Integriertes Seminar Datenbanken und Informationssysteme
im Sommersemester 2005

Data Streams

Thema: Operatoren auf Data Streams

Bearbeiter: Ou Yi

Betreuer: Dipl.-Inf. Christian Mathis

10. Juni 2005

Inhaltsverzeichnis

1 Einleitung 1
2 Anforderungen an Operatoren auf Datenströmen 3
  2.1 Blockierende Operatoren müssen nicht-blockierend arbeiten 3
  2.2 Unterstützung der Approximierung 3
  2.3 Behandlung des Reihenfolgeproblems 4
3 Semantik der Operatoren auf Datenströmen 4
  3.1 Klassifikation der Operatoren 5
  3.2 STREAM 5
    3.2.1 Abstrakte Daten-Typen: Strom und Relation 6
      3.2.1.1 Strom 6
      3.2.1.2 Relation 6
    3.2.2 Abstrakte Semantik einer kontinuierlichen Anfrage: 7
    3.2.3 Implementierte Operatoren in STREAM-Projekt 8
      3.2.3.1 Strom-zu-Relation Operatoren 8
      3.2.3.2 Relation-zu-Relation-Operatoren 11
      3.2.3.3 Relation-zu-Strom-Operatoren 11
      3.2.3.4 Zusammenspiel der drei Operatorklassen 13
      3.2.3.5 System-Operatoren 13
      3.2.3.6 Zusammenfassung STREAM 14
  3.3 Aurora 14
    3.3.1 SQuAl-Datenmodell 14
    3.3.2 Reihenfolgeirrelevante (Order-agnostic) Operatoren 14
      3.3.2.1 Filter 15
      3.3.2.2 Map 15
      3.3.2.3 Union (Vereinigung) 15
    3.3.3 Reihenfolgesensitive (Order-sensitive) Operatoren 16
      3.3.3.1 BSort 17
      3.3.3.2 Aggregate 18
      3.3.3.3 Join (Verbund) 19
      3.3.3.4 Resample (Interpolation) 20
    3.3.4 Ein komplizierteres Beispiel 21
  3.4 Vergleich Aurora und STREAM 23
4 Über die Implementierung des Join-Operators 23
5 Zusammenfassung und Konklusion 24
6 Literatur 25

Operatoren auf Data Streams

OU YI
TU Kaiserslautern
o_yi@informatik.uni-kl.de

Zusammenfassung: Die Auswertung der kontinuierlichen Anfragen auf Datenströmen (Data Streams) bedarf neuer Überlegungen in vielen Aspekten. Insbesondere ist die Semantik der Operatoren auf Datenströmen anders als auf Relationen im Relationenmodell. In dieser Ausarbeitung wird die Semantik von Operatoren auf Datenströmen anhand der Ansätze von STREAM und Aurora, zweier DSMSen (Data Stream Management Systems), vorgestellt.

1 Einleitung

Traditionelle Datenbanksysteme (DBMS) verwalten alle Daten als persistente *Datensätze*. Es gibt jedoch viele Anwendungen, für die sich die Daten besser als *Datenströme* modellieren lassen. Oft ist ihre Verarbeitung in DBMS einfach nicht möglich, weil die Datenflut unbeherrschbar ist. Die Datensätze dort werden von der Datenquelle schnell und kontinuierlich generiert und die Datenquelle erschöpft möglicherweise nie. Beispiele von solchen Anwendungen sind Finanzanwendungen, Netzwerküberwachung, Webanwendungen, Produktionssteuerung, Sensornetzwerküberwachung usw. Das Datenstrommodell unterscheidet sich von dem traditionellen Relationenmodell in folgenden Aspekten[2]:

1. Die Daten-Elemente kommen kontinuierlich mit potenziell hoher Ankunftsrate beim DSMS an;
2. Die Reihenfolge, nach der die Daten-Elemente ankommen, kann verfälscht sein, z. B. wenn die von einem Sensornetzwerk generierten Daten in IP-Paketen gesendet werden;
3. Datenströme sind potenziell unendlich lang, d. h. es können unendlich viele Daten-Elemente in einem Strom enthalten sein;
4. Nach der Verarbeitung eines Daten-Elementes wird es entweder verworfen oder archiviert, d. h. es darf normalerweise nicht im Hauptspeicher bleiben, da dieser in der Regel im Vergleich zur Größe der Datenströme sehr klein ist.

Aus diesen Unterschieden geht hervor, dass traditionelle DBMS für solche Anwendungen nicht geeignet sind. Ein typisches Verarbeitungsvorgehen bei DBMS ist es z. B., Datensätze vor der Verarbeitung zu speichern. Das ist aber bei der Größe und der Menge sowie dem schnellen Aufkommen von Datenströmen unmöglich. Mit DSMSen (Data Stream Management Systems) versucht man, die Auswertung der Anfragen auf Datenströmen zu ermöglichen.

Anfragen in DSMS lassen sich in *kontinuierliche Anfragen* und *„One-time"-Anfragen* klassifizieren. Kontinuierliche Anfragen sind Anfragen, die einmal formuliert und dann kontinuierlich in der Datenbank ausgewertet werden, im Gegensatz zu „One-time"-Anfragen, die nur einmal ausgewertet werden um Antworten aus der Datenbank zu einem Zeitpunkt zu berechnen[2]. Eine weitere Unterscheidung der Anfragen ist zwischen *vordefinierten Anfragen* und *„Ad hoc"-Anfragen*. Vordefinierte Anfragen sind dem System bekannt, bevor die relevanten Daten-Elemente ankommen. Dagegen werden „Ad hoc"-Anfragen formuliert auf den ankommenden Daten-Elementen. Daher werden möglicherweise auch die historischen Daten von den „Ad hoc"-Anfragen referenziert. Vordefinierte Anfragen sind in der Regel auch kontinuierliche Anfragen, wobei vordefinierte „One-time"-Anfragen auch möglich sind. „Ad hoc"-Anfragen können sowohl kontinuierliche Anfragen als auch „One-time"-Anfragen sein. Für Datenstrom-Anwendungen sind kontinuierliche Anfragen die interessanteste Anfrage-Klasse. Neben der Anforderung für kontinuierliche Anfragen charakterisieren sich die Datenstrom-Anwendungen auch durch die Anforderung für rechtzeitige „Online"-Antwort, d. h. Verzögerung der Antwort oder Unterbrechung des Datenflusses sind unerwünscht.

Ein Ziel vom DSMSen (Data Stream Management Systems) lautet daher: die Anforderung von Datenstrom-Anwendungen zu erfüllen, d. h. rechtzeitige Auswertung der kontinuierlichen Anfragen. In manchen DSMS-Projekten versucht man neben der Datenstrom-Verarbeitung auch die Funktionen eines traditionellen DBMSs beizubehalten, also die Verarbeitung von Relationen weiter zu unterstützen. Die Unterscheidungen zwischen Datenstrommodell und Relationenmodell bedingen bei der Entwicklung eines DSMSs viele Herausforderungen. Insbesondere muss man nun die neue Semantik der Operatoren bedenken, weil sie auf Datenströmen, gegebenfalls auf eine Mischung von Datenströmen und Relationen, arbeiten. Wir werden im Abschnitt 2 im Zusammenhang mit der Strom-Verarbeitung diskutieren, welche neuen Probleme ein Operator behandeln muss.

Wie ein DBMS muss ein DSMS auch dem Benutzer die Formulierung von Anfragen ermöglichen. Aufgrund der Verschiedenheit der Verarbeitungsmodelle und der neuen Semantik der Operatoren kann man SQL, die weit verbreitete Anfragesprache im DBMS, nicht direkt auf DSMSe übertragen. Manche DSMS-Projekte verwenden eine SQL-basierte, deklarative Anfragesprache mit Strom-orientierter Erweiterung, z. B. CQL im STREAM-Projekt, um die formale Grundlage und zahlreiche Implementierungstechniken des Relationenmodells wiederverwenden zu können. Manche DBMS verzichten ganz auf SQL und bieten eine neue Benutzerschnittstelle, z. B. der „Box and Flow"-Ansatz von Aurora. Die Semantik der Operatoren auf Datenströmen werden wir im Abschnitt 3 vorstellen.

Die Relationenalgebra spielt eine fundamentale Rolle bei der Realisierung von DBMS, insbesondere bei der Anfrageoptimierung. Leider gibt es im Datenstrom-Bereich noch keine standardisierte Strom-Algebra. Jedes DSMS definiert seine eigene Menge von Operatoren. Auch sind die Strom-Elemente unterschiedlich modelliert. Wegen der großen Vielfalt der Konzepte ist eine vollständige Vorstellung aller möglichen Operatoren auf Datenströmen hier nicht möglich.

2 Anforderungen an Operatoren auf Datenströmen

2.1 Blockierende Operatoren müssen nicht-blockierend arbeiten

Eine der größten Herausforderungen für ein DSMS ist die Behandlung von *blockierenden Operatoren*. Ein blockierender Operator kann keine Elemente in seiner Ausgabe erzeugen, bis er seine ganze Eingabe gelesen hat. Typische blockierende Operatoren sind der Join-Operator, der Sortieroperator und die Aggregationsoperatoren wie *Sum*, *Count*, *GroupBy* usw. Ein *nicht-blockierender Operator* kann dagegen ein Ergebnistupel ausgeben, sobald ein Daten-Element aus der Eingabe bei ihm ankommt. Beispiele für nicht-blockierende Operatoren sind *Filter* und *Map*. Datenströme sind potenziell unendlich lang. Ein blockierender Operator, der auf einem Strom arbeitet, kann nie die ganze Eingabe lesen und deswegen nie in der Lage sein, eine Ausgabe zu erzeugen. Anwendungen oder andere Operatoren, die die Ausgabe des blockierenden Operators als Eingabe erwarten, müssen möglicherweise ewig warten. Das ist natürlich inakzeptabel. Trotzdem soll die Funktionalität des Operators im System zur Verfügung stehen. Deshalb wurden für solche Operatoren nicht-blockierende Verfahren mit ähnlicher Semantik vorgeschlagen, die im nächsten Abschnitt behandelt werden.

