das Tupel aus dem Fenster, das an der Spitze der Join-Folge steht, prüft das Join-Prädikat mit dem in der Reihenfolge nachfolgenden Fenster, und für jeden Match führt er ein Durchlauf des noch bleibenden Fensters aus. Das Fenster an der Spitze der Join-Folge besteht dabei nur aus

einem Datentupel und ändert sich in Abhängigkeit davon, aus welchem Fenster das neue Tupel kommt. Es ergeben sich folgende Möglichkeiten wie in Tabelle 1 gezeigt:

Herkunft des neuen Datentupels	Join-Reihenfolge
S_1	$S_1 \bowtie (S_2 \bowtie S_3)$
S_2	$S_2 \bowtie (S_1 \bowtie S_3)$
S_3	$S_3 \bowtie (S_1 \bowtie S_2)$

Tabelle 1: Join-Reihenfolge

Die Join-Reihenfolge zu ändern ist nur möglich, wenn man ein gemeinsames Join-Attribut über allen Datenströmen voraussetzt. Die eben geschilderte Variante des Eager Multi-Way-Join-Algorithmus ist bereits eine Verbesserung des allgemeinen Eager Multi-Way-Join-Algorithmus. Der Unterschied zwischen dem allgemeinen Eager Multi-Way-Join-Algorithmus und seiner verbesserten Variante besteht nur darin, dass dieser immer nach der allgemeinen Join-Reihenfolge vorgeht. Die allgemeine Join-Reihenfolge wird als die Reihenfolge definiert, die der Algorithmus benutzen wird, wenn die neuen Tupel nicht an der Spitze der Reihenfolge stehen. Im obigen Beispiel mit den drei Fenstern ist dies die Reihenfolge S1, S2, S3. Der allgemeine Eager Multi-Way-Join-Algorithmus würde also bei Ankunft eines neuen S3-Tupels zuerst das Join-Prädikat zwischen S1 und S2 prüfen und für jeden Match dann das Join-Prädikat zwischen S2 und S3 für das neue S3-Tupel prüfen. Analog für neue S1 und S2-Tupel.

In den vorangehenden Erläuterungen wird davon ausgegangen, dass ein Datenelement sofort vom Join-Operator verarbeitet wird, wenn es im System ankommt. Der Algorithmus prüft bei Neuankunft eines Datenelements mit Zeitstempel ts aus einem Datenstrom das Join-Prädikat mit den anderen Fenstern für Datenelemente aus dem Zeitintervall [ts-T, ts]. Da man die Zeitbeschränkung T hat, genügt es auch, wenn alle T Zeiteinheiten eine Neuauswertung des Join-Prädikats vorgenommen wird. Dies ist eine Optimierung des allgemeinen Eager Multi-Way-Join-Algorithmus.

4.2 Problem bei Zeitstempeln im Zusammenhang mit dem Ergebnis von Joins

Wenn die Daten aus einem einzigen Datenstrom kommen ist das implizite Timestamping unproblematisch. Was ist jedoch, wenn die Datentupel wie z.B. beim Multi-Way-Join aus verschiedenen Datenströmen kommen, wobei für die Datentupel ein Join auf einem gemeinsamen Attribut ausgeführt werden soll, und diese Datentupel unterschiedliche Zeitstempel besitzen?

Dann stellt sich die Frage, welchen Zeitstempel das Ergebnis von Join besitzt [BBD+02].

Für dieses Problem gibt es mehrere Lösungen, wovon zwei besprochen werden.

Eine Lösungsmöglichkeit ist, keine besondere Reihenfolge über den Ausgabetupeln des Join-Operators festzulegen, sondern einfach dafür zu sorgen, dass Datentupel, die eher im System ankommen, auch eher vom Join-Operator bearbeitet werden. Diese Lösungsmöglichkeit hat allerdings den Nachteil, dass man keine zeitbasierten Fenster über den Resultaten des Join- Operators definieren kann, da die Resultate keinen Zeitstempel mehr haben.

Der zweite Lösungsansatz kann sowohl für implizites als auch für explizites Timestamping verwendet werden. Hier muss der Benutzer als Teil der Anfrage spezifizieren, wie der Zeitstempel für Resultate des Join-Operators vergeben werden soll. Eine Möglichkeit ist, dass die Reihenfolge, in der die Streams in der FROM Klausel der Anfrage aufgelistet sind, den Zeitstempel bestimmt.

Beispiel 1:

```
Select *
From  S1[ROWS 1000 PRECEDING],
      S2[ROWS 100 PRECEDING]
Where S1.A = S2.B
```

Dabei ist `S1[ROWS 1000 PRECEDING]` ein Fenster der letzten 1000 Datentupel, wobei diese Datentupel streng nach Zeitstempel geordnet sind. Bei dieser Anfrage bekommt das Resultat des Join-Operators den Zeitstempel des Datentupels aus S1.

Dieser Lösungsansatz führt aber dazu, dass viele Resultate des Join-Operators, den gleichen Zeitstempel bekommen können.

Will man die Resultate nach Zeitstempel ordnen, wird dieses Problem gelöst, indem man an die Join-Resultate, die den gleichen Zeitstempel wie ein anderes Join-Resultat bekommen würden, den Zeitstempel des Datentupels aus S2 vergibt. Die Ausgabetupel werden dann zunächst nach dem Zeitstempel aus dem Datenstrom S1 sortiert. Gibt es Tupel, die den Zeitstempel aus S2 erhalten haben, so werden diese Tupel nach dem Zeitstempel aus S2 sortiert.

