Seminar Data Streams

Thema: Anfragesprachen für Datenströme

Wintersemester 2004/2005

Sebastian Glöckner

Seminar: Data Streams

WS 2004/05

Sebastian Glöckner

Thema: Anfragesprachen

Inhaltsgliederung:

1. **Einleitung**
2. **Grundlagen Datenstrom-orientierter Anfragesprachen**
3. **Vorstellung einiger Anfragesprachen**
    a) **CQL**
    b) **Hancock**
    c) **Framework Aurora**
    d) **GSQL**
    e) **StreaQuel**
4. **Vergleich der vorgestellten Anfragesprachen**
5. **Vergleich mit herkömmlichen Anfragesprachen und deren Verarbeitung**
6. **Zusammenfassung/Konklusion**
7. **Quellenangaben**

Zusammenfassung:

In dieser Ausarbeitung geht es um Anfragesprachen für Datenströme. Zuerst geht es um die Besonderheiten und Einordnungsmöglichkeiten datenstromorientierter Anfragesprachen. Daraufhin werden einige Anfragesprachen vorgestellt (Hancock, StreaQuel, CQL und GSQL). Zunächst erfolgt eine Vorstellung von Struktur und Funktionsweise der Anfragesprache um diese dann anschließend anhand von Beispielen zu verdeutlichen.

Weitere, interessante Ansätze beinhaltet das Framework Aurora (entwickelt von den Universitäten Brown, Brandeis und MIT). Insbesondere seine Anfragesprache und die Möglichkeit Quality of Service (QoS) zu spezifizieren verdienen hier Aufmerksamkeit. Darauf folgt dann ein vergleichender Überblick über die vorgestellten Anfragesprachen. Auch ein kurzer Vergleich der neuen Ansätze mit denen konventioneller Datenbanksysteme (und SQL) ist angebracht. Abschließend gibt es eine Zusammenfassung und die wichtigsten Aspekte werden noch einmal dargestellt.

1. Einleitung

Mit dem Entstehen neuer Anwendungen wie Sensor-Netzwerke, Positionsüberwachung und der Zunahme des Datenvolumens bei bekannten Anwendungen, wie z.B. der Netzwerküberwachung, ergeben sich Anforderungen an Datenverwaltungssysteme, die von den herkömmlichen Vertretern ihrer Art nicht mehr, oder zumindest nicht in zufriedenstellender Art und Weise, bewältigt werden können. Die von den neuen Anwendungen bereitgestellten Datenströme erfordern in den meisten Fällen eine andere Weise der Verarbeitung und evtl. auch der Speicherung. Die Anfragen werden um Zeitbezüge und Datenströme erweitert. Verbreitete SQL-basierte Systeme liefern dafür noch keine brauchbaren Mechanismen, erst mit der Verbreitung von SQL2003 könnte sich dies ändern. Traditionelle DBMS bieten nicht genug Performanz und setzen zudem ohnehin andere Schwerpunkte. Die bisherigen Lösungsansätze für diese Probleme stammen aus Industrie und Forschung und umfassen eigenständige Datenverwaltungssysteme, die sogenannten *Data Stream Management Systeme* (DSMS). Diese operieren mit neuen, bzw. speziell angepassten Anfragesprachen, auf die im Folgenden eingegangen werden wird.

2. Grundlagen datenstromorientierter Anfragesprachen [1]

Datenströme (DS) unterscheiden sich in wesentlichen Aspekten von den Daten konventioneller Datenbanksysteme. Dort werden die Daten permanent gespeichert und unterliegen in der Regel keinen großen Änderungen. Bei den auf diesen Daten spezifizierten Anfragen wird üblicherweise großen Wert darauf gelegt, dass die Ergebnisse exakt sind, wobei die Verarbeitungsgeschwindigkeit nicht die höchste Priorität setzt. Bei Datenströmen hingegen kommt es darauf an, die Daten möglichst schnell – am besten in Echtzeit – zu verarbeiten. Oftmals ist es auch nicht nötig oder überhaupt möglich, alle Daten permanent zu speichern, sondern es genügt ein komprimiertes bzw. aggregiertes Anfrageergebnis. Außerdem ist es in manchen Anwendungen durchaus vertretbar, wenn die Anfrageergebnisse nicht exakt sind, sondern sie die realen Werte nur approximieren (z. B. durch Sampling).