2.2 Unterstützung der Approximierung

Operatoren im Relationenmodell haben die Semantik für eine genaue Antwort. Das ist im Datenstrommodell oft unmöglich. Zum einen, da Datenströme potenziell unendlich sind, erwächst der Speicherplatzbedarf von einer Operation unbegrenzt, wenn eine exakte Antwort berechnet werden muss. In der Realität haben wir aber nur *begrenzten Speicherplatz*. Zum anderen, wenn die Datenquelle Daten-Elemente zu schnell generiert. Dann kann ein Operator „überfüttert“ sein, wenn er eine exakte Antwort berechnen muss. Das hat eine verzögerte Antwort oder einen unkontrollierten Datenverlust zur Folge. In der Praxis hat ein Operator nur *begrenzte Verarbeitungszeit* für jedes Daten-Element, denn die Antwortzeit ist für manche Anwendung kritisch. D. h., unter Umständen muss man bei der Datenstrom-Verarbeitung auf eine exakte Antwort verzichten und eine *Approximierung* akzeptieren. Falls zum Beispiel eine exakte Antwort nicht möglich ist, verarbeitet der Operator nicht mehr jedes Daten-Element, sondern nur strategisch ausgewählte Stichproben, um die Antwortzeit oder den Speicherplatzbedarf gering zu halten. Man muss also zwischen Speicherplatz, Antwortzeit und Genauigkeit einen Kompromiss schließen. Die Operatoren auf Datenströmen müssen dementsprechend eine neue Semantik haben, die auch approximierte Antworten berücksichtigt.

Sliding Windows (gleitende Fenster) sind eine gut akzeptierte Approximierungstechnik. *Sliding Window* bedeutet, man wertet die Anfrage nicht für alle angekommenen Daten-Elemente aus, sondern nur für ein Fenster über den Datenstrom. Typischerweise sind immer die jüngsten angekommenen Daten-Elemente im Fenster enthalten. Dieses Fenster ist entweder *zeitbasiert* oder

tupelbasiert. Zeitbasierte- und Tupelbasierte-Fenster werden in Abschnitt 3 vorgestellt. Für manche Anfragen ist diese Technik nicht nur eine Approximierung, die man zwangsweise in Kauf nehmen muss, sondern sogar gewünscht. Denn die neuesten Daten sind normalerweise interessanter als veraltete. Deshalb müssen Operatoren auf Datenströmen unter Umständen das Fenster-Konzept unterstützen.

2.3 Behandlung des Reihenfolgeproblems

Ein gleitendes Fenster bezieht sich auf einen *Zeitstempel* oder ein Sequenznummer-Attribut, das die Ankunftszeit beim System eines Daten-Elementes identifiziert. Das wäre klar und eindeutig, wenn die Daten-Elemente in einem einzigen Strom ankämmen und in der gleichen Reihenfolge, in der der Zeitstempel erzeugt wurde. Leider ist es in der Praxis oft nicht der Fall. Daten-Elemente kommen asynchron in verschiedenen Strömen an und die Reihenfolge kann verfälscht sein. In diesem Fall muss man z. B. bedenken: Soll ein Tupel mit Zeitstempel 4 noch berücksichtigt werden, wenn es spät ankommt, d. h. wenn ein Tupel mit Zeitstempel 5 schon im Fenster enthalten ist? Oder, was soll der Zeitstempel eines in einer Verbund-Operation neu erzeugten Daten-Elementes sein, wenn die zwei verbundenen Daten-Elemente verschiedene Zeitstempel haben? Operatoren auf Datenströmen müssen in einem solchen Fall auch eine deterministische Antwort erzeugen können.

Man unterscheidet *implizite Zeitstempel* und *explizite Zeitstempel*. Impliziter Zeitstempel bedeutet, dass das System einen Zeitstempel oder ein Sequenznummer-Attribut als ein Extra-Feld zu jedem ankommenden Daten-Elemente hinzufügt. Expliziter Zeitstempel bedeutet, ein (monotones und sequenzielles) Attribut des Daten-Elementes ist als Zeitstempel vorgesehen. Ein impliziter Zeitstempel wird verwendet, wenn ein solches Attribut nicht von der Datenquelle erzeugt wird, oder wenn der zu einem Daten-Element gehörige, exakte Zeitpunkt nicht wichtig ist. Wenn der exakte Zeitpunkt wichtig ist, dann muss einen expliziten Zeitstempel von der Datenquelle erzeugt werden.

Explizite Zeitstempel haben wegen der potenziell verfälschten Ankunftsreihenfolge von Daten-Elementen einen Nachteil. Daten-Elemente mit jüngeren Zeitstempeln können vor den Elementen mit älteren Zeitstempeln ankommen. Das macht die Fenster-basierte Berechnung oder andere von dieser Reihenfolge abhängige Berechnungen schwerer. Allerdings ist das Problem mit Pufferung und Sortierung lösbar, unter der Annahme, dass die Daten-Elemente immer in einer grob richtigen Reihenfolge ankommen. Die Pufferung und Sortierung kann auch die Aufgabe eines Operators sein.

3 Semantik der Operatoren auf Datenströmen

In diesem Abschnitt betrachten wir zuerst die Klassifikation von Operatoren in DSMS. Dann stellen wir die typischen Operatoren in zwei DSMS-Projekten, STREAM und Aurora, vor.

3.1 Klassifikation der Operatoren

Zum Vergleich listen wir zuerst die typischen Operatoren im Relationenmodell auf:

1. Klassische Mengenoperationen:
   Union, *Difference*, *Catesian Product*, (ableitbar: *Intersection*)
2. Relationsoperationen:
   Selection, *Projection*, (ableitbar: *Join*, *Division*)

Die o. g. Operatoren sind typischerweise mit erweiterter Semantik auch in DSMS vorhanden, weil diese statt auf reinen Relationen auch auf Datenströmen arbeiten müssen. Es gibt noch keine standardisierte „Strom-Algebra". Verschiedene DSMS unterstützen verschiedene Mengen von Operatoren. Typische Operatoren auf Datenströmen sind: *Filter*, *Map*, *Union*, *Aggregate* und *Join*. *Filter* und *Map* sind jeweils vergleichbar mit *Selection* und *Projection* im Relationenmodell. *Union* hat normalerweise die Multimengensemantik (duplizierte Elemente erlaubt). *Aggregate* ist für Gruppierung und Aggregation verantwortlich. *Join* hat normalerweise die Fenstersemantik.

Nach [7] lassen sich Operatoren in DSMS in *logische Operatoren* und *physische Operatoren* klassifizieren. Logische Operatoren dienen zur Festlegung der Semantik von Anfragesprachen und zur Anfrageoptimierung, z. B. „selection push-down". Physische Operatoren bilden die Implementierung von logischen Operatoren. Zwischen den beiden gibt es aber keine Eins-zu-Eins-Abbildung. In dieser Ausarbeitung betrachten wir hauptsächlich die Semantik von Operatoren auf Datenströmen, d. h. wir werden uns auf logische Operatoren konzentrieren.

Physische Operatoren lassen sich weiter klassifizieren in zustandslose Operatoren und zustandsbehaftete Operatoren. Typische zustandslose Operatoren sind *Filter*, *Map* und *GroupBy*. Der Filter Operator z. B. wertet ein benutzerdefiniertes Prädikat auf jedes ankommende Element aus und gibt das Element sofort dem Ausgabe-Strom, falls das Prädikat erfüllt ist. Ansonsten verwirft er das Element. Solche Operatoren, die sofort eine Ausgabe erzeugen können, müssen ihren Zustand nicht halten, deshalb sind sie zustandslos. Andere Operatoren brauchen eine interne Datenstruktur, um ihre Status zu merken, z. B. Sie müssen ein Fenster von Elementen puffern. Solche Operatoren heißen zustandsbehaftete Operatoren. Typische zustandsbehaftete Operatoren sind *Join* und *Aggregation*.

Bei manchen DSMS gibt es zusätzlich die Unterscheidung zwischen „Query-Operatoren" und „System-Operatoren". Query-Operatoren legen die Semantik der Anfragen fest und System-Operatoren isolieren Query-Operatoren von Aspekten der unteren Ebenen.

3.2 STREAM

STREAM [4][5][6] ist ein DSMS-Prototyp der Stanford Universität. STREAM orientiert sich für die Auswertung der kontinuierlichen Anfragen auf Datenströmen und Relationen. Für diese

Anfragen wurde zuerst eine abstrakte Semantik auf der Basis von drei „black box"-Komponenten definiert:

1. Eine Menge von Strom-zu-Relation-Operatoren,
2. Eine Menge von Relation-zu-Relation-Operatoren,
3. Eine Menge von Relation-zu-Strom-Operatoren.

Bei der Implementierung des Systems werden dann die drei Komponenten instanziert.

3.2.1 Abstrakte Daten-Typen: Strom und Relation

Im abstrakten Modell von STREAM existieren neben Relationen auch Ströme. Diese zwei Datentypen zusammen mit der Abbildung zwischen ihnen dienen als die Basis für die abstrakte Semantik, die wir im folgenden Abschnitt vorstellen wollen.

Jeder Strom besitzt, wie eine Relation, eine feste Menge von Attributen. Des weiteren existiert eine globale *Zeit-Domäne* T, mit diskreten und geordneten Elementen, welche nicht unbedingt der Uhrzeit entstammen müssen. Beispielsweise kann eine Zeit-Domäne die geordnete Menge der natürlichen Zahlen sein. Die Zeit-Domäne dient dazu, den Daten eines ankommenden Stroms sowie den Änderungen von Relationen Zeitwerte zuzuordnen.

3.2.1.1 Strom

Ein *Strom* wird dabei formal wie folgt definiert: Ein *Strom* S ist eine möglicherweise unendliche Multimenge von Elementen $<s, t>$, wobei s ein Tupel der Attributmenge von S ist und $t \in T$ der Zeitstempel des Elements.

Ein Strom-Element $<s, t> \in S$ bedeutet, das Tupel s kommt beim Strom S zu Zeitpunkt t an. Der Zeitstempel gehört nicht zur Menge der Attribute. Es können mehrere Elemente mit dem gleichen Zeitstempel existieren. Insbesondere sind duplizierte Elemente nach dem Multimengekonzept erlaubt. Es gibt zwei Arten von Strömen: *Basisströme*, diese sind die Quellströme, wie sie im DSMS ankommen, sowie *abgeleitete Ströme*, die von Operatoren einer Anfrage erzeugt wurden. Der Begriff „Tupel eines Stroms" bezeichnet den *Datenanteil (ohne Zeitstempel) eines Elementes* eines Stroms.

3.2.1.2 Relation

Eine *Relation* wird wie folgt definiert: Eine Relation R ist eine Abbildung von T auf eine endliche, aber unbegrenzte Multimenge von Tupeln, die zu Attributmenge von R gehören.