5. Block-Operatoren

Block-Operatoren haben die Eigenschaft, dass sie auf Datenströmen blockieren können, da der Datenstrom möglicherweise nicht abreißt, und dann diese Operatoren keine Ausgabe produzieren. Bekannte Block-Operatoren sind SORT, sowie sämtliche Aggregationsoperatoren wie AVG oder GROUP BY. Die Blockade kann gelöst werden, indem sogenannte Markierungen (punctuations) eingesetzt werden, die im Abschnitt 5.4 vorgestellt werden. Es gibt aber auch Ansätze, wie man schnell zu Resultaten einer Anfrage (wenn auch nur Schätzungen) kommen kann, wenn man auf großen Datenmengen arbeitet. Wenn eine Anfrage, wie die aus Abschnitt 5.1 im Falle von Online-Aggregation, auf großen Datenmengen arbeitet, sieht es so aus, dass die Berechnung der Anfrage entsprechend lange dauert. Jedoch blockiert sie nicht, da im Anwendungsbeispiel auf endlichen Datenmengen gearbeitet wird. Man kann dies dann aber auch als eine Art Blockade bezeichnen, da der Benutzer eines Systems schnell Resultate erhalten will. Hier gibt es einen RE-ORDER-Operator durch den man geschätzte Resultate erhalten kann. Dieser Operator kann so-

wohl für große relationale Datenmengen als auch auf Datenströmen eingesetzt werden. Der REORDER-Operator wird in Abschnitt 5.2 vorgestellt. Der folgende Abschnitt illustriert die Online-Aggregation als Anwendungsfall, bei dem man den REORDER-Operator einsetzen kann und geht genauer auf die Benutzung des REORDER-Operators ein.

5.1 Online-Aggregation

Ein Ziel von Aggregation auf Datenströmen ist, entscheidungsunterstützende Anfrageverarbeitung interaktiv zu gestalten, indem Schätzungen des definitiven SQL-Anfrageresultats schon zur Verfügung gestellt werden während die Verarbeitung der Anfrage noch läuft.

Stellen wir uns eine Person vor, die, um eine Entscheidung zu treffen, Daten über die Durschnittseinnahmen eines Unternehmens in den verschiedenen Ländern analysieren will.

Die SQL-Anfrage könnte dann so aussehen:

```
Select avg(revenue), nation
From sales, branches
Where sales.id = branches.id
Group by nation
```

Wenn dieses System auf Datenströmen arbeitet, kann der Benutzer während der Verarbeitung der Anfrage schon Schätzungen des definitiven Ergebnisses erhalten. Das Ergebnis wird immer weiter verfeinert. Der Benutzer kann beispielsweise sofort aufgrund der Schätzungen erkennen, dass die Einnahmen in China höher sind als die in Japan.

Der Benutzer soll die Möglichkeit haben anzugeben, welche Daten ihn bei der Auswertung der Anfrage näher interessieren und welche nicht. Dies geschieht durch Gruppierung der Daten und Vergabe einer Präferenz des Benutzers für die Daten einer bestimmten Gruppe. Die Gruppen, die bei dieser Anwendung betrachtet werden, entstehen, durch Zuordnung der ankommenden Datenelemente zu den verschiedenen Nationen gemäß GROUP BY Klausel der Anfrage. Da die Resultate der Online-Aggregation für jede Gruppe nur Schätzungen sind, ist jeder Gruppe, zusätzlich zur Präferenz für diese Gruppe von Datenelementen, ein Konfidenzintervall zuzuordnen. Ziel der Online-Aggregation ist es, dieses Konfidenzintervall, für Gruppen, die vom Benutzer als interessant eingestuft wurden, so schnell wie möglich klein zu machen. Die Angabe einer Präferenz durch den Benutzer führt dazu, dass für interessante Gruppen genauere Resultate berechnet werden, als für andere. Somit kann ein Resultat für uninteressantere Gruppen schneller ermittelt werden. Der Benutzer soll außerdem die Präferenz der Datengruppen während dem Verlauf der Berechnung verändern können. Dies bietet dem Benutzer eine gewisse Interaktivität mit dem System.

Eine ständige Neuordnung der Datenelemente sorgt dafür, dass stets der korrekte Anteil an Datenelementen, für jede Gruppe im Hauptspeicher vorhanden bleibt, basierend auf den Präferenzen des Benutzers. Diese Neuordnung der Datenelemente führt ein REORDER-Operator durch. Er gibt die Datenelemente zur Verarbeitung an die Anwendung weiter. Der Benutzer hat außerdem für dieses Anwendungsbeispiel der Online-Aggregation mit Hilfe des REORDER-Operators die

Möglichkeit, die Konfidenz der Berechnung über alle Gruppen hinweg einigermaßen konstant zu halten, und so die Berechnung der Anfrage fair ablaufen zu lassen. Unter Konfidenz versteht man die Wahrscheinlichkeit (1-a), dass das Endresultat der Berechnung für eine Gruppe von Datenelementen innerhalb eines bestimmten Intervalls, um den Erwartungswert (das Konfidenzintervall) liegt. a ist die Irrtumswahrscheinlichkeit. Diese Art der Berechnung einer Anfrage auf großen Datenmengen kann immer dann vorgenommen werden, wenn genaue Resultate nicht benötigt werden, und der Benutzer auch die Absicht hat, die Berechnung noch während ihrem Verlauf abzubrechen, wenn sie ihm zufriedenstellende Ergebnisse geliefert hat. Es soll nun dieser REORDER-Operator vorgestellt werden, der eine Neuordnung der Datenelemente vornimmt. Eine weitere gute Eigenschaft des REORDER-Operators ist, dass er auf Datenströmen nicht blockiert.