Ein Datenstrom ist potenziell unendlich, d. h. es kann eine quantitativ unbegrenzte Menge von Tupeln, zeitlich unbegrenzt, eintreffen. Dies stellt ein großes Problem bei der Anfragestellung und - verarbeitung dar. Entweder wird daraus ein potenziell unendlicher Ausgabedatenstrom erzeugt, oder man begrenzt die zu bearbeitende Datenmenge zeitlich oder quantitativ mit Hilfe von *Sliding Windows*. Anfragen können vor Laufbeginn im System vorhanden sein oder zur Laufzeit eingebracht werden. Sie können außerdem entweder eine kontinuierliche Abarbeitung verlangen (und ständig neue Ausgaben generieren) oder sie können einmalig gestellt und verarbeitet werden.

Neu ist auch, dass nun Ströme mit herkömmlichen Relationen verknüpft werden können. Dabei stellt sich die Frage, was nun der Ergebnistyp einer solchen Verknüpfung sein soll: ein Strom oder eine Relation.

Ein weiterer Aspekt ist jener der Zeit. Die Daten eines Stroms unterliegen einer zeitlichen Ordnung, diese ist in der Regel diskret (z.B. Sequenznummer, Uhrzeit) und muss in der Anfrageverarbeitung berücksichtigt werden. Dabei ist es je nach Anwendung und Architektur möglich, dass die Ankunftsreihenfolge der Datensätze nicht der tatsächlichen zeitlichen Ordnung entsprechen (implizite und explizite Zeitmarken).
Da die Daten alle einzeln in Reihe ankommen (wie in einer Pipeline) bereitet auch der Einsatz blockierender Operatoren Probleme. Es handelt es sich hierbei um Operatoren, die erst dann eine Ausgabe erzeugen, wenn sie alle Eingabedaten verarbeitet (oder erhalten) haben. Dies sind zum Beispiel Sortier-, Gruppier- und Aggregatoperatoren. Bei einem unendlichen Eingabestrom ist es aber unmöglich, vor Ausführung der Operation auf alle Eingaben zu warten, da die Anfrage nie vollständig ausgeführt werden könnte.
Eine geeignete Anfragesprache muss (möglichst) all diesen Anforderungen gerecht werden. Bisher haben sich dabei verschiedene Arten von Anfragesprachen herausgebildet:

- deklarative Anfragesprachen beschreiben eine Anfrage ähnlich wie in SQL, bieten aber spezielle Operatoren und eine eigene Syntax. Vertreter sind CQL, GSQL und StreaQuel
- prozedurale Anfragesprachen stellen eine Anfrage aus Datenströmen und Operatoren zusammen und ähneln einer Programmiersprache (z.B. C). Ein Vertreter ist Hancock

Es gibt jedoch auch Anfragesprachen, die sich nicht eindeutig klassifizieren lassen. Etwa finden sich in Aurora Aspekte, die je nach Betrachtungsweise auf die eine oder die andere Klasse hindeuten. Vom Standpunkt des Benutzers aus, der eine Anfrage erstellt, ist der Ansatz eher als deklarativ zu sehen.

3. Vorstellung einiger Anfragesprachen

a) CQL (Continuous Query Language) [3]

Die Sprache CQL stammt aus dem STREAM-Projekt und stellt im wesentlichen eine stromorientierte Erweiterung von SQL dar (SQL2003).

Abstrakte Semantik:

Im Modell von CQL existieren neben Relationen auch Ströme. Die Ströme besitzen, wie die Relationen, ein feste Menge von Attributen. Desweiteren existiert eine Zeit-Domäne T, mit diskreten und geordneten Elementen, welche nicht unbedingt der Uhrzeit entstammen müssen. Diese dient dazu, den Daten eines ankommenden Stroms sowie den Änderungen von Relationen Zeitwerte zuzuordnen.
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 ∈ T der Zeitstempel des Elements.

Der Zeitstempel gehört nicht zur Menge der Attribute. Es können mehrere Elemente mit dem gleichen Zeitstempel existieren.

Es gibt zwei Arten von Strömen: Basisströme, dies 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.

Eine Relation wird wie folgt definiert (Einführung eines Zeitbezugs):

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 bestimmt eine ungeordnete Multimenge von Tupeln zu jedem Zeitpunkt t ∈ T, geschrieben R(t). Auch hier gibt es Basis-Relationen und abgeleitete Relationen.