Eine Relation R im STREAM ist eine Abbildung von der Zeitdomäne auf eine Multimenge. Diese Abbildung bestimmt eine ungeordnete Multimenge von Tupeln zu jedem Zeitpunkt $t \in T$, geschrieben $R(t)$. $R(t)$ kann als eine traditionelle Relation in Relationenmodell betrachtet werden. Ein Unterschied zwischen der Relationsdefinition im STREAM und der herkömmlichen Relati-

on ist jedoch zu beachten: Im Relationenmodell ist eine Relation einfach eine Menge (oder Multimenge) von Tupeln. Dabei ist der Zeitbezug für die Semantik einer Anfrage irrelevant. Eine Relation in einer STREAM-Anfrage ändert sich mit der Zeit. Mit R bezeichnet man in STREAM eine zeitvariierende Relation. Informal gesagt, wenn R zum Beispiel die Ausgabe-Relation einer Anfrage ist, kann der Datenbestand von R zu jedem Zeitpunkt verschieden sein. Denn die Anfrage wird kontinuierlich ausgewertet und deswegen jedes neue Tupel in der Eingabe kann die Ausgabe R ändern. Auch hier gibt es ***Basis-Relationen*** und ***abgeleitete Relationen***, die eine gespeicherte Relation bzw. Ausgaberelation eines Operators bezeichnen.

Der Zeitstempel t eines Strom-Elementes $\langle s, t\rangle$ bezieht sich auf die logische Zeit, die von der Anwendungssemantik bestimmt wird. Z. B., für eine Anwendung ist der exakte Zeitpunkt, zu dem ein Daten-Element erzeugt wird, nicht interessant. Für sie ist aber die Reihenfolge der Elemente wichtig. Dann kann die Zeitdomäne für diese Anwendung die Menge der natürlichen Zahlen sein. Der Zeitstempel t, der von der Datenquelle erzeugt werden kann, muss natürlich nicht der physischen Ankunftszeit beim System entsprechen. Ähnlicherweise bezieht sich der Zeitpunkt t in einer Relation $R(t)$ auch auf die logische Zeit und nicht auf die physische Zeit.

3.2.2 Abstrakte Semantik einer kontinuierlichen Anfrage:

Die abstrakte Semantik einer kontinuierlichen Anfrage basiert auf drei Klassen von Operatoren auf Strömen und Relationen:

1. Ein Strom-zu-Relation-Operator nimmt einen Strom S als Eingabe und gibt eine Relation R mit dem gleichen Schema wie S aus. Zu jedem Zeitpunkt t muss $R(t)$ aus der Menge $\{\langle s,t'\rangle \in S \mid t' \leq t\}$ (alle Elemente in S, deren Zeitstempel t' kleiner gleich t sind) berechnet werden können. Nach [5] ist eine beliebige Fensterspezifikationssprache eine Menge von Strom-zu-Relation-Operatoren.
2. Ein Relation-zu-Relation-Operator nimmt einen oder mehrere Relationen $R_1, \ldots, R_n$ als Eingabe und gibt eine Relation R aus. Zu jedem Zeitpunkt t muss $R(t)$ aus $R_1(t), \ldots, R_n(t)$ berechnet werden können. Nach [5] ist eine beliebige relationale Anfragesprache eine Menge von Relation-zu-Relation-Operatoren.
3. Ein Relation-zu-Strom-Operator nimmt eine Relation R als Eingabe und gibt einen Strom S mit dem gleichen Schema wie R aus. Zu jedem Zeitpunkt t müssen die Elemente in S mit Zeitstempel t aus $R(t')$ für alle $t' \leq t$ berechnet werden können.

Es existiert kein Strom-zu-Strom-Operator. Dieser kann jedoch durch eine Zusammensetzung aus den anderen drei Operator-Typen kombiniert werden. So kann die Semantik im Relationenmodell möglichst weitgehend ausgenutzt werden. Z. B. sieht so die Auswertung einer Anfrage aus, die einen Strom als Eingabe nimmt und einen Strom ausgibt: Der Eingabe-Strom wird zuerst mit einem Strom-zu-Relation-Operator in eine Relation umgewandelt, auf der Relation wird die Berechnung mit Relationensemantik ausgeführt und das Ergebnis wird mit einem Relation-zu-Strom-Operator in Ausgabe-Strom umgewandelt. Zu beachten: es ist natürlich unmöglich, einen ganzen Strom als Relation vor der Verarbeitung abzuspeichern. Der Strom-zu-Relation-

Operator hier ist im Wesentlichen ein Fensteroperator und die Relation, die er erzeugt, ist ein Fenster über den Eingabe-Strom. Weil die Anfrage kontinuierlich ausgewertet wird, wird den Inhalt des Fensters kontinuierlich aktualisiert. Die Interaktion zwischen den drei Klassen von Operatoren ist in Abbildung 3-1 gezeigt.

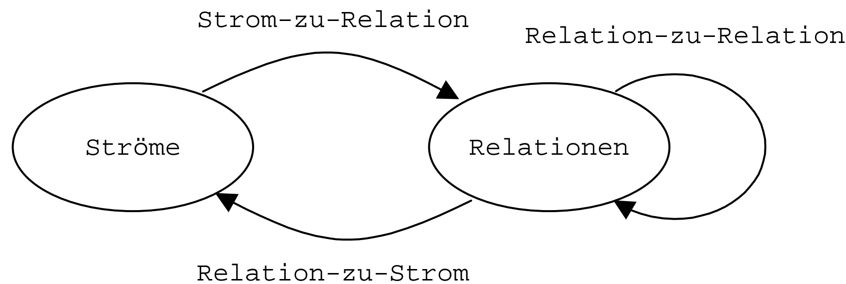

Abbildung 3-1 Operatorklassen und Abbildungen zwischen Strömen und Relationen

Eine kontinuierliche Anfrage ist eine typkonsistente Zusammensetzung aus den Strömen, Relationen und Operatoren wie in Abbildung 3-1 dargestellt ist. Die Ausgabe einer kontinuierlichen Anfrage ist entweder eine Relation oder ein Strom. Wir können die Semantik einer kontinuierlichen Anfrage in diesen zwei Fällen betrachten:

1. Eine kontinuierliche Anfrage, die einen Strom ausgibt: Zu einem Zeitpunkt t sind alle Eingabe-Elemente mit Zeitstempel $t' \leq t$ bearbeitet und die kontinuierliche Anfrage sendet jedes neue Ergebnis-Element mit Zeitstempel t aus.
2. Eine kontinuierliche Anfrage, die eine Relation ausgibt: Zu einem Zeitpunkt t sind alle Eingabe-Elemente mit Zeitstempel $t' \leq t$ bearbeitet und die kontinuierliche Anfrage aktualisiert die Ausgabe-Relation zu $R(t)$.

3.2.3 Implementierte Operatoren in STREAM-Projekt

Die implementierten Operatoren in STREAM-Projekt sind in Tabelle 3-1 aufgelistet [6]. Jeder Operator ist entweder ein CQL[1]-Operator oder ein System-Operator. Jeder CQL-Operator gehört zu einer der drei Operator-Klassen, wie sie im letzten Abschnitt vorgestellt wurden. Die Anfragesprache CQL im STREAM-Projekt ist im Wesentlichen eine Strom-orientierte Erweiterung von SQL99. System-Operatoren isolieren die CQL-Operatoren von den Aspekten der unteren Systemebenen wie z. B. die Störung in der Ankunftsreihenfolge der Strom-Elemente, „load shedding“, und externes Strom-Datenformat.

3.2.3.1 Strom-zu-Relation Operatoren

Der CQL-Operator *seq-window* ist der einzige Strom-zu-Relation-Operator. Er unterstützt drei Typen von Fenster-Operationen: tupelbasiert, zeitbasiert und partitioniert. Aus einer allgemeineren Sicht können auch andere Typen von Fenstern oder andere Techniken implementiert werden,

[1] CQL (Continuous Query Language) ist die Anfragesprache des STREAM-DSMSs

um Ströme in Relationen zu verwandeln, solange sie der abstakten Semantik aus letztem Abschnitt entsprechen.

<table>
<tr><th>Name</th><th>Operator-Typ</th><th>Beschreibung</th></tr>
<tr><td>seq-window</td><td>Strom-zu-Relation</td><td>implementiert zeitbasierte, tupelbasierte und partitionierte Fenster</td></tr>
<tr><td>select</td><td>Relation-zu-Relation</td><td>filtert Tupel durch Prädikat(e)</td></tr>
<tr><td>project</td><td>Relation-zu-Relation</td><td>Projektion mit Duplikaten</td></tr>
<tr><td>binary-join</td><td>Relation-zu-Relation</td><td>Binär-Verbund</td></tr>
<tr><td>mjoin</td><td>Relation-zu-Relation</td><td>Multi-Verbund</td></tr>
<tr><td>union</td><td>Relation-zu-Relation</td><td>Multimengen-Vereinigung</td></tr>
<tr><td>except</td><td>Relation-zu-Relation</td><td>Multimengen-Differenz</td></tr>
<tr><td>intersect</td><td>Relation-zu-Relation</td><td>Multimengen-Durchschnitt</td></tr>
<tr><td>antisemijoin</td><td>Relation-zu-Relation</td><td>Antisemi-Verbund von zwei Eingabe-Relationen</td></tr>
<tr><td>aggregate</td><td>Relation-zu-Relation</td><td>Gruppierung und Aggregation</td></tr>
<tr><td>duplicate-eliminate</td><td>Relation-zu-Relation</td><td>Duplikateliminierung</td></tr>
<tr><td>i-stream</td><td>Relation-zu-Strom</td><td>implementiert IStream-Semantik</td></tr>
<tr><td>d-stream</td><td>Relation-zu-Strom</td><td>implementiert DStream-Semantik</td></tr>
<tr><td>r-stream</td><td>Relation-zu-Strom</td><td>implementiert Rstream-Semantik</td></tr>
<tr><td>stream-shepherd</td><td>System-Operator</td><td rowspan="6">isolieren die CQL-Operatoren von den Problemen der unteren Ebenen</td></tr>
<tr><td>stream-sample</td><td>System-Operator</td></tr>
<tr><td>stream-glue</td><td>System-Operator</td></tr>
<tr><td>rel-glue</td><td>System-Operator</td></tr>
<tr><td>shared-rel-op</td><td>System-Operator</td></tr>
<tr><td>output</td><td>System-Operator</td></tr>
</table>