5.2 Der REORDER-Operator

Der REORDER-Operator geht davon aus, dass die Daten auf denen gearbeitet wird, gruppiert werden, z.B. nach GROUP BY Klausel. In der Anwendung in Abschnitt 5.1 kann dann der REORDER-Operator dazu dienen eine Performanzsteigerung zu erreichen, die so aussieht, dass die Resultate für die Datengruppen, die den Benutzer weniger interessieren, schneller berechnet werden und somit ungenauer sind.

Der Aufbau des Systems für den REORDER-Operator ist in Abbildung 2 dargestellt:

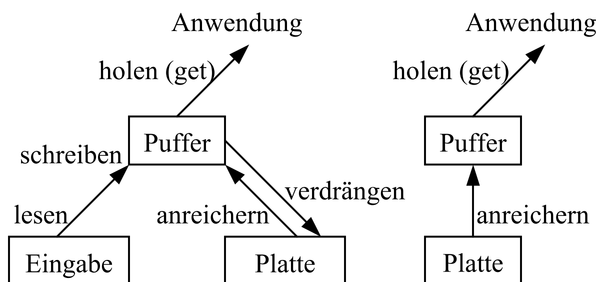

Abbildung 2: Reordering

Das System versucht, soviele interessante Datentupel wie möglich in einem Puffer im Hauptspeicher abzulegen. Der Algorithmus nutzt die Zeit zwischen den get-Anfragen der Anwendung, um den korrekten Anteil an Daten für die verschiedenen Datengruppen, im Hauptspeicher zu halten, basierend auf den Präferenzvorgaben des Benutzers. Falls der Benutzer seine Präferenzvorgaben bezüglich der Gruppe an Daten, die ihn interessiert verändert, könnten Datenelemente, die vorher interressant waren, plötzlich uninteressant werden, also muss dafür gesorgt werden, dass der richtige Anteil an interessanten Daten für jede Gruppe im Hauptspeicher vorhanden bleibt, indem uninteressante Daten auf Platte verdrängt und der Hauptspeicher mit den jetzt interessanten Daten von der Platte angereichert wird.

Eine Schwierigkeit bei diesem Operator liegt darin, eine Möglichkeit zu finden, um *gute* Datenelemente (d.h. Datenelemente, die den Interessen des Benutzers möglichst gut entsprechen) aus

dem Hauptspeicher zu wählen und an die Anwendung zu liefern. Dies wird an folgendem Beispiel erklärt:

Gegeben sei ein Datenstrom $t_1 \ldots t_n$. Man will eine gute Permutation $t_{\pi 1} \ldots t_{\pi n}$ finden. D.h. hat der Benutzer die Datenelemente in zwei Gruppen A und B unterteilt und seine Präferenzfunktion so definiert, dass ihm die Gruppe A doppelt so wichtig ist wie die Gruppe B, dann sind beispielsweise die Permutationen A A B A A B A A ... und B A A B A A B ... gleich gut.

Es gibt mehrere Möglichkeiten, um Datenelemente aus dem Hauptspeicher zu wählen, und an die Anwendung zu liefern. Wir werden zwei davon vorstellen. Dazu muss die Benutzerpräferenz, welche der Benutzer für jede Gruppe von Daten angibt, und die sein Interesse an der Datengruppe wiederspiegelt, auf eine Priorität für die Datenelemente dieser Gruppe abgebildet werden. Um dies zu tun definiert man eine anwendungsspezifische „quality-feedback“ Funktion $F(UP(t_{\pi 1}), \ldots ,UP(t_{\pi n}))$ für eine Permutation $t_{\pi 1} \ldots t_{\pi n}$. Dabei ist $UP(t_i)$ die Präferenz des Benutzers für das Datenelement t_i aus einer bestimmten Gruppe von Datenelementen. F modelliert das Interesse des Benutzers an der bisherigen Permutation von Datenelementen, die der REORDER-Operator zur Verarbeitung an die Anwendung geliefert hat. Es wird stets ein Element aus einer Gruppe zur Verarbeitung ausgewählt, die die größte Steigung von F hervorruft. Wir werden später noch beispielhaft sehen, wie solch eine Funktion F aussehen kann.

Die erste Möglichkeit, die wir vorstellen, um neue Datenelemente aus dem Hauptspeicher zu wählen, kann genutzt werden bei einem System, das gemessene Werte von mehreren Sensoren als Eingabe hat, die in Datenströmen ankommen. Wenn sich der Benutzer die Sensorwerte anschaut, und feststellt, dass ein Sensor ungewöhnliche Werte geliefert hat, will er diese genauer analysieren. Jeder Sensor bekommt wiederum eine Gruppe zugeordnet. Die anormalen Werte von diesem Sensor, sollen hier so an die Anwendung geliefert werden, dass die Anzahl der Datentupel, die von diesem Sensor an die Anwendung geliefert wird, proportional zur Präferenz ist, die der Benutzer, für diese Datengruppe anzugeben hat [RaRH99].