Es ist zu beachten, dass es nicht unbedingt nötig ist, beide Konzepte (Ströme und Relationen) zu unterstützen. Eines würde vom Standpunkt der Ausdrucksmächtigkeit vollkommen genügen, schließlich verzichten andere Ansätze auf das eine oder das andere Konzept s. u. . Jedoch mag die eine Darstellung in gewissen Anwendungen dem intuitiven Realweltmodell eher entsprechen.

Es gibt drei Klassen von Operatoren, *Strom-zu-Relation*-Operatoren, *Relation-zu-Relation*-Operatoren sowie *Relation-zu-Strom*-Operatoren. Die Bezeichnung bezieht sich dabei auf den Eingabe- und den Ausgabetyp der Operatoren. Es existiert kein Strom-zu-Strom-Operator, dieser kann aber mit den anderen drei Operator-Typen simuliert werden. Jedoch verursacht diese Lösung einen unnötigen Overhead bei Anfragen, bei denen eigentlich nur auf Strömen operiert werden soll. Vermutlich wurde dieser Ansatz deshalb gewählt, weil somit bei der Implementierung der Operatoren weitgehend auf bereits vorhandene SQL-Varianten zurückgegriffen werden konnte und die Semantik der Anfragen eher dem aus SQL bekannten Modell entsprach (s. u.). Für alle diese Operatoren gilt, dass sie eine Ausgabe erzeugen, die sich auf den aktuellen Zeitpunkt t bezieht und sie dafür alle Elemente mit Zeitstempeln kleiner t kennen müssen.

Konkrete Semantik:

Um dem weit verbreiteten und wohl bekannten Ansatz von Relationen so nahe wie möglich zu bleiben, und somit all seine Vorteile nutzen zu können (Semantik, Operatoren, Algebra, SQL etc.), bietet CQL eine große Menge von Relation-zu-Relation-Operatoren, die den größten Teil der Datenmanipulation vornehmen. Strom-zu-Relation- und Relation-zu-Strom-Operatoren sind lediglich in geringer Zahl vorhanden um eine Transformation zwischen beiden Typen zu ermöglichen. Ein Nachteil dieser Entscheidung ist die Tatsache, dass selbst eine einfache Filterfunktion auf einem Strom drei Operatoren benötigt (Umwandeln des Stroms in eine Relation, Filtern und Rückwandeln der Relation in einen Strom).

Strom-zu-Relation Operatoren basieren auf dem Sliding-Window-Konzept, die Syntax wurde SQL2003 entlehnt. Ein Sliding Window auf einem Strom ist ein Fenster, dass zu jedem Zeitpunkt einen historischen Schnappschuss eines endlichen Teils des Stroms beinhaltet.

Es werden Zeit-basierte Sliding-Windows, Tupel-basierte Sliding-Windows und partitionierte Sliding-Windows unterschieden. Zeit-basierte Sliding-Windows verwenden ein Zeitintervall T als Eingabeparameter und werden folgendermaßen spezifiziert: *[Range T].* Die Ausgaberelation R eines Eingabestromes S mit dem Operator *[Range T]* ist wie folgt definiert:

$R(t) = \{ s \mid \langle s,t' \rangle \in S \wedge (t' \leq t) \wedge (t' \geq \max\{t - T,0\}) \}$

Für T = 0 existiert die Syntax *[Now],* für T = ∞ *[Range unbounded].*

Tupel-basierte Sliding-Windows werden spezifiziert mit *[Rows N]* wobei *N* ein positiver, ganzzahliger Wert ist. Es werden von S die N Tupel ausgewählt, die den größten Zeitstempel haben. Haben alle (mehrere) Tupel des Stroms (der mehr als N Tupel hat) den gleichen Zeitstempel, so werden die Tupel zufällig ausgewählt. Für den Sonderfall N = ∞ gibt es die Syntax *[Rows Unbounded].*

Partitionierte Sliding-Windows werden wie folgt spezifiziert: *[Partition by A1,A2,...,Ak Rows N],* wobei A1,A2,..,Ak Attribute des Stroms sind. Zu jeder Partition wird ein Substrom erzeugt, basierend auf Gleichheit der Attribute, ähnlich einem Group By. Auf dem Substrom wird ein Tupel-basiertes Fenster der Größe N erzeugt wird. Als Ausgabe ergibt sich die Vereinigung dieser Fenster (Union).

Eine zeit-basierte Partitionierung würde keinen zusätzlichen Nutzen bringen, da die Tupel, die in das Zeitfenster fallen, in jedem Fall in der Ausgabe erscheinen. Die Gruppierung kann durch eine Sortierung herbeigeführt werden.