Tabelle 3-1 Die implementierten Operatoren in STREAM

Zeitbasierte Fenster

Ein *zeitbasiertes Fenster* auf einen Strom S nimmt ein Zeitintervall Z als Parameter und gibt eine Relation aus, die aus den neuesten Tupeln in S besteht. Die Ausgabe-Relation zum Zeitpunkt t kann formaler definiert werden als:

$$R(t) = \{ s \mid < s, t' > \in S \wedge t' \in [\max(t - Z, 0), t] \}$$

D. h., die Ausgabe Relation R zum Zeitpunkt t enthält die Datenanteile von solchen Elementen aus S als Tupel, deren Zeitstempel größer gleich 0 und kleiner gleich t ist, wobei die Entfernung zwischen t' und t kleiner gleich Z sein muss. Zwei Extremfälle sind wichtig: $Z = 0$ und $Z = \infty$. Falls $Z = 0$, $R(t)$ enthält die Datenanteil als Tupel nur von solchen Elementen aus S, deren Zeitstempel gleich t sind. Falls $Z = \infty$, $R(t)$ enthält die Datenanteil als Tupel von allen Elementen aus

S, deren Zeitstempel kleiner gleich *t* sind.

```
Beispiel
Eingabe-Strom-Schema: <Datenanteil, Zeitstempel>
Eingabe-Strom:
<a, 0>, <b, 0>, <c, 1>, <c, 2>, <d, 2>, <e, 3>, <f, 4>, <f, 5>...
Parameter: Z = 2

 [max(t-Z, 0), t]:      [0, 0]  [0, 1]  [0, 2]  [1, 3]  [2, 4]

 Ausgabe-Relation:       R(0)    R(1)    R(2)    R(3)    R(4)    ...
 R(t)                    <a>     <a>     <a>     <c>     <c>
                         <b>     <b>     <b>     <c>     <d>
                                 <c>     <c>     <d>     <e>
                                         <c>     <e>     <f>
                                         <d>
```

Tupelbasierte Fenster

Ein *tupelbasiertes Fenster* auf einen Strom S nimmt einen positiven Integer *N* als Parameter und gibt eine Relation aus, die aus den letzten *N* Tupeln eines geordneten Stroms besteht. Die Ausgabe-Relation *R* zum Zeitpunkt *t* enthält die *N* Tupel, die die Datenanteile von solchen Elementen aus *S* sind, welche die größten Zeitstempel haben, oder alle Elemente, wenn die Größe von *S* bis zum Zeitpunkt *t* kleiner oder gleich *N* ist.

```
Beispiel
Eingabe-Strom-Schema: <Datenanteil, Zeitstempel>
Eingabe-Strom:
<a, 0>, <b, 0>, <c, 1>, <c, 2>, <d, 2>, <e, 3>, <f, 4>, <f, 5>...
Parameter: N = 2
Ausgabe-Relation R(t):

    R(0)      R(1)       R(2)     R(3)       R(4)      ...
    <a>     <a>or<b>     <c>    <c>or<d>     <e>
    <b>       <c>        <d>      <e>        <f>
```

Tupelbasiertes Fenster hat einen Nachteil: Falls die Zeitstempel im Eingabe-Strom nicht eindeutig sind, erzeugt der Fenster-Operator ein nicht-deterministisches Ergebnis. Das sieht man im obigen Beispiel: zum Zeitpunkt *t=1,* es gibt drei Elemente <a, 0>, <b, 0>, <c, 1>, die die Kriterien erfüllen, aber die Relation *R* darf zu jedem Zeitpunkt nur zwei Tupel enthalten.

Partitionierte Fenster

Ein *patitioniertes Fenster* auf einen Strom *S* nimmt einen positiven Integer *N* und eine Submenge der Attributmenge von *S* als Parameter. Basierend auf Gleichheit der Attribute, ähnlich einem *GroupBy* in Relationenmodell, werden Subströme erzeugt. Auf jedem Substrom wird ein Tupelbasiertes Fenster der Größe *N* erzeugt. Als Ausgabe ergibt sich die Vereinigung dieser Fenster.

Beispiel

Eingabe-Strom-Schema: <A, B, Zeitstempel>

Eingabe-Strom:

<a, 1, 0>, <b, 1, 0>, <a, 2, 1>, <c, 1, 1>, <a, 3, 2>,
<b, 2, 2>, <c, 2, 2>, <c, 3, 3>, <a, 4, 4>, <a, 5, 5>...

Parameter: N = 2, partitioniert nach A

Subströme, die auf Gleichheit von Attribut A basieren:

A	t = 0	t = 1	t = 2	t = 3	t = 4	t = 5	...
a	<a, 1, 0>	<a, 2, 1>	<a, 3, 2>		<a, 4, 4>	<a, 5, 5>	...
b	<b, 1, 0>		<b, 2, 2>				...
c		<c, 1, 1>	<c, 2, 2>	<c, 3, 3>			...

Ausgabe-Relation R:

R(0)	R(1)	R(2)	R(3)	R(4)	R(5)	...
<a, 1>	<a, 1>	<a, 1>	<a, 2>	<a, 3>	<a, 4>	
<b, 1>	<a, 2>	<a, 2>	<a, 3>	<a, 4>	<a, 5>	
	<b, 1>	<b, 1>	<b, 1>	<b, 1>	<b, 1>	
	<c, 1>	<b, 2>	<b, 2>	<b, 2>	<b, 2>	
		<c, 1>	<c, 2>	<c, 2>	<c, 2>	
		<c, 2>	<c, 3>	<c, 3>	<c, 3>	

3.2.3.2 Relation-zu-Relation-Operatoren

Die Relation-zu-Relation-Operatoren sind abgeleitet vom Relationenmodell. Überall, wo eine traditionelle Relation in einer SQL-Anfrage referenziert werden kann, kann sie auch in CQL referenziert werden. Das ist ein Vorteil des STREAM-Ansatzes: die formale Grundlage und zahlreiche Implementierungstechniken für das Relationenmodell können wiederverwendet werden. Die Relation-zu-Relation-Operatoren in STREAM sind sehr ähnlich zu den Operatoren im Relationenmodell und haben eine offensichtliche Semantik, deswegen werden wir auf ihre Vorstellung verzichten. Im Kontext von einer kontinuierlichen Anfrage muss man nun bedenken, dass die Relationen mit der Zeit variieren.

3.2.3.3 Relation-zu-Strom-Operatoren

Es gibt drei CQL-Operatoren, um Relationen auf Ströme abzubilden: *IStream, DStream* und *RStream*. Die formale Semantik ist wie folgt definiert (seien $\cup$, $\times$ und – die Multimengen-Operatoren):

IStream

IStream (insert stream) angewandt auf eine Relation R enthält ein Strom-Element <*s,t*> immer dann, wenn das Tupel *s* in *R(t) - R(t-1)* ist: (*R(-1)* sei die leere Menge).

$$IStream(R) = \bigcup_{t \geq 0}((R(t) - R(t-1)) \times \{t\})$$

Beispiel

```
        R(0)            R(1)            R(2)            ...
        <a>             <c>             <c>
        <b>                             <d>

   R(0) - R(-1)      R(1) - R(0)     R(2) - R(1)
     <a>, <b>           <c>             <d>

Ausgabe-Strom: <a, 0>, <b, 0>, <c, 1>, <d, 2> ...
```

D. h., *IStream* gibt alle Tupel, die bis Zeitpunkt *t* je in Relation *R* eingefügt wurden, in einem Strom aus.

DStream

DStream (delete stream) angewandt auf eine Relation *R* enthält ein Strom-Element <*s*,*t*> immer dann, wenn das Tupel *s* in *R(t) - R(t-1)* ist:

$$DStream(R) = \bigcup_{t > 0}((R(t-1) - R(t)) \times \{t\})$$

Beispiel

```
        R(1)            R(2)            R(3)            ...
        <a>             <c>             <c>
        <b>                             <d>

   R(0) - R(1)      R(1) - R(2)     R(2) - R(3)
        ∅             <a>, <b>           ∅

Ausgabe-Strom: <a, 2>, <b, 2>,  ...
```

Analog zu *IStream* gibt *DStream* alle Tupel, die bis Zeitpunkt *t* je aus der Relation *R* gelöscht wurden, in einem Strom aus.

RStream

RStream (relation stream) angewandt auf Relation *R* enthält ein Element <*s*,*t*> immer dann, wenn das Tupel *s* in *R* zum Zeitpunkt *t* ist:

$$RStream(R) = \bigcup_{t \geq 0}(R(t) \times \{t\})$$

Beispiel

```
        R(0)            R(1)            R(2)            ...
        <a>             <c>             <c>
        <b>                             <d>

Ausgabe-Strom: <a, 0>, <b, 0>, <c, 1>, <c, 2>, <d, 2> ...
```

D. h., im Ausgabe-Strom von *RStream* sind alle Tupel enthalten, die bis zum Zeitpunkt *t* je in der

Relation *R* enthalten waren und sind.

3.2.3.4 Zusammenspiel der drei Operatorklassen

Durch folgendes Beispiel möchten wir erklären, wie man die Operatoren in einer Anfrage verwendet.

Beispiel

Aktienpreisinfos kommen kontinuierlich als Daten-Elemente mit dem gleichen Schema an. Jedes Daten-Element ist ein Tupel <s, t>, wobei s zum Schema S(StockId, Price) gehört und t zur Zeitdomäne T. Die Zeitdomäne T ist in diesem Fall vom SQL-Typ Datetime. Der Zeitstempel t wird von der Datenquelle erzeugt und spezifiziert den Zeitpunkt, zu dem der Preis angeboten wird. Wir interessieren uns nur für solche Aktien, deren Preise höher als 50 sind. Die folgende Anfrage kann in CQL formuliert werden:

```
Select Istream(*)
From S [Range Unbounded]
Where Price > 50
```

Diese Anfrage besteht aus drei Operatoren:

1. Der Strom-zu-Relation-Operator [Range Unbounded], welcher dem Extremfall $Z = \infty$ vom zeitbasierten Fenster (Abschnitt 3.2.3.1) entspricht. Er erzeugt eine Relation, die zum Zeitpunkt *t* alle Aktienpreisinfos enthält, die bis zum *t* angeboten wurden. Diese Relation ist zeitvariierend, da neue Aktionpreisinfos als Daten-Elemente kontinuierlich in Strom ankommen und jedes neue Element zum Fenster [Range Unbounded] gehört.
2. Der Relation-zu-Relation-Operator Filter, der die Tupel der Relation aus 1 nach der Bedingung *Price > 50* filtert. Die Ausgabe-Relation dieses Operators ist auch zeitvariierend, da jedes neue Tupel in Relation aus 1 ein Kandidat für ihn sein kann.
3. Der Relation-zu-Stream-Operator Istream, der alle neu eingefügten Tupel in Relation aus 2 in einen Strom umwandelt.