Die zweite Möglichkeit die vorgestellt wird, besteht darin, dass der REORDER-Operator zunächst alle Datenelemente aus der Gruppe mit der höchsten Benutzerpräferenz liefert, dann alle Datenelemente aus der Gruppe mit der zweithöchsten Benutzerpräferenz usw. Die „quality-feedback“ Funktion F sieht in diesem Fall so aus: $F = \Sigma\ UP_i \quad i = 1, \ldots ,n$. Dabei steht Up_i für die Benutzerpräferenz die der Benutzer einer Gruppe i von Datenelementen zugeordnet hat, und die sein Interesse an dieser Datengruppe wiedergibt. n ist die Anzahl an Datenelementen, die bis zum aktuellen Zeitpunkt verarbeitet wurden. Hier wird die Priorität, mit der ein Datenelement t_i an die Anwendung gegeben wird mit $Dp_i = Up_i$ angegeben. Die stärkste Steigung von F erhält der REORDER-Operator in diesem Fall, wenn er ein Datenelement aus der Gruppe mit der höchsten Benutzerpräferenz wählt.

Nachdem wir jetzt den REORDER-Operator vorgestellt haben, der die Blockierungsproblematik von Blockoperatoren noch nicht löst, da er nicht helfen kann, wenn man genaue Resultate berechnen möchte, ist jetzt das Ziel die Blockierungsproblematik zu lösen. Damit lassen sich dann auch genaue Resultate über Daten aus Strömen berechnen.

5.3 Die Blockierungsproblematik

Wir wissen, dass man Block-Operatoren wegen ihrer Blockierungsproblematik nicht ohne Anpassung auf Datenströme übernehmen kann. Ein Operator, der auf Datenströmen arbeitet, sollte nämlich sofort nach Ankunft der ersten Datenelemente auch die ersten Ergebnisse liefern können. Dies ist bei Blockoperatoren nicht der Fall. Ein Operator der auf Datenströmen arbeitet, muss als Stream-Iterator geschrieben werden können. Wir werden den Begriff Stream-Iterator definieren. Er hat die Eigenschaft, dass er seine Eingabe nie vollständig sieht.

Definition eines Stream-Iterators: Ein Datenstrom wird definiert als eine potentiell unbegrenzte Sequenz von Elementen eines Typs T. Ein Strom wird mit {| ... |} geschrieben, um ihn von endlichen Listen, die mit [...] geschrieben werden, zu unterscheiden. Dann kann ein Datenstrom dargestellt werden als unbegrenzte Sequenz von Teillisten von Datenelementen eines Typs T [TMS+02].

Beispiel 2:

{|1, 2, 3, 4, 5, ... |} kann dargestellt werden als { | [1, 2, 3], [4, 5], ... | }

Es können auch endliche Datenströme modelliert werden, indem man eine Sequenz von nachfolgenden leeren Listen benutzt: { | [1, 2] , [3], [4] , [ ], [ ], ... |}

Dieser Ansatz zur Modellierung von Datenströmen steht nicht im Widerspruch dazu, dass Datenelemente aus den Datenströmen mit unterschiedlichen Geschwindigkeiten im System ankommen können und dass es längere Zeiten geben kann, in denen kein Datenelement aus einem Datenstrom kommt [TMS+02].

Wenn S der Strom { |1, 3, 5, 7, ... | } ist, notiert man mit S[3] die Teilliste der ersten 3 Elemente [1, 3, 5]

Nun kann der Begriff Stream-Iterator definiert werden:

Eine Funktion f: Stream (T) => Stream (U) heißt Stream-Iterator, wenn es eine Funktion q: List(T) => List(U) gibt, so dass für jedes S aus Stream(T) $f(S) = (q(S[1]) \otimes q(S[2]) \otimes q(S[3]) \otimes \ldots \otimes q(S[i]) \otimes \ldots)$ gilt. Dabei wird $\otimes$ benutzt, um einen Datenstrom aus einer endlichen Liste und einem anderen Datenstrom zu konstruieren. f ist also ein Stream-Iterator, wenn f durch wiederholte Anwendung von q definiert werden kann.

Beispiel 3:

Die Operation **select** kann als Stream-Iterator beschrieben werden. Sei p ein **select** Prädikat. Dann wird q definiert als:

q (L ++ [a]) = [a] falls p[a] , q (L ++ [a]) = [ ] sonst (++ steht für die Verkettung von Listen.)

Die Operation **select** kann deshalb als Stream-Iterator beschrieben werden, da diese Operation immer nur das letzte Element seiner Eingabe braucht. Hierauf wird das Prädikat p ausgewertet. Abhängig vom Zutreffen von Prädikat p wird das Datenelement an den Ausgabestrom angehangen oder nicht.

Ein unärer Stream-Iterator (binärer Stream-Iterator siehe [TMS+02]) lässt sich auch als Tripel (*initial state, step, final*) beschreiben.

- *initial state* ist der initiale Zustand, bevor Daten aus dem Datenstrom im System ankommen.

- *step* wird aufgerufen, wenn ein neues Datentupel im System ankommt, das verarbeitet werden muss. Step liefert auch den neuen Zustand des Systems.

- *final* wird aufgerufen, falls der Datenstrom endet und liefert die letzte Ausgabe sowie den Endzustand.

Operatoren wie SORT oder GROUP BY können jedoch nicht mit dieser Definition als Stream-Iterator geschrieben werden. Sie gehören zu den blockierenden Operatoren, und müssen somit ihre gesamte Eingabe sehen, die, wie bei select, aus einem Strom von Listen (der potentiell nicht abbricht) bestehen kann, um ihre Ausgabe zu produzieren. Im Gegensatz hierzu muss select nur das letzte Element seiner Eingabe sehen.

Wenn man den Stream-Iterator als generelle Definition für eine stream-to-stream Funktion hat, stellt sich die Frage ob sich der Mangel, den Stream-Iteratoren nach der bisherigen Definition bezüglich der Block-Operatoren haben, beheben lässt. Dies führt zu Markierungen.