Relation-zu-Relation-Operatoren wurden SQL entnommen. Eine Relation kann in CQL genauso referenziert werden wie in SQL.

Es gibt in CQL drei Relation-zu-Strom Operatoren: *Istream* (insert stream), *Dstream* (delete stream) und *Rstream* (relation stream). Die formale Semantik ist wie folgt (∪ sei der Vereinigungsoperator):

Def 1.1:

Istream 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).

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

Also alle Elemente, die je in R eingefügt wurden, auch wenn sie zwischenzeitlich wieder gelöscht wurden.

Def 1.2 analog:

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

Also alle Elemente, die je aus R gelöscht wurden.

Def 1.3.

Rstream angewandt auf Relation R enthält ein Element <s,t> immer dann wenn das Tupel s in R zum Zeitpunkt t ist:

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

Also alle Elemente, die sich je in R befanden und befinden.

Beispiele:

1.

Gegeben sei eine Anwendung, welche dazu dient den Verkehr auf einer Autobahnbrücke mittels Sensoren und Lichtschranken zu erfassen. Jedes Fahrzeug, das von den Sensoren auf einer Spur in einem Streckenabschnitt mit einer Geschwindigkeit erfasst wird erzeugt einen Datensatz.

Das dazugehörige Schema sieht wie folgt aus: PosSpeedStr (Spur, Streckenabschnitt, speed).

Die Anfrage:

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

liefert einen Strom mit den Daten derjenigen Fahrzeuge, die seit Erfassungsbeginn gemessen wurden und schneller als 65 (km/h) fuhren.

Die Anfrage:

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

liefert das gleiche Ergebnis, sofern davon ausgegangen werden kann, dass kein Fahrzeug innerhalb eines Streckenabschnitts und einer Spur mehrmals erfasst wird.

(Formal: Wenn $R(t) \cap R(t-1) = 0$ für alle t)

Ansonsten liefert es diejenigen Daten, die zum aktuellen Zeitpunkt erfasst wurden.

2.

Gegeben sei das Schema wie bei 1.

Die Anfrage:

```
Select Spur, Streckenabschnitt, speed as Geschwindigkeit
From PosSpeedStr [Partition by Spur, Streckenabschnitt Rows 3]
Where speed > 65
```

liefert Spur, Streckenabschnitt und Geschwindigkeit der letzten 3 Fahrzeuge jeweils gruppiert nach Spur und Streckenabschnitt (als Relation).

Beispieleingabestrom (zeitlich aufsteigend geordnet):

[(1,1,71), (2,1,88), (3,2,85), **(1,1,67**), (**2,1,67**), (3,2,85), (**1,2,90**), (2,1,65), **(3,2,80),**
(**2,2,71**), (**2,1,77**), (**3,1,87),** (**1,1,77**), (2,1,65), (**3,2,85),** (**1,1,66),** (**2,1,68**), **(3,2,85),]**

Ausgabe (Fett markierte Tupel):

Spur	Streckenabschnitt	Geschwindigkeit
1	1	67
1	1	77
1	1	66
1	2	90
2	1	68
2	1	77
2	1	67
2	2	71
3	1	87
3	2	85
3	2	85
3	2	80

CQL erweist sich als Sprache, die sehr nahe an SQL angelehnt ist. Dies bietet viele bereits vordefinierte und implementierte Operatoren und erleichert Anwendern mit SQL-Kenntnissen den Einstieg. Nachteilig wirkt sich die Entscheidung, aus keine Strom-zu-Strom-Operatoren anzubieten, dies macht Anfragen mitunter in der Umsetzung unnötig aufwendig und in der Spezifikation unnötig komplex. Als Datenkonzept Ströme und Relationen zu unterstützen, erleichtert die Integration mit bestehenden, relationalen Datenbanksystemen, erfordert aber auch zusätzliche Operatoren, was wiederum die Anfragespezifikation erschwert. Außerdem scheint die zugrundeliegende Semantik etwas unglücklich, es findet keine reine Datenstromverarbeitung statt (Strom-zu-Strom-Operatoren) und auch die Zeitsemantik auf Relationen ist nicht immer intuitiv.

b) Hancock [2]