Das ist ein typisches Beispiel für eine kontinuierliche Anfrage auf Datenstrom. Die Anfrage wird einmal formuliert. Während Daten in Strom kontinuierlich ankommen, wird die Anfrage kontinuierlich ausgewertet. Jedes neue Ergebnis-Tupel wird als Strom-Element gleich nach seiner Erzeugung dem Benutzer oder den Anwendungen weitergegeben.

3.2.3.5 System-Operatoren

Die Funktionen der System-Operatoren ist es hauptsächlich, die CQL-Operatoren von den Problemen der unteren Ebenen zu befreien. Wie nehmen nur ein Beispiel: der Operator *stream-shepherd* für einen Strom S dient als die Quelle für alle Anfrage-Pläne, die auf S zugreifen wollen. Er empfängt die asynchron ankommenden Tupel beispielsweise über das Netzwerk, und

transformiert sie dann in die STREAM-interne Darstellung und schreibt die in die Eingangswarteschlangen der entsprechenden verarbeitenden Operatoren. In der zukünftigen Implementierung wird dieser Operator planmäßig auch Aufgaben wie z. B. Pufferung der Eingabe-Tupel übernehmen, um eine verfälschte Reihenfolge zu korrigieren und „load shedding“ bei Überlast zu kontrollieren.

3.2.3.6 Zusammenfassung STREAM

Die abstrakte Semantik von kontinuierlichen Anfragen basiert auf „black box“-Abbildungen zwischen Strömen und Relationen. Die CQL-Operatoren sind eine Instanzierung der „Black-Boxes“: CQL-Operatoren, die die SQL-Semantik haben, gehören zu der Menge der Relation-zu-Relation-Operatoren. Die von SQL-99 abgeleiteten CQL-Operatoren, welche die Fenster-Spezifikationen unterstützen, bilden Ströme auf Relationen ab. Die drei neuen Operatoren, *IStream, DStream* und *RStream* werden verwendet, um Relationen auf Ströme abzubilden.

3.3 Aurora

In Aurora [1][3] fließen die Daten-Elemente durch einen zyklusfreien gerichteten Graph von ausführenden Operatoren. Auroras Anfrage-Algebra, die „Stream Query Algebra (SQuAl)“, unterstützt sieben primitive Operatoren. Diese sind *Filter, Map, Union, BSort, Aggregate, Join* und *Resample*. Zwischen diesen und den Operatoren der Relationenalgebra gibt es sowohl Ähnlichkeiten als auch Unterschiede. Wir werden uns ihre Semantik in folgenden Abschnitten im Detail betrachten.

3.3.1 SQuAl-Datenmodell

Aurora ist im Wesentlichen ein Strom-zu-Strom-System. D. h. Relationen werden nicht explizit unterstützt. Deshalb unterscheiden wir hier nicht mehr zwischen den Begriffen: *Tupel* und *Element*. Wobei diese Unterscheidung für STREAM sinnvoller ist. In Aurora wird ein Strom als eine “append-only”-Sequenz von Tupeln mit einheitlichem Schema modelliert. Anwendungs-spezifische Datenfelder werden mit $A_1, ..., A_n$ repräsentiert. Jedes Tupel hat zusätzlich noch einen Zeitstempel (*ts*), der die Ankunftszeit des Tupels in Aurora-Netzwerk vermerkt. Dieser Zeitstempel wird vom System erzeugt und für die Berechnung von QoS (Quality of Service) verwendet. Für QoS wird somit der explizite Zeitstempel verwendet. Ein Strom-Typ hat also die Form: $(TS, A_1, ..., A_n)$. Die Strom-Tupel haben die Form: $(ts, v_1, ..., v_n)$. Zur Übersichtlichkeit wird in dieser Ausarbeitung die Angabe von *ts* bei der Darstellung eines Tupels oder *TS* eines Schemas weggelassen.

3.3.2 Reihenfolgeirrelevante (Order-agnostic) Operatoren

Wie wir diskutiert haben, kann die Reihenfolge von Daten-Elementen verfälscht sein. In Aurora unterscheidet man zwischen reihenfolgeirrelevanten (order-agnostic) Operatoren und reihenfol-

gesensitiven (order-sensitive) Operatoren. *Filter, Map* und *Union* können Ausgabe generieren, auch wenn die Daten-Elemente aus der Eingabe in falscher Reihenfolge ankommen. Sie gehören zu den Order-agnostic Operatoren. *Aggregate, Join* und *Resample* können nur richtige Ausgabe erzeugen, wenn die Störungen der Reihenfolge begrenzt sind und diese Begrenzung auch durch eine *Reihenfolge-Spezifikation* (order-specification) angegeben ist. Sie gehören also zu den reihenfolgesensitiven Operatoren, welche die Reihenfolge-Spezifikation als ein besonderes Argument verlangen.

3.3.2.1 Filter

Der Operator *Filter* ist zu dem relationalen Operator *Selection* ähnlich. Der Unterschied ist: *Filter* kann nach mehreren (multiple) Prädikaten filtern und das Tupel entsprechend weiterleiten, je nach dem, welches Prädikat es erfüllt. Nach [3] hat *Filter* die Form:

$Filter\ (P_1, ..., P_m)\ (S)$

Wobei $(P_1, ..., Pm)$ Prädikate auf Tupeln aus dem Eingabe-Strom S sind. Die Ausgabe des *Filters* besteht aus $m + 1$ Strömen: $S_1, ..., Sm_{+1}$, so dass jedes Eingabe-Tupel t in Strom S_i geleitet wird, falls die Bedingung $(i = m + 1\ oder\ P_i(t))$ erfüllt ist und $\forall j < i: (\neg P_j(t))$. D. h. die Tupel, die das Prädikat P_1 erfüllen, werden in Strom S_1 ausgegeben; diejenigen, die das Prädikat P_2 erfüllt, werden in Strom S_2 ausgegeben usw. Alle Tupel, die keine Prädikate erfüllen, werden in den $(m+t)$-ten Strom ausgegeben. Die Ausgabe-Tupel haben das gleiche Schema wie die Eingabe-Tupel. Auch die QoS-Zeitstempel bleiben unverändert.

3.3.2.2 Map

Der Operator *Map* ist zu dem relationalen Operator *Projection* ähnlich. Der Unterschied ist: *Map* kann beliebige Funktionen, einschließlich benutzerdefinierte Funktionen auf den Tupeln ausführen. D. h. *Map* ist eine Generalisierung von *Projection*. Nach [3] kann *Map* so dargestellt werden:

$Map\ (B_1 = F_1, ..., B_m = F_m)\ (S)$

Wobei $B_1, ..., B_m$ Namen der Attribute und $F_1, ..., F_m$ Funktionen auf den Tupeln der Eingabe sind. Für jedes Tupel t aus der Eingabe gibt Map einen Strom in der Form aus:

$(TS = t.TS, B_1 = F_1(t), ..., B_m = F_m(t))$

Der Ausgabe-Strom kann ein zum Eingabe-Strom unterschiedliches Schema haben, d. h.,

$(A_1, ..., A_n) <> (B_1, ..., B_m)$

3.3.2.3 Union (Vereinigung)

Der Operator *Union* ist zu dem Mengenoperator *Vereinigung* ähnlich. *Union* mischt zwei oder mehrere Eingabe-Ströme zu einem einzigen Ausgabe-Strom. Nach [3] ist *Union* in der Form:

$Union\ (S_1, ..., S_n)$

Wobei sind $S_1, ..., S_n$ die Eingabe-Ströme. *Union* kann die Eingabe-Tupel in beliebiger Reihenfolge ausgeben. Eine offensichtliche Strategie ist es jedoch, Eingabe-Tupel nach der Reihenfolge ihrer Ankunft beim Operator auszugeben. Da *Union* alle Eingabe-Tupel ausgibt, ohne ihre Attri-

bute zu ändern, haben alle Eingabeströme $S_1, ..., S_n$ und der Ausgabestrom S_{out} das gleiche Schema.

3.3.3 Reihenfolgesensitive (Order-sensitive) Operatoren

Ein Order-sensitiver Operator kann die Ausführung mit begrenztem Speicherplatz und in begrenztem Zeitraum nur dann garantieren, wenn die Störung der Ankunfts-Reihenfolge spezifiziert ist. D. h. der Operator muss durch die Spezifikation informiert werden, wie er beurteilen kann, ob er die Störung akzeptieren oder ablehnen soll bzw. was er tun soll, wenn er Tupel in falscher Reihenfolge bekommt. Typischerweise kann er die Tupel entweder verwerfen oder mit Hilfe der Pufferung bearbeiten. Dazu dient die Reihenfolge-Spezifikation:
Order (On A[, Slack n, GroupBy $B_1, ..., B_m$ *])*
Wobei $A, B_1, ..., B_m$ Attribute sind, n ist eine natürliche Zahl. Wenn der optionale Anteil der Formel nicht angegeben ist, wird Slack den Default-Wert 0 nehmen und der Eingabe-Strom nicht partitioniert. Die Semantik der Reihenfolge-Spezifikation kann mit einem Beispiel erklärt werden:

Beispiel
Eingabe-Strom-Schema: (A, B)
Eingabe-Strom: ..., (1, x), (2, x), (2, y), (2, y), (5, x), (4, x), (5, y), **(3, x)**, (6, x), ...
Reihenfolge-Spezifikation O = Order (on A, Slack 1, GroupBy B)

Hier sind die Tupel in zwei Gruppen (B = x, oder B = y) getrennt zu betrachten. Das Tupel (3, x) wird vom Operator verworfen. Denn hier ist n = 2, und es gibt schon 2 Tupel, nämlich (5, x) und (4, x), mit Wert des Attributes A größer als 3 in der Gruppe (B = x).

Wäre Slack = 3, dann würde (3, x) akzeptiert werden. Der Operator bräuchte einen größeren Puffer, um die Akzeptanz der Störung zu überprüfen. Wäre Slack = 0, würde auch (4, x) verworfen. Dafür bräuchte der Operator aber keine Pufferung mehr, da jedes Tupel, das in falscher Reihenfolge ankommt, sofort verworfen wird.