Markierungen (punctuations) bieten eine Möglichkeit, um sowohl endliche als auch unendliche Ströme zu repräsentieren. Somit kann das Ergebnis erzeugt werden, sobald das Ende eines Datenstroms erkannt wird.

5.4 Punctuations

Markierungen können als Prädikate auf Datenstromelementen verstanden werden, die, nachdem die Markierung vorbei ist, in diesem Strom nicht mehr auftauchen.

Eine Möglichkeit, einen endlichen Datenstrom zu repräsentieren, besteht z.B. darin, dass man ein spezielles Schlüsselwort EOS einführt, um zu markieren, dass keine weiteren zulässigen Datenelemente mehr kommen.
Betrachten wir folgenden endlichen Datenstrom { | 8, 2, 6,11, 3, EOS, EOS, ... | }.

Wenn man den SORT-Operator definieren möchte, so kann die Funktion q bei Eingabe [8, 2, 6, 11, 3, EOS], [2, 3, 6, 8, 11] ausgeben, und [EOS] bei längeren Eingaben. Das Schlüsselwort EOS kann sogar noch verstärkt werden.

Nehmen wir an, dass wir an einem bestimmten Punkt eines Datenstromes wissen, dass keine ganzen Zahlen aus dem Bereich 1-10 mehr kommen werden. Dies kann durch EOS(1-10) markiert werden und eine Funktion q kann dann bereits partielle Resultate für einen Datenstrom ermitteln. Weitere Beispiele für Markierungen sind z.B. c oder [c1,c2] oder {c1, c2, ... }: Die Match-Bedingungen für diese Markierungen sind i == c, c1 ≤ i ≤ c2 und i∈{c1, c2, ... } für ein Datenelement i. Nachdem diese Markierungen vom System empfangen wurden, werden Datenelemente, die solche Markierungen erfüllen, verarbeitet, und werden anschließend nicht mehr im Folgestrom zu finden sein.

Nun benötigt man noch Operatoren für den Umgang mit Markierungen auf Datenströmen.

Die zwei wichtigsten sind:

1. *match*: Die match-Funktion nimmt eine Markierung und ein Datenelement t als Eingabe, prüft ob das Datenelement die Markierung erfüllt und gibt TRUE zurück falls dies der Fall ist. Match wird genutzt, um aufgrund von Markierungen zu entscheiden, ob Datenelemente zur Verarbeitung an blockierende Operatoren weitergereicht werden oder nicht.

2. *combine*: nimmt zwei Markierungen p1 und p2 als Eingabe und erzeugt eine neue Markierung, die eine Überschneidung der beiden anderen Markierungen ist.

Beispiel: match (t, combine (p1, p2)) => match (t, p1) ∧ match (t, p2)

Wie bereits bekannt ist, gibt es die Definition eines unären Stream-Iterators als Tripel (*initial state, step, final*). Wenn man erreichen möchte, dass Stream-Iteratoren sich auf Datenströme mit Markierungen anwenden lassen, weil diese bei Block-Operatoren verwendet werden müssen, muss man dieses Tripel um Funktionen für den Umgang mit Markierungen erweitern.

Der Stream-Iterator kann dann folgendermaßen beschrieben werden: (*initial state, step, pass, prop, purge*).

Initial state und step erfüllen die gleiche Funktionalität wie vorher. Zu erläutern bleibt pass, prop und purge.

- *pass*: Nimmt neue Markierungen und den aktuellen Zustand des Systems als Eingabe, um neue Tupel zu bestimmen, die aufgrund der Markierung ausgegeben werden können.

- *prop*: Nimmt neue Markierungen und den aktuellen Zustand des Systems, um die Markierungen zu bestimmen, für die aufgrund des Systemzustands neue Tupel ausgegeben werden können.

- *purge:* Nimmt neue Markierungen, und den Zustand des Systems und bestimmt den neuen Zustand des Systems aufgrund von Markierungen. Dies entspricht der Säuberung des Zustandes.

Nun lassen sich auch Block-Operatoren als Stream-Iteratoren formulieren [TMS+02].

5. 5 Anwendungsbeispiel

Betrachtet wird ein Report-System für Stromausfälle. Subsysteme melden die Probleme an ein Hauptsystem. Zusätzlich können die Kunden des Stromunternehmens Stromausfälle an eine Telefondatenbank melden [TMS+02].

Mann kann dieses Report-System mit Hilfe einer Anfrage implementieren, die auf dem Kundendatenstrom alle geschilderten Probleme findet, die nicht in den Meldungen der Subsysteme enthalten sind. Jede Meldung der Subsysteme dient als Eingabe für einen UNION- Operator. Die Ausgabe dieses UNION-Operators wird als negative Eingabe an einen Differenzoperator gesendet, wobei die Meldungen der Kunden die positive Eingabe ist.

Diese Anfrage sieht wie in Abbildung 3 aus:

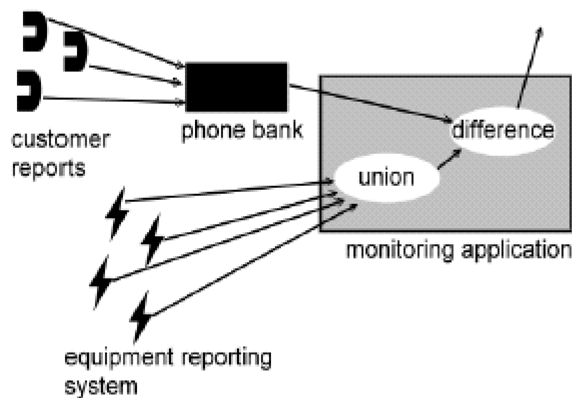

Abbildung 3: Report-System für Stromausfälle

Das System wird aus zwei Gründen versagen:

1. Der Differenzoperator muss warten, bis die gesamte negative Eingabe gelesen wurde, bevor er eine Ausgabe machen kann.

2. Der UNION-Operator entfernt Duplikate und hat eine über die Zeit unbeschränkt wachsende Zahl an Datenelementen zu verwalten.