Um die Sprache und ihre Entstehung zu verstehen sind zunächst einige Informationen über den anwendungsspezifischen Hintergrund notwendig. Hancock ist eine Sprache, die von AT&T entwickelt wurde, um Daten aus dem Telekommunikationsbereich zu verarbeiten. So werden beispielsweise Gesprächsdaten von Mobiltelefonen erfasst um Auswertungen über die Dienstnutzung der Kunden zu machen. Darauf folgend können neue Dienste ausgearbeitet werden oder Telefonabrechnungen durchgeführt werden. Die dabei anfallenden Datenmassen sind jedoch enorm (300 Millionen Datensätze pro Tag von 100 Millionen Kunden [2]). Einerseits ist eine Speicherung aller Daten aufgrund der Menge oft gar nicht möglich, andererseits ist es mitunter auch gar nicht erwünscht, sämtliche Daten im Detail persistent zu speichern. Gründe dafür sind hohe Datenredundanzen oder

Existenz von für die Auswertung unwichtiger Daten. Häufig wird durch die Aggregation der Daten erst eine relevante Information generiert.. Es werden lediglich spezielle Profile, sogenannte Signaturen, gespeichert. Diese enthalten die relevanten Daten, die aus bestimmten Feldern des Datenstroms gewonnen, und immer wieder aktualisiert werden. So könnte man sich etwa vorstellen, dass die Anrufe, die von einem Anschluss ausgehen, zunächst einzeln erfasst und dann als Anzahl der ausgehenden Anrufe von diesem Anschluss in einer (bzw. als Teil einer) Signatur gespeichert werden. Bei solch einer großen Menge an Daten werden selbst die Signaturen so groß, dass sie nicht in den Hauptspeicher passen und deshalb extern (auf Festplatten) gespeichert werden müssen. Die Algorithmen erfordern also aufgrund der hohen Datenmenge und Ein-/Ausgabe-intensiver Operationen eine hohe Effizienz.

Früher verwendete man dazu C-Programme. Diese waren sehr effizient aber leider auch nur schwer lesbar, wartbar und verifizierbar. Da häufig Änderungen nötig waren und die C-Programme dabei hinderlich wurden, entwickelte man eine eigene, den Anforderungen gezielter entsprechende Sprache: Hancock. Hancock basiert auf C, bietet aber spezielle Konstrukte, die es dem Entwickler ermöglichen, von elementaren Problemen (Speicherung, Umwandlung, Durchlauf etc.) zu abstrahieren und sich so den anwendungsspezifischen Problemen zuwenden zu können. Außerdem wird die Lesbarkeit verbessert und eine evtl. chaotische Programmstruktur vermieden. Die Sprache wurde so gestaltet, dass es möglich ist, unabhängig von der Datenmenge, einfach Signaturprogramme zu lesen und zu schreiben. Da Hancock die Skalierung handhabt, ist es für Datenanalysten möglich schnell und einfach neue Signatur-Prototypen zu erstellen [2].

Zunächst einmal wird kurz die typische Struktur eines Signaturprogramms vorgestellt und anschließend folgt eine schrittweise Einführung in die Konstrukte von Hancock anhand einer Beispiel-Signatur für Mobiltelefonanfruferfassung. Datensätze von Aktionen, hier also Telefongesprächen, werden für eine gewisse Zeitperiode erfasst. Die Dauer hängt von der Anwendung ab. Danach erfolgt die Verarbeitung der Datensätze um die Signaturen zu aktualisieren. Jedoch ist es üblich aus Sicherheitsgründen zuvor eine Kopie der alten Signaturen zu speichern. Während der Verarbeitung werden die Datensätze üblicherweise nach bestimmten Feldern sortiert. Nach jeder Sortierung wird ein Durchlauf (engl. pass) durch die Daten gemacht. In diesem Durchlauf wird der Teil einer Signatur verändert, zu dem die sortierten Datensätze passen. Es wird der Teil der Signatur geändert auf den sich die sortierten Felder beziehen. Dazu wird der Teil der Signatur, der relevant ist, von der Platte geladen, überarbeitet und wieder gespeichert. Dadurch, dass die Sätze sortiert sind, ergibt sich in der Regel eine hohe Lokalität beim Zugriff auf die Daten auf der Platte und die Daten des Stroms werden nach dem relevanten Kriterium gruppiert.