Mit der Reihenfolge-Spezifikation kann man also auf Kosten der Verzögerung Genauigkeit gewinnen. Bei einem *Slack* $(n = 0)$ müssen alle Tupel mit einem monoton wachsenden Wert vom Attribut A ankommen. Tupel, die nicht nach dieser Reihenfolge ankommen, werden von dem reihenfolgesensitiven Operator ignoriert. Das macht das Ergebnis einer Berechnung weniger genau. Z. B. bei der Aggregat-Funktion ***Count***, da nicht alle Eingabe-Tupel gezählt sind. Bei einem positiven *Slack* $(n > 0)$ wird die Anforderung an die Reihenfolge der Eingabe-Tupel etwas gelockert: Ein Tupel, das in falscher Reihenfolge bzgl. Attribut A ankommt, wird noch berücksichtigt, solange die Anzahl der Tupel, die mit einem größeren Wert von A vor ihm angekommenen sind, das n nicht überschreitet.

3.3.3.1 BSort

BSort ist ein approximierter Sortierungsoperator in der Form:
BSort (Assuming O) (S), mit O = Order (On A, Slack n, GroupBy B_1, ..., B_m)
Wie im Abschnitt 2 schon diskutiert: wir haben begrenzten Speicherplatz und begrenzte Verarbeitungszeit. Damit ist es unmöglich, alle Tupel von einem unendlichen Strom vollständig zu sortieren. Ein *BSort,* mit *Slack* = *n*, ist eine Puffer-basierte approximierte Sortierung, die equivalent zu einem n-Durchläufe-Bubble-Sort ist. Zu beachten: Das *O* spezifiziert die Reihenfolge nicht für Tupel im Eingabe-Strom, sondern im Ausgabe-Strom. Der Operator verwaltet ein Puffer von *n+1* Tupel für die Eingabe. Sobald der Puffer voll ist, wird ein Tupel im Puffer mit dem kleinsten Wert für *A* aus dem Puffer genommen und in die Ausgabe gegeben. Die Tupel im Ausgabe-Strom behalten ihr originales Schema.

Zeitpunkt	1	2	3	4	5	6	7	8	9	10	...
Eingabe-Strom	2	3	1	2	5	4	8	3	4	6	...
Puffer	2	2	2	2	2	3	4	3	4	5	
	-	3	3	3	3	4	5	5	5	6	...
	-	-	1	2	5	5	8	8	8	8	
Ausgabe-Strom			1	2	2	3	4	3	4	5	...
1-Durchlauf-Bubble-Sort	2	1	2	3	4	5	3	4	6	8	
2-Durchläufe-Bubble-Sort	1	2	2	3	4	3	4	5	6	8	

Abbildung 3-2 Beispielausführung von BSort

Beispiel

In Abbildung 3-2 wird eine Beispielausführung von BSort auf einen Strom dargestellt. Die Tupel im Eingabe-Strom haben die Werte von *A: 2, 3, 1, 2, 5, 4, 8, 3, 4, 6*. Hier im Beispiel ist *Slack = 2*. In diesem Fall verwaltet der Operator einen Puffer von 3 Tupeln. Zum Vergleich sind unten im Bild die Ergebnisse von Bubble-Sort nach dem 1. und 2. Durchlauf dargestellt. Wegen der Verzögerung durch Pufferung haben wir in der Ausgabe zum Zeitpunkt 10 nur noch die ersten acht Tupel. Sie stimmen mit dem Ergebnis vom 2-Durchläufe-Bubble-Sort überein. Wählt man hier einen größeren Slack, z. B. *n = 5*, hat man eine bessere Genauigkeit, die einem 5-Durchläufe-Bubble-Sort entspricht, muss aber eine größere Verzögerung der Antwortzeit in Kauf nehmen.

3.3.3.2 Aggregate

Der Operator *Aggregate* hat die Form:

Aggregate (F, Assuming O, Size s, Advance i[, timeout t]) (S)

S ist der Eingabe-Strom mit Reihenfolge-Spezifikation *O = Order (On A, Slack n, GroupBy* $B_1, ..., B_m$ *)*. *F* ist entweder eine Aggregat-Funktion in SQL-Stil oder eine benutzerdefinierte Funktion, die vom Operator auf ein Fenster über *S* ausgeführt wird. Die Fenstergröße wird mit *s* bzgl. *A* spezifiziert. Die Ganzzahl *i* spezifiziert wie weit das Fenster sich bewegt um neue Tupel zu holen. Die Tupel des Ausgabe-Stroms hat die Form:

$(TS = ts, A = a, B_1 = u_1, ..., B_m = u_m) + + (F (W))$

Ein Tupel in der Ausgabe hat einen Zeitstempel *ts*, einen *A*-Wert *a*, die Werte von $B_1, ..., B_m$, und den Funktionswert *F(W)*. Durch *O* wird der Eingabe-Strom nach den Werten von $B_1, ..., B_m$ partitioniert. Das Fenster *W* enthält alle Tupel in einer Partition, deren *A*-Wert $\in [a, a + s - 1]$ ist. *ts* ist der kleinste Zeitstempel im Fenster. Das Zeichen „+ +“ ist die Konkatennation von Tupeln. Denn neben Attribut-Werte kann die Funktion *F* auch ein ganzes Tupel liefern. Die Semantik des Operators wird mit dem folgenden Beispiel erklärt:

Beispiel

Angenommen die Aktienpreise verändern sich mit der Zeit und die Börseninfos in einem Markt kommen in einem Strom beim System an. Der Eingabe-Strom hat das Schema *(StockId, Time, Price)*. Man möchte aus dem Strom die stündlichen durchnittlichen Preise für jede Aktie berechnen. Diese Anfrage kann mit dem Aggregate-Operator formuliert werden (der Eingabe-Strom sei *S*):

Aggregate(Avg(Price), Assuming O, Size 1 hour, Advance 1 hour)(S), mit

O = Order(On Time, GroupBy StockId)

Die Ausführung dieser Operation auf 11 Tupeln wird in Abbildung 3-3 dargestellt. Links befinden sich die Tupel des Eingabe-Stroms. Die Ganzzahl von 1 bis 11 repräsentieren die Ankunftszeiten. In der Mitte sind die Fenster dargestellt. Man sieht: der Eingabe-Strom ist nach den Werten von *Stock-Id* (spezifiziert mit *GroupBy Sid*) in drei Partitionen aufgeteilt. Die Fenster haben eine Größe von einer Stunde und bewegen sich jeweils um eine Stunde. Rechts sind die Tupel vom Ausgabe-Strom dargestellt, dessen Schema *(StockId, Time, AvgPrice)* ist.

Im Beispiel hat *Slack* den Default-Wert 0. Das Reihenfolgeproblem könnte behandelt, falls *Slack* einen nicht negativen Wert hat. Z. B. nehmen wir an, dass das 8. Tupel verspätet wäre und *Slack = 1*. Die Ankunftsreihenfolge wäre: 1, 2, 3, 4, 5, 6, 7, 9, 10, 11, 8. Das verspätete Tupel wird immer noch im Fenster *IBM: 1-1:59* mitberücksichtigt. Denn in der Partition, definiert durch *StcokId = IBM,* gibt es nur ein Tupel mit *Time*-Attribut größer als *1:45*.

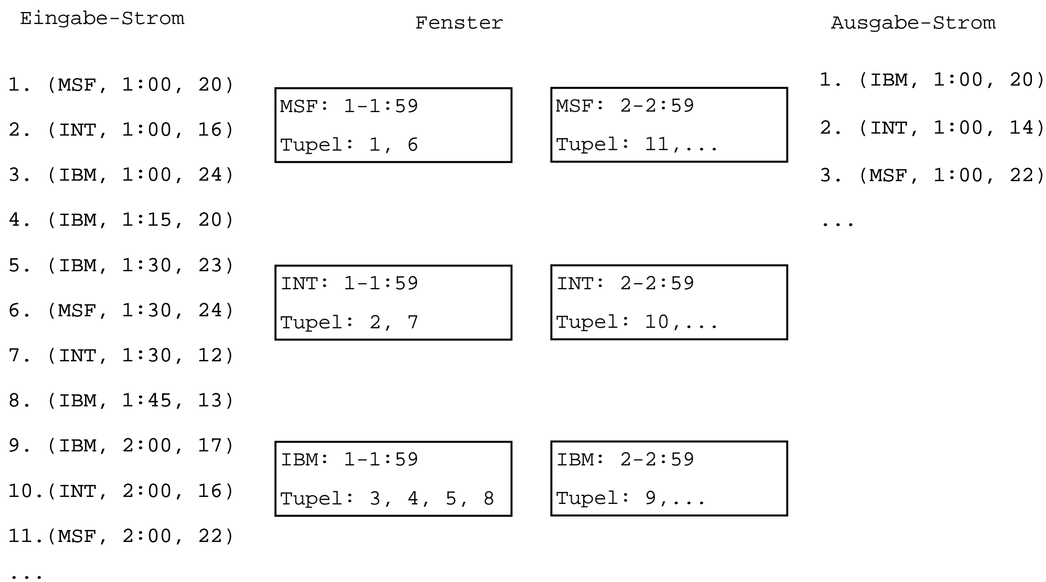

Abbildung 3-3 Beispielausführung von Aggregate

Was passiert wenn das 8. Tupel noch länger verspätet ist oder sogar verloren wurde? Sollte die Berechnung für das Fenster *IBM: 1-1.59* angehalten werden, um auf das 8. Tupel zu warten? Dabei ergibt sich ein weiteres Problem: Blockierung durch die Eingabe. Der optionale Parameter *Timeout t* in der Formel behandelt dieses Problem. Wenn *t* angegeben ist, das System notiert zum Beginn der Berechnung für jedes Fenster einen Zeitstempel. Nach *t* Zeiteinheiten wird die Berechnung für ein Fenster zwangsweise unterbrochen. Das bisherige Ergebnis wird ausgegeben. Tupel, die eigentlich zu diesem Fenster gehören aber später ankommen, werden für die Berechnung in diesem Fenster nicht mehr berücksichtigt.