Diese Probleme kann man durch Einfügen von Markierungen in den Datenstrom lösen. Unser Datenstrom besteht zusätzlich zu den Markierungen aus den Datentupeln < zoneid, hour >. Eine Markierung, die alle Meldungen für Stunde 3 beschreibt ist dann z.B. < *, 3 >. Eine Markierung, die alle Meldungen aus den Zonen 1 und 6 für die Stunden zwischen 5 und 8 beschreibt ist gegeben durch < {1, 6}, [5, 8] >.

Wenn man nun für das Report-System Markierungen einführt, lassen sich die beiden oben genannten Probleme beheben. Die Tupel der Eingabeströme haben die Form <zoneid,hour>, und sowohl die Telefondatenbank als auch die Informationen der Subsysteme enthalten Markierungen, die eine bestimmte Uhrzeit betreffen. Erhält der UNION-Operator eine Markierung, die eine bestimmte Uhrzeit betrifft, so kann er die Datentupel, die diese Markierung erfüllen, an den Differenzoperator als negative Eingabe weiterleiten und anschliessend diese Tupel aus seinem aktuellen Zustand entfernen. Wenn nun der Differenzoperator auf seinem negativen Eingabestrom (also die Ausgabe des UNION-Operators) Markierungen besitzt, kann der Differenzoperator Tupel der positven Eingabe ausgeben, die diese Markierungen erfüllen, und nicht in der negativen Eingabe enthalten sind. Damit sind die oben genannten Probleme gelöst.

6. Kombination von Datenströmen mit statischen Relationen

Wenn man Datenströme mit statischen Relationen kombinieren möchte, wird man zuerst versuchen Verknüpfungsoperatoren auf konzeptioneller Ebene zu definieren. Das Ziel wird sein, eine konkrete Anfragesprache zu definieren, welche sowohl auf Datenströmen, als auch auf Relationen arbeitet. Man merkt dann schnell, dass es zu Problemen kommt, die es zu lösen gilt: Da auf Relationen und Datenströmen gearbeitet wird, stellt sich die Frage, ob die Ausgabe einer Anfrage ein Datenstrom oder eine Relation sein wird. Diese Frage wird auch zunächst auf konzeptionellem Niveau zu klären sein. Wenn alle Konzepte auf abstraktem Niveau klar sind, kann dazu übergegangen werden, einige konkrete Operatoren in CQL vorzustellen und die besprochenen Konzepte umzusetzen. Wir werden dabei sehen, dass es Operatoren gibt, die sich aus SQL übernehmen lassen, dass es aber auch notwendig ist, neue Operatoren einzuführen.

Man hat Ströme und Relationen und unterscheidet drei Arten von Operatorklassen, um diese miteinander zu verknüpfen [ArBW03] .

- stream-to-relation-Operatoren
- relation-to-relation-Operatoren
- relation-to-stream-Operatoren

Stream-to-stream-Operatoren sind in dieser Aufzählung nicht vorhanden. Sie lassen sich aus den drei Basisklassen zusammensetzen.

Wir geben eine Definition eines Datenstroms und einer Relation und werden dann diese Operatoren genauer betrachten. Gegeben sei ein Zeitbereich T mit Werten τ aus den natürlichen Zahlen.

Wir definieren einen Datenstrom als eine möglicherweise unendliche Multimenge von Elementen $< s, \tau >$, wobei s ein Tupel aus einem Relationenschema ist.

Eine Relation R ist eine Abbildung vom Zeitbereich T auf eine Multimenge von Tupeln, die zum Relationenschema von R gehören.

Zunächst noch einige Schreibweisen, die im folgenden benutzt werden:

S bis zum Zeitpunkt τ ist die Menge der Datentupel aus dem Strom S mit Zeitstempel $\leq \tau$. Schreibweise: $\{<s, \tau`> \in S: \tau` \leq \tau\}$. R bis zum Zeitpunkt τ ist die Sammlung aller Relationen R(0), … ,R(τ). Mit R(τ) wird die Menge aller Tupel bezeichnet, die zum Zeitstempel τ in einer Relation R sind.

Ein stream-to-relation-Operator nimmt einen Datenstrom S als Eingabe und erzeugt eine Relation mit dem gleichen Schema wie S. Zu einem beliebigen Zeitpunkt τ muss R(τ) aus S bis zum Zeitpunkt τ berechenbar sein.

Ein relation-to-relation-Operator nimmt eine oder auch mehrere Relationen als Eingabe und erzeugt eine Relation R als Ausgabe. Zu einem beliebigen Zeitpunkt τ muss R(τ) aus $R_1(\tau)$, ... , $R_n(\tau)$ bis zum Zeitpunkt τ berechenbar sein.

Ein relation-to-stream-Operator nimmt eine Relation R als Eingabe und erzeugt einen Datenstrom mit dem gleichen Schema wie R. Diese Operation muss ebenfalls zu einem beliebigen Zeitpunkt ausgeführt werden können.