Ein Beispiel:

Gegeben sei ein Datenstrom, der aus den Daten besteht, die beim Mobil-Telefonieren erfasst werden, um den Kunden eine Rechnung für ein- und ausgehende Mobiltelefonate zu erstellen. Der Strom enthält etwa 80 Millionen Sätze von 20 Millionen Kunden täglich. Für die Signatur sind aber nur

folgende Felder relevant: Mobil-Telefonnummer (MPN), gewählte Telefonnummer, erste und letzte benutze Sendestation.

Das Ziel der Anwendung ist es herauszufinden, wie das Mobiltelefon genutzt wird, vornehmlich in wenigen, benachbarten Gebieten (engl. Cells) oder über größere Regionen hinweg. Um dies zu bewerkstelligen wurde eine Signatur entworfen, die zu jeder MPN die fünf am häufigsten (und zuletzt) genutzten Sendestationen erfasst und außerdem wie oft andere Stationen genutzt wurden.

```
#define NDIV 5                      /*Anzahl der zu speichernden Sendestat.
#define MAX_TOWER_NAME 12           /*max. Länge des Sendestationsnamens
typedef struct {
      char celltower[NDIV][MAX_TOWER_NAME];     /*Name der Sendestationen
      float count[NDIV];            /*Häufigkeit ihrer Nutzung
      float other;                  /*Häufigkeit der Nutzung anderer
                                    /*Stationen
}profile;                           /*Name der Struktur
```

Die Variable `celltower` soll die Namen der 5 am meisten genutzten Sendestationen als Zeichenketten speichern. Die Variable `count` speichert die Häufigkeit (engl. Frequency) mit der die zugeordneten Türme genutzt werden und die Variable `other` zählt wie viele Anrufe von den fünf am häufigsten genutzten Sendestationen ausgingen.

Üblicherweise existiert von einem Strom eine logische und eine physikalische Repräsentation, wobei letztere zwecks Platzersparnis durch Enkodierung und Packen aus der ersten erzeugt wird. Bei den früheren C-Programmen hat man häufig beide verwendet, was mitunter zu Konfusion führte. Hancock führt hierfür ein Konstrukt ein, um dies zu verhindern, beziehungsweise zu verdeutlichen, welche Form gerade verwendet wird. Datenströme werden in Hancock als Typ definiert. Dazu gibt es eine Funktion, welche die Umwandlung von der physikalischen in die logische Darstellung durchführt:

```
stream AWS_s {                                  /*Stream, Name AWS_s
      getvalidAWS : awsPhy_t  => awsLog_t;      /*Umwandlungsfunktion
};
```

Hier wird der Stream-Typ AWS_s deklariert. Die Funktion `getvalidAWS` bestimmt, ob der physikalische Satz (vom Typ. `awsPhy_t`) gültig ist und liefert den logischen Satz (vom Typ `awsLog_t`) zurück.

Da die Daten einer Signatur oft noch zu groß sind, wird versucht, sie zu komprimieren, um beim Schreiben so wenig Bytes wie möglich ändern zu müssen. Dazu werden Quantisierungen oder Approximationen vorgenommen. Es ergibt sich also noch eine dritte Darstellung der Daten: eine approximative. Um zwischen verschiedenen Repräsentationen der Daten umzuschalten gibt es ein spezielles Konstrukt, die sogenannte View. Diese ist hier im Beispiel allerdings nicht von Bedeutung.

Ein weitere Komponente sind sogenannte Maps. Sie dienen dazu Werte mit Schlüsseln zu assozieren, und durch direkte Adressierung auf Werte in komprimierter Form zuzugreifen oder sie zu speichern.

Der Programmierer kann die approximierte Darstellung der Daten direkt speichern, die Komprimierung (in die physikalische Form) übernimmt das System.

```
map cellTower_m {
      key pn_t;                    /*verwende Schlüssel des Typs pn_t
      value profile;               /*verwende Werte des Typs profile
      default {{'\0', '\0', '\0', '\0', '\0'}, /*Default-Wert
            {0.0, 0.0, 0.0, 0.0, 0.0},
            0.0};
      compress ctSqueeze;          /*zu verwendende Komprimierungfunktion
      decompress ctUnsqueeze;      /*zu verwendende Dekomprimierungsfkt.
};
```