3.3.3.3 Join (Verbund)

Der *Join* Operator in SQuAl simuliert den *Band-Join* der Relationenalgebra. *Join* ist ein Binär-Operator in der Form:
Join (P, Size s, Left Assuming O_1*, Right Assuming* O_2*)* (S_1, S_2)
Wobei *P* das Prädikat für Tupel in Eingabe-Strömen S_1 und S_2 ist, ähnlich des Prädikats im *Theta-Join* der Relationenalgebra. Der Integer *s* schränkt die in der Verbund-Operation beteiligenden Tupel ein. O_1 und O_2 sind die Reihenfolge-Spezifikationen bzgl. der Attribute *A* für S_1 bzw. *B* für S_2. Ein Tupel in der Ausgabe besteht aus der Kombination zweier Tupel: $t \in S_1$ und $u \in S_2$, falls $| t.A - u.B |$ kleiner gleich *s* und *P(t, u)* wahr ist. Jedes Ausgabe-Tupel eines Operators wird mit dem Zeitstempel des ältesten Tupels, das in der Verbund-Operation beteiligt, gestempelt.

Beispiel

Aktienpreisinfos aus den zwei Märkten seien durch zwei Eingabe-Ströme *X, Y* mit dem gleichen Schema *(StockId, Time, Price)* modelliert. Man möchte die Aktien-Paare finden, deren Preise innerhalb von 10 Minuten in zwei verschiedenen Märkten gleich sind. Die Anfrage kann mit dem Join-Operator fol-

gendermaßen formuliert werden:

Join(P, Size 10 Min, Assuming Left O, Assuming Right O) (X, Y)

Wobei *O = Order(On Time)* und *P(x, y) <=> x.Price = y.Price*

Die Ausführung der Join-Operation ist in Abbildung 3-4 dargestellt:

X	Y	Ausgabe-Strom von Join
01. (IBM, 2:00, 3)	01. (SMS, 1:55, 3)	01. (IBM, 2:00, 3, SMS, 1:55, 3)
02. (MSF, 2:00, 1)	02. (SAP, 1:55, 4)	02. (IBM, 2:00, 3, SMS, 2:05, 3)
03. (INT, 2:00, 1)	03. (BMW, 1:55, 5)	03. (IBM, 2:05, 4, SAP, 1:55, 4)
04. (IBM, 2:05, 4)	04. (DML, 1:55, 6)	04. (IBM, 2:05, 4, SAP, 2:05, 4)
05. (MSF, 2:05, 2)	05. (SMS, 2:05, 3)	05. (IBM, 2:05, 4, SMS, 2:15, 4)
06. (INT, 2:05, 1)	06. (SAP, 2:05, 4)	06. (IBM, 2:05, 4, SAP, 2:15, 4)
07. (IBM, 2:10, 4)	07. (BMW, 2:05, 5)	07. (IBM, 2:10, 4, SAP, 2:05, 4)
08. (MSF, 2:10, 3)	08. (DML, 2:05, 6)	08. (IBM, 2:10, 4, SMS, 2:15, 4)
09. (INT, 2:10, 1)	09. (SMS, 2:15, 4)	09. (IBM, 2:10, 4, SAP, 2:15, 4)
10. (IBM, 2:15, 2)	10. (SAP, 2:15, 4)	10. (MSF, 2:10, 3, SMS, 2:05, 3)
11. (MSF, 2:15, 4)	11. (BMW, 2:15, 5)	11. (MSF, 2:15, 4, SAP, 2:05, 4)
12. (INT, 2:15, 1)	12. (DML, 2:15, 6)	12. (MSF, 2:15, 4, SMS, 2:15, 4)
...	...	13. (MSF, 2:15, 4, SAP, 2:15, 4)
		...

Abbildung 3-4 Beispielausführung von Join

3.3.3.4 Resample (Interpolation)

Der Operator ***Resample*** interpoliert Tupel in einem Strom nach einem geordneten Attribut von einem anderen Strom. Nach [3] hat ***Resample*** die Form:

Resample (F, Size s, Left Assuming O1, Right Assuming O2) (S1, S2)

Wobei F ist eine Funktion (eine Aggregat-Funktion oder benutzerdefinierte Funktion) auf Tupeln in Eingabe-Strom S_1. Der Integer s ist ähnlich wie bei ***Join***. O_1 spezifiziert die Reihenfolge von Tupeln in S_1 bzgl. Attribut A. O_2 spezifiziert die Reihenfolge von Tupeln in S_2 bzgl. Attribut B. Für jedes Tupel t aus S_1, besteht das Schema der Ausgabe-Tupel aus drei Teilen:

1. Attribute $B_1, ..., B_m$ (durch O_2 spezifiziert) von S_2;
2. Attribut A;
3. ein mit F berechnetes Attribut, F nimmt für jedes Tupel t aus S_1 solche Tupel aus S_2 als Eingabe: $\{ u \in S_2 \mid u$ erfüllt O_2 und $| t.A - u.B | \leq s \}$.

Anders gesagt: ***Resample*** interpoliert Tupel in S_2 nach einem geordneten Attribut A von S_1.

Beispiel

Man möchte die Aktienpreise zu jeder Viertelstunde analysieren (Preise zu vier Zeitpunkten jeder Stunde). Für jede Aktie möchten wir den durchnittlichen Preis innerhalb von 20 Minuten (10 Minuten vor dem Zeitpunkt der Analyse und 10 Minuten danach) wissen. Die Aktienpreise kommen in einem Strom

X(StockId, Time, Price) an. Zusätzlich haben wir einen Eingabe-Strom *Z(Time_to_Analyse)*, der die vier Zeitpunkte in jeder Stunde simuliert. Bei diesem Anwendungsszenario kann man den Resample-Operator verwenden:
Resample (F, Size 10 Minutes, Left Assuming O_1, Right Assuming O_2) (Z, X)
mit F = Avg (Price), O_1 = Order (on Time_to_Analyse), O_2 = Order (on Time, GroupBy StockId)

Die Beispielausführung dieser Operation ist in Abbildung 3-5 dargestellt. In der mittleren Abbildung 3-5 sind die Eingabe-Tupel dargestellt. Rechts ist der Ausgabe-Strom, welcher vom Resample-Operator generiert wird. Zu beachten: die Tupel werden nach der Reihenfolge ausgegeben, in der die Berechnung abschließt. Sobald der Operator das 3. Tupel (INT, 2:11, 1) aus *X* sieht, weißt er, dass die Berechnung des Fensters $\{(\mathrm{INT},t,p)\in X \mid t\in[1{:}50,2{:}10]\}$ abgeschlossen werden kann, denn $2{:}11\notin[1{:}50,2{:}10]$.

Z(Time_to_Analyse)	X(StockId, Time, Price)	Ausgabe-Strom von Resample
1. (2:00)	1. (IBM, 2:08, 3)	1. (INT, 2:00, 2)
2. (2:15)	2. (INT, 2:03, 2)	2. (IBM, 2:00, 3)
3. (2:30)	3. (INT, 2:11, 1)	3. (IBM, 2:15, 2.33)
4. (2:45)	4. (IBM, 2:12, 1)	4. (MSF, 2:00, 1)
5. (3:00)	5. (IBM, 2:21, 3)	5. (INT, 2:15, 2.5)
...	6. (INT, 2:14, 4)	6. (MSF, 2:15, 2.33)
	7. (MSF, 2:10, 1)	7. (MSF, 2:30, 4)
	8. (IBM, 2:34, 4)	...
	9. (MSF, 2:19, 2)	
	10. (INT, 2:28, 3)	
	11. (MSF, 2:25, 4)	
	12. (MSF, 2:41, 1)	
	...	

Abbildung 3-5 Beispielausführung von Resample

3.3.4 Ein komplizierteres Beispiel

Zum Schluss der Vorstellung der SQuAl betrachten wir noch ein Beispiel, bei dem mehrere Operatoren kombiniert verwendet werden:

Beispiel
Generiere eine Nachricht (Warnung), wenn die Preise von mehr als m Aktien gleichzeitig über eine Grenze k gehen. In diesem Beispiel haben wir das gleiche Schema für Aktienpreisinfo: *S (StockId, Time, Preis)*
Um die gewünschte Nachricht (in Form eines Ausgabe-Stroms) zu bekommen, können wir Schritt für Schritt den Eingabe-Strom S verarbeiten:
1. Alle Tupel, die einen Preis höher als k haben, aus S auswählen:
   Filter(Price $\geq$ k)

2. Eine Aggregate-Funktion(Count) auf den Strom aus 1 ausführen: für jeden Zeitpunkt zählen wir die Aktien, deren Preise höher als k sind.
   Aggregate(CNT, Assuming O, Size 1, Advance 1),
   mit O = Order(On Time, Slack n)
3. Den Strom aus 2 weiter filtern: nur solche Zeitpunkte gehen in die Ausgabe ein, zu denen mehr als m Aktien teuerer als k sind.
   Filter(CNT ≥ m)

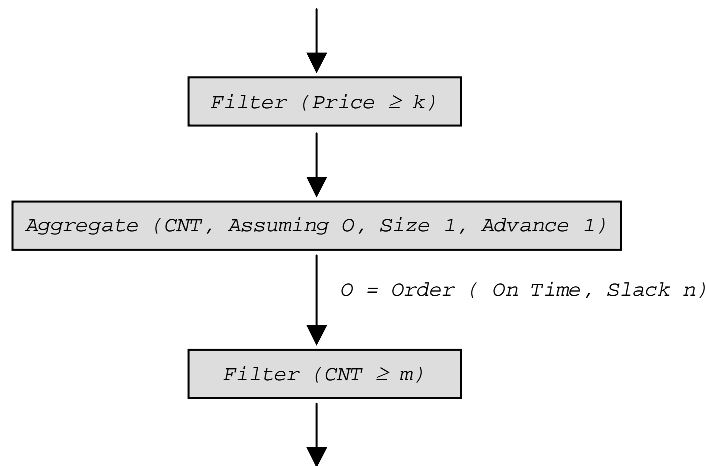

Abbildung 3-6 Ein komplizierteres Beispiel

Die Abbildung 3-6 zeigt die drei Schritte in Auroras „Box und Arrow"-Stil. Eine Beispielausführung, mit der Annahme $k = 30$, $m = 3$ und $Slack = 1$, ist in Abbildung 3-7 dargestellt:

S(StockId, Time, Price)		(StockId, Time, Price)		(Time, Cnt)		(Time, Cnt)
(1, 1, 34)		(1, 1, 34)		(1, 2)		(2, 5)
(1, 2, 38)		(1, 2, 38)		(2, 5)		(3, 3)
(2, 1, 24)	Filter	(3, 1, 35)	Aggregate	(3, 3)	Filter	...
(3, 1, 35)	=>	(3, 2, 38)	=>	...	=>	
(3, 2, 38)		(2, 2, 31)				
(3, 3, 18)		(4, 2, 36)				
(4, 1, 21)		(4, 3, 30)				
(5, 1, 20)		(5, 2, 31)				
(2, 2, 31)		(2, 3, 41)				
(4, 2, 36)		(5, 3, 31)				
(4, 3, 30)		...				
(5, 2, 31)						
(1, 3, 28)						
(2, 3, 41)						
(5, 3, 31)						
...						