Nachdem die Operatorklassen bekannt sind, bleibt noch die Frage der Ausgabe von Anfragen zu klären.

Zu betrachten ist eine Anfrage Q, die sich aus Operatoren zusammensetzt, die aus den eben genannten Operatorklassen stammen. Es wird angenommen, dass an den Blättern des Anfragegraphen Datenelemente sowohl aus Datenströmen S_1, ..., S_n ($n \geq 0$) als auch Relationen R_1 , ... , R_m ($m \geq 0$) kommen können.

Es gibt zwei Fälle zu unterscheiden:

Fall 1: Der oberste Operator der Anfrage Q gehört zu der relation-to-stream Klasse. Dann wird die Ausgabe der Anfrage zu einem Zeitpunkt τ ein Datenstrom S bis zum Zeitpunkt τ sein.

Fall 2: Der oberste Operator der Anfrage Q gehört zu der Klasse der stream-to-relation oder relation-to-relation Klasse. Dann wird das Anfrageresultat zu einem Zeitpunkt τ die Relation R(τ) sein.

6.1 Die Anfragesprache CQL

Betrachtet wird folgende CQL (Continuous Query Language) Anfrage [ArBW03]:

```
Select P.price
From Items[Rows 5] as I, PriceTable as P
Where I.itemID = P.itemID
```

Hier ist die Semantik dieser Anfrage nicht offensichtlich. Es stellt sich die Frage, ob das Ergebnis dieser Anfrage eine Relation oder ein Datenstrom sein wird. Ausserdem stellt sich bezogen auf dieses Beispiel noch eine weitere Frage, nämlich, was mit dem Resultat der Anfrage passieren wird, wenn der Preis eines kürzlich gekauften Artikels in dem 5-Elemente-Fenster, das in CQL

durch `[Rows 5]` spezifiziert wird, noch im Fenster enthalten ist und sich verändert. Diese beiden Fragen werden sich am Ende dieses Abschnittes beantworten lassen.

Wir werden jetzt die bereits auf konzeptionellem Niveau vorgestellten Operatoren zur Verknüpfung von Datenströmen und Relationen konkretisieren.

- Stream-to-relation-Operatoren

Stream-to-relation-Operatoren basieren auf dem Konzept eines Fensters. Man unterscheidet in CQL Zeitfenster, Tupel-basierte Fenster und partitionierte Fenster.

Zeitbasierte Fenster: Ein zeitbasiertes Fenster definiert eine Ausgaberelation über der Zeit durch Gleiten eines Intervalls der Grösse T und Festhalten der jeweils jüngsten Daten.

Formal: $R(\tau) = \{s \mid \langle s, \tau` \rangle \in S \wedge (\tau` \leq \tau) \wedge (\tau` \geq \max\{\tau - T, 0\})\}$

Es gibt zwei Spezialfälle bei der Definition von zeitbasierten Fenstern:

T = 0, in CQL umgesetzt durch S[Now] ($R(\tau)$ besteht aus den Datenstromelementen mit Schema S und Zeitstempel τ)

T = ∞, in CQL umgesetzt durch S[Range Unbounded] ($R(\tau)$ besteht aus allen Datenstromelementen bis zum aktuellen Zeitpunkt τ)

Tupelbasierte Fenster: Ein tupelbasiertes Fenster definiert eine Ausgaberelation über der Zeit durch Gleiten eines Intervalls der Grösse N und Festhalten der jeweils N letzten Datentupel.

Spezialfall:
N = ∞, in CQL umgesetzt durch S[Rows Unbounded] äquivalent zu S[Range Unbounded].

Ein partitioniertes Fenster auf einem Datenstrom S hat eine positive ganze Zahl N und eine Teilmenge der Schemaattribute von S als Parameter. Dieses partitioniert dann den Datenstrom S basierend auf der Gleichheit der Schemaattribute (vgl. GROUP BY in SQL).

- Relation-to-relation-Operatoren

Ein relation-to-relation-Operator in CQL ist direkt aus SQL abgeleitet.

Beispiel:

```
Select Distinct vehicleId
From PosSpeedStr [Range 30 seconds]
```

Diese Anfrage setzt sich zusammen aus einem stream-to-relation-Operator (Fensteroperator) und einem relation-to-relation-Operator, der eine Projektion mit Duplikateliminierung vornimmt (distinct). Die Ausgaberelation dieser Anfrage enthält zu einem beliebigen Zeitpunkt τ für aktive Fahrzeuge diejenigen, die in den letzten dreißig Sekunden eine Positions- und Geschwindigkeitsmessung an ein System gesendet haben. Dabei ist zu bemerken, dass die Ausgaberelation zeitvariant ist, d.h. wenn man die Anfrage zu zwei unterschiedlichen Zeitpunkten auslöst, können die jeweiligen Ausgaberelationen zu den beiden Zeitpunkten auch verschieden aussehen.

- Relation-to-stream-Operatoren

CQL besitzt drei relation-to-stream-Operatoren: Istream, Dstream und Rstream

Istream („insert stream“) angewendet auf die Relation R enthält ein Stream-Element
$< s, \tau >$, wenn s in $R(\tau) - R(\tau - 1)$ enthalten ist.

Istream (R) = $\cup ((R(\tau) - R(\tau -1)) \times \{\tau\})$ für $\tau \geq 0$

Mit Istream(R) können alle Datentupel die zwischen den Zeitpunkten τ und τ -1 in die Relation R neu hinzugekommen sind, einem Datenstrom angehangen werden.