Hiermit wird ein Map-Typ namens *cellTower_m* definiert. Als Schlüssel definiert sind Werte vom Typ *pn_t* (Typ der physikalischen Darstellung). Als Wert definiert sind Objekte vom Typ *profile* (die Signatur), also der logischen Darstellung. Überlicherweise werden aber Typen der approximierten Darstellung verwendet und es wird nachher im Programm mit Hilfe von Views zwischen logischer und approximierter Darstellung hin-und hergeschaltet. Die *default*-Klausel gibt an, welcher Wert geliefert werden soll, falls kein Wert zum angegebenen Schlüssel gefunden wird. Mit *compress* und *decompress* können Funktionen spezifiziert werden, welche die Kompression der Werte übernehmen sollen. Es existieren jedoch auch Standardversionen für die Kompression. Um auf Werte in der Map zuzugreifen existiert ein spezieller Operator <: ... :>:

```
cellTower_m ct;
pn_t mpn;
c = ct<:mpn:>;          /*ordne c (Typ profile) den Wert aus der map zu,
                         *der mit dem Schlüssel mpn verknüpft ist
…
ct<:mpn:> = c;          /*ordne der map unter dem Schlüssel mpn den Wert
                        /*von c zu
```

Ein weiteres (implizites) Konstrukt sind Events. Mit ihrer Hilfe lassen sich bestimmte Ereignisse im Datenstrom signalisieren und anschließend verarbeiten. Dazu verwendet man sogenannte *Multi-Unions*. Mit letztgenannten definiert man eine Reihe von Marken, die Namen der eingetretenen Events, und ordnet jeder einen Typ zu, der Objekttyp, durch dessen Vorkommen im Datenstrom ein Event signalisiert wird. Jede Untermenge von Instanzen, der im Multi-Union definierten Objekttypen (auch die leere Menge {: :}), kann ein Objekt vom Typ dieses Multi-Unions sein.

```
                    Typ             Marke
munion line_e {:    areacode_t      npa_begin,
                    exchange_t      nxx_begin,
                    pn_t            line_begin,
                    awsLog_t        call,
                    pn_t            line_end,
                    exchange_t      nxx_end,
```

```
                areacode_t        npa_end :} ;
```

Nachdem die Daten (ein Auftauchen signalisiert einen Event), die von Interesse sind, definiert wurden, benötigt man Funktionen, um diese Events zu erfassen. Eine Erfassungs-Funktion betrachtet einen kleinen Teil (ein Fenster) des Stroms und erzeugt als Rückgabe eine *Multi-Union*, um das erkannte Event zu beschreiben.

Ein Fenster wird wie folgt definiert:

```
awsLog_t *w[3:1]
```

mit awsLog_t Typ (hier der logische Datensatz unsere Stroms), w als Name des Fensters, 3 als Größe des Fensters und 1 als Index des aktuellen Satzes. Alle Sätze des Fensters mit Index kleiner 1 stellen frühere Sätze im Strom dar, alle Sätze mit Index größer 1 spätere Sätze des Stroms (beginnend bei 0).

Eine Erfassungsfunktion kann wie folgt ausssehen:

```
line_e originDetect(awsLog_t *w[3:1])  /*Rückgabewert vom Typ line_e, Name
                                        *originDetect,
                                        *Eingabe Fenster vom Typ awsLog_t
{ line_e b,e;                          /*Deklaration zweier Variablen vom
                                        *Typ line_e
b = beginDetect(w[0], w[1]);           /*Hilfsfunktion, bestimmt den Anfang
e = endDetect(w[1], w[2]);             /*Hilfsfunktion, bestimmt das Ende
return b:+: {: call = *w[1] :} :+: e;  /*Rückliefern eines multi-unions vom
                                        *Typ line_e
}
```

Um letztlich einen Strom zu verarbeiten, muss ein Durchlauf definiert werden. Dieser beinhalted Filterung, Sortierung und Reagieren auf Events. Dazu gibt es das *iterate*-Konstrukt:

```
iterate
      (over stream variable              /*zu durchlaufender Strom
      filteredby filter predicate        /*Prädikat oder Funktion das
                                         *auszuschließende Sätze ausfiltert
      sortedby sorting order             /*Sortierung nach angegebenen
                                          *Feldern
      withevents event detection function)/*zu verwendende Event-
                                          *Erfassungsfunktion
{
event clauses     /*Aktionen, die auszuführen sind, wenn ein
                   *entsprechendes Event erkannt wurde. Die
                   *Reihenfolge der Aufführung bestimmt die der Abarbeitung
                   *(bei simultanem Auftreten von Events d.h. im selben
                   *Multi-Union-Wert), Hancock bestimmt daraus einen Fluss-
                   *Graphen
};
```