Abbildung 3-7 Beispielausführung von mehreren SQuAl-Operatoren

An diesem Beispiel sieht man, dass die Ausgabe-Tupel von Aggregate nicht sofort erzeugt werden, sobald m Aktien erkannt sind, die teuerer als 30 sind. Z. B., bei der Berechnung des Fens-

ters mit `Time=1`, mit dem 3. Tupel `(3, 1, 35)` ist uns schon bekannt, dass es zwei Aktien gibt, die zu Zeitpunkt `1` teuer als `30` sind. Aber erst mit dem 5. Tupel `(2, 2, 31)` können wir dann feststellen, dass in dieses Fenster kein Tupel mehr kommt und das Tupel `(1, 2)` ausgegeben werden kann.

3.4 Vergleich Aurora und STREAM

STREAM und Aurora sind beide Strom-orientiert. Sie haben sowohl Gemeinsamkeiten als auch Unterschiede. Beide Systeme unterstützen die Auswertung kontinuierlicher Anfragen auf Datenströmen. Die Strom-Daten-Elemente sind in beiden Systemen ähnlich modelliert: ein Strom hat immer eine feste Attributmenge, d. h., jedes Element in einem Strom hat das gleiche Schema. Beide Systeme verwenden expliziten Zeitstempel, wenn es um die Semantik einer Operation geht.

STREAM orientiert sich nach der Auswertung von kontinuierlichen Anfragen auf Datenströmen und Relationen. Datenstrom und Relation sind zwei abstrakte Datentypen in STREAM. Aurora ist grundsätzlich ein Strom-zu-Strom-System, wobei gespeicherte Relationen nicht explizit unterstützt werden.

Die Benutzerschnittstellen in beiden Systemen sind auch verschieden. STREAM hat eine SQL-basierte Anfragesprache mit Strom-Erweiterungen. Aurora verwendet dagegen den „box and flow“-Ansatz, wobei „boxes“ die Operatoren repräsentieren und „flows“ die Ströme sind.

Im Abschnitt 2 haben wir über das Reihenfolgeproblem diskutiert, also das Problem der verfälschten Ankunftsreihenfolge der Tupel. In STREAM ist die Behandlung dieses Problems Aufgabe der System-Operatoren. In Aurora wird das Problem mit Hilfe der Reihenfolgespezifikation behandelt. Dadurch können die reihenfolgesensitiven Operatoren vernünftige Maßnahme ergreifen, wenn Tupel in falsche Reihenfolge ankommen. Da die Reihenfolge-Spezifikation auf jedem Attribut definiert werden können, unterstützt Aurora Fenster auf jedem Attribut. Dabei ist ein Fenster (auch ein tupelbasiertes Fenster) im STREAM immer von einem Zeitdomäne-Attribut abhängig.

Der Aggregate-Operator in Aurora unterstützt benutzerdefinierte Funktionen und ist deswegen sehr ausdrucksmächtig. Er kann auch optionale Parameter wie *timeout* enthalten, um eine nicht-blockierende Operation zu garantieren. Dies hat aber eine nicht-deterministische Semantik zufolge.

4 Über die Implementierung des Join-Operators

Die Implementierung des Join-Operators ist für die Realisierung eines DSMSs besonders interessant. Zum einen, da *Join, Differenz* und *Durchschnitt* sich auf sehr ähnliche Weise implemen-

tieren lassen. Während der Join-Operator Attribute zweier Tupel verglicht, vergleichen *Differenz* und *Durchschnitt* das ganze Tupel. Deshalb bezeichnet man sie auch als *binäre Zuordnungsoperatoren* [11]. Zum anderen, ist der *Join* eine sehr ressourcenaufwändige Operation, weil potenziell alle Tupel der Eingabemengen zu durchlaufen sind. Dies ist im Kontext der Strom-Verarbeitung generell unmöglich oder sogar unerwünscht ist.

Klassische Algorithmen für die Join-Implementierung (*Nested-Loop-Join, Hash-Join, Sort-Merge-Join* und viele ihrer Varianten) stammen aus DBMS und lassen sich nicht direkt auf unendliche Datenströme übertragen. Auch wenn die Fenstertechnik eingesetzt wird, sind die klassischen Algorithmen für die Join-Operation auf Datenströmen problematisch, denn das Fenster sehr groß sein und gar nicht zum Hauptspeicher passen kann. Die Varianten der klassischen Algorithmen sind meistens entweder *hash-basiert* oder *sortierbasiert*. Der *Nested-Loop-Join* ist für beliebige Join-Prädikate geeignet und lässt sich einfach implementieren. Aber er ist in der Praxis wegen geringerer Effizienz von nicht so großer Bedeutung.

Hash-basierte Joins sind auf den *Equi-Joins* und *Natural-Joins* spezialisiert, weil sie auf einem Gleichheitsprädikat eingeschränkt sind. Zu hash-basierten Joins gehört der *Symmetrische Hash-Join* (SHJ), der für eine Verarbeitung nach dem Prinzip des „Pipelining“ entworfen wurde. SHJ ist ein Hauptspeicher-Algorithmus und deswegen für sehr große Datenmengen (was im Strom-Kontext wichtig ist) nicht geeignet. Interessanter ist eine Erweiterung von SHJ: der *XJoin* [8]. Neben der Fähigkeit von „Pipelined“-Verarbeitung ist der *XJoin* nicht mehr auf dem Hauptspeicher eingeschränkt. Der *XJoin* behandelt ein spezifisches Problem: die negative Wirkung von unstabilen Eingaben. Z. B., würde ein SHJ blockieren, wenn die Datensätze von einem unstabilen Netzwerk ankommen und eine oder beide Eingaben vorübergehend angehalten sind. Der *XJoin* kann in dieser Situation dennoch kontinuierlich Ergebnis-Tupel erzeugen.

Sortierbasierte Joins beschränken sich nicht auf den *Equi-Joins* und *Natural-Joins*. Sie haben aber einen Nachteil: generell muss die ganze Eingabe in der Sortierphase eingelesen werden, bevor die ersten Ergebnis-Tupel in der Join-Phase erzeugt werden können. Die meisten Sortierbasierten Joins werden deswegen auch als blockierend bezeichnet. Unter sortierbasierten Joins ist der *Sort-Merge-Join* im DBMS am wichtigsten. Für Strom-Verarbeitung ist jedoch seine nicht-blockierende Variante interessanter, der *Progressive Merge Join* (PMJ)[9]. PMJ kann schon in der Sortierphase Ergebnis-Tupel erzeugen. Mit Fenstersemantik kann dieser Algorithmus nach [10] auf Strom-Verarbeitung adaptiert werden.

5 Zusammenfassung und Konklusion

Wir haben in dieser Ausarbeitung zuerst die Motivation von DSMS und die Unterschiede zwischen Datenstrommodell und Relationenmodell vorgestellt. Im Abschnitt 2 wurden einige Probleme diskutiert, die für die Semantik der Operatoren auf Datenströmen relevant sind. Eine Übersicht und die Klassifikationen der Strom-Operatoren befinden sich am Anfang des 3. Abschnittes. In weiteren Teilen von Abschnitt 3 wurde die Semantik der typischen Strom-Operatoren vorge-

stellt, dabei wurden STREAM- und Aurora-Projekte als Beispiele genommen. Denn die zwei DSMS sind typisch: Aurora ist explizit Strom-orientiert und hat eine vollständige Menge von Strom-Operatoren. STREAM unterstützt sowohl Ströme als auch Relationen. Die Abbildungen zwischen Strömen und Relationen sind für unsere Betrachtung interessant. Deswegen sind die Fenster-Operatoren und die drei Relation-zu-Strom Operatoren vorgestellt. Da Join-Operator für die Realisierung eines DSMSs sehr wichtig ist, wird einen kurzen Überblick über die Implementierung von Join-Operator im Abschnitt 4 gegeben.

Die Auswertung von Anfragen auf Datenströmen bedarf neuer Überlegungen in der Semantik der Operatoren, auch wenn ein DSMS eine SQL-basierte Anfragesprache verwendet. Denn die Strom-Anwendungen haben besondere Anforderungen an die Strom-Systeme und es gibt fundamentale Unterschiede zwischen dem Datenstrommodell und dem Relationenmodell.

6 Literatur

[1] **Monitoring Streams – A New Class of Data Management Applications.** Don Carney, Ugur Cetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Greg Seidman, Michael Stonebraker, Nesime Tatbul and Stan Zdonik, Proceedings of Very Large Databases (VLDB), Hong Kong, China, August 2002

[2] **Models and Issues in Data Stream Systems.** B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom, Proceedings of the Twenty-First ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, Madison, Wisconsin, June 2002

[3] **Aurora: A New Model and Architecture for Data Stream Management.** Daniel Abadi, Don Carney, Ugur Cetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik, VLDB Journal, August, 2003.

[4] **Continuous Queries over Data Streams.** S. Babu and J. Widom, In SIGMOD Record, September 2001

[5] **An Abstract Semantics and Concrete Language for Continuous Queries over Streams and Relations.** A. Arasu, S. Babu and J. Widom, Technical Report, November 2002

[6] **The CQL Continuous Query Language: Semantic Foundations and Query Execution.** A. Arasu, S. Babu and J. Widom, Technical Report, Oct. 2003

[7] **A Temporal Foundation for Continuous Queries over Data Streams.** Jürgen Krämer, Bernhard Seeger, Technical Report, July 2004.

[8] **XJoin: Getting Fast Answers From Slow and Bursty Networks.** Tolga Urhan, Michael J. Franklin, Technical Report CS-TR-3994, University of Maryland, February 1999.

[9] **Progressive Merge Join: A Generic and Non-Blocking Sort-Based Join Algorithm.** Jens-Peter Dittrich, Bernhard Seeger, David Scot Taylor, Peter Widmayer, Proc. VLDB 2002: 299-310

[10] **Sortierbasierte Joins über Datenströmen.** Michael Cammert, Christoph Heinz, Jürgen Krämer, Bernhard Seeger, Technical Report No. 42, Department of Mathematics and Computer Science, University of Marburg, 2004

[11] **Datenbanksysteme.** A. Kemper, A. Eickler, Oldenburg Verlag, München, 2004