Analog ist Dstream („delete stream“) definiert für $\tau > 0$.

Rstream („relation stream“) angewendet auf die Relation R enthält ein Stream-Element
$< s, \tau >$ wenn s in R ist zum Zeitpunkt τ.

Rstream(R) = $\cup (R(\tau) \times \{\tau\})$ für $\tau \geq 0$

Rstream(R) nimmt alle Datenelemente aus der Relation R die zum Zeitpunkt τ in der Relation R sind und wandelt sie in Datenstromelemente.

Beispiel 4:

(a) Betrachtet wird die CQL Anfrage

```
Select Istream(*)
From PosSpeedStr[Range Unbounded]
Where speed > 65
```

Die Anfrage setzt sich aus drei Operatoren zusammen: Einem unbeschränkten Fenster, das eine Relation erzeugt, die zum Zeitpunkt τ alle Geschwindigkeitsmessungen bis zum Zeitpunkt τ enhält; einem Filter-Operator, der die Relation auf Messungen >65 MPH einschränkt und einem Istream-Operator, der neue Datentupel in die eingeschränkte Relation einfügt und so das Resultat der Anfrage erzeugt. Die Relation wird so zu einem Ausgabestrom.

Istream kann in Rstream umgeschrieben werden :

(b) Dann sieht die Anfrage so aus:

```
Select Rstream(*)
 From PosSpeedStr[Now]
 Where speed > 65
```

Sie kann auch in Kurzform geschrieben werden als:

(c) `Select * From PosSpeedStr where speed >65`

Wenn in CQL auf ein Datenstrom zugegriffen wird, wo eigentlich eine Relation erwartet wird, wird ein unbeschränktes Fenster standardmäßig benutzt. (siehe Beispiel 4 (a)). Deshalb ist die Umformung von (a) bis (c) möglich. Weil die Ausgaberelation des Fensters und der Filter-Operator monoton bezüglich der Zeit sind, kann man den Istream-Operator auf das durch die Filterung eingeschränkte Fenster anwenden.

Jetzt lässt sich auch die Anfangsfrage dieses Abschnittes eindeutig beantworten: Die Ausgabe der Anfrage wird eine Relation sein, da die FROM Klausel die Ausgabe auf die letzten 5 Tupel mit dem Relationen-Schema, das durch den Join von PriceTable und Items entsteht, einschränkt. Wie bereits erwähnt, sind diese Tupel zeitvariant, und deshalb wird bei einer Preisänderung ein Tupel, was bereits in der Ausgabe enthalten war, ein zweites Mal mit verändertem Preis auftauchen.

7. Fazit

Wir haben die Operatoren auf dem neuen Datenmodellierungskonzept, den Datenströmen, mit ihren Hauptproblematiken behandelt und festgestellt, dass man die meisten Operatoren nicht ohne Anpassung aus dem Relationenmodell übernehmen kann. Das liegt hauptsächlich daran, dass man es mit potentiell unbegrenzten Datenmengen zu tun hat. Man hat erkannt, dass die notwendige Anpassung, Markierungen sind. Diese wurden hier eingeführt. Damit ist man nun in der Lage, Operatoren, die vorher nicht aus dem Relationenmodell auf das Datenstrommodell übertragbar waren, auf dieses zu übertragen. Es wurde auch ein Anwendungsbeispiel vorgestellt, in dem solche Markierungen verwendet werden.

Es wurde auch eine Implementierung eines Join-Operators vorgestellt. Zum Abschluss wurde betrachtet, wie man Relationen und Datenströme kombinieren kann. Dies wird notwendig, wenn man eine konkrete Anfragesprache definieren will, die sowohl auf Relationen als auch auf Datenströmen arbeitet. Es wurden die damit verbundenen Probleme behandelt. Es ist also ein Umdenken notwendig gewesen als man erkannt hat, dass es eine Vielzahl von Anwendungen gibt, die ein neues Datenmodell erforderlich machen, und da sicherlich auch noch neue Anwendungen hinzukommen werden, muss dieses Gebiet auch in Zukunft noch behandelt werden.

Literaturverzeichnis

[BBD+02] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, Jennifer Widom:
Models and Issues in Data Stream Systems
In: Proceedings of 21st ACM Symposium on Principles of Database Systems (PODS), 2002
Elektronisch verfügbar unter: http://dbpubs.stanford.edu/pub/2002-19

[GoTa03] Lukasz Golab, M.Tamer Özsu:
Processing Sliding Window Multi-Joins in Continuous Queries over Data Streams
VLDB Conference, Berlin, Germany, 2003
Elektronisch verfügbar unter:
http://db.uwaterloo.ca/~ddbms/publications/stream/vldb03.pdf

[RaRH99] Vijayshankur Raman, Bhaskaran Raman, Joseph M. Hellerstein:
Online Dynamic Reordering for Interactive Data Processing
VLDB Conference, Edinburgh, Scotland, 1999
Elektronisch verfügbar unter: http://citeseer.ist.psu.edu/raman99online.html

[TMS+02] Pete Tucker[1], David Maier[1], Tim Sheard[1], Leonidas Fegaras[2]:
Punctuating Continuous Data Streams
[1]OGI School of Science & Engineering at OHSU, [2]University of Texas at Arlington
Tech. Report, February 2002

[ArBW03] Arvind Arasu, Shivnath Babu, Jennifer Widom:
The CQL Continuous Query Language: Semantic Foundations and Query Execution
Stanford University
Tech. Report, 2003
Elektronisch verfügbar unter: http://citeseer.ist.psu.edu/arasu03cql.html