Hier eine Beispielmethode, die das iterate-Konstrukt verwendet:

```
Void out(AWS_s calls, cellTower_m ct)          /*Methode out, führt eine
                                                *Eventerfassung durch und
                                                *ändert Signaturen
{
      profile p;                               /*definiere Signatur vom Typ
                                                *profile

      iterate                                  /*iteriere
            ( over calls                       /*über den Strom calls
              filteredby completeCellCall      /*mit Filterfunktion
                                                *completeCellCall
              sortedby origin                  /*sortiere nach Attribut
                                                *origin
              withevents originDetect ) {      /*erfasse Events mit der
                                                *Funktion originDetect

            event nxx_begin(exchange_t npanxx) {/*definiere Event
                                                *nxx_begin, übergebe
                                                *Parameter npanxx, führe
                                                *Funktion degradeBlock aus
                  degradeBlock(ct, npanxx);
            }

            event line_begin(pn_t mpn) {       /*Event line_begin …
                  initProfile(&p);
            }

            event call(awsLog_t c) {           /*Event call …
                  diversify(&p, c.cellid);
            }

            event line_end(pn_t mpn) {         /*Event line_end …
                  profile mytemp;
                  mytemp = ct<:mpn:>;
                  ct<:mpn:> = update(&mytemp,&p);
            }
      };
}
```

Schließlich wird das Ganze als eine Anwendung nach dem zu Beginn vorgestellten Schema ausgeführt. Dazu dient die Methode *sig_main*, welche die Kommandozeileneingaben auf Programmvariablen abbildet.

```
void sig_main (const AWS_s calls <a:>,    /*Parameter a wird als Stream
                                           *abgebildet (Typ AWS_s)
            exists const cellTower_m oldCT <m:>,
                                          /*Parameter m wird als Map
                                           *vom Typ cellTower_m auf Variable
                                           *oldCT abgebildet
            new cellTower_m newCT <M:>) { /*Parameter M wird als Map
                                           *vom Typ cellTower_m auf Variable
                                           *newCT abgebildet

      newCT :=: oldCT;                    /*Kopie der alten Map erzeugen
      out(calls,newCT);                   /*Methode out Aufrufen (Stream-
                                           *und Signaturverarbeitung)
}
/*die Schlüsselwörter exists und new sind hier nicht relevant
```

Wie man sieht, ist Hancock eine ganz andere Art von Anfragesprache. Sie ist aus einer anwendungsspezifischen Domäne heraus entstanden, wobei die früher verwendeten C-Programme durch eine eigene, den besonderen Anforderungen angepasste, C-ähnliche Sprache ersetzt wurden. Hancock-Programme werden übrigens von einem Precompiler in C übersetzt und laufen auf einem speziellen Laufzeitsystem. Sie können einerseits durch individuelle Funktionen spezifisch gestaltet werden, andererseits durch den Einsatz generischer Methoden schnell entwickelt und zusammengesetzt werden. Was bleibt ist eine hohe Effizienz aber auch eine sehr starke Einschränkung in der Einsetzbarkeit aufgrund der geringen Ausdruckskraft und Universalität (bzw. des hohen Aufwandes zu deren Erreichung). Um die Möglichkeiten zu erweitern wird eine sehr genaue Kenntnis über die zu verarbeitenden Datenströme benötigt. Der Kode wird umfangreich und für den Laien schwer verständlich.

c) Aurora ([4],[5],[6])

Aurora ist ein DSMS, welches von Mitarbeitern des MIT, der Brandeis Universität und der Brown Universität entwickelt wurde. Bekannte Anwendungen sind z.B. Sensornetzwerke, Finanzapplikationen oder Auswertung von standortbasierten Daten (z. B. mit Hilfe von GPS). Anders, als bei den bisher vorgestellten Ansätzen, verwendet man zum Erstellen von Anfragen durch den Endnutzer weder eine deklarative, textbasierte Sprache wie CQL, noch eine prozedurale wie Hancock. Statt dessen gibt es eine graphische Umgebung, auf der sich Anfragen mit sogenannten Boxen (engl. *Boxes*) und Pfeilen (engl. *Arrows*) zusammenstellen lassen (die system-interne Umsetzung sieht

etwas anders aus). Die Boxen stellen Operatoren dar. Es gibt von ihnen eine überschaubare Anzahl mit unterschiedlichen Funktionen. Die Pfeile verbinden nun diese Boxen, sie repräsentieren den Datenfluss. Die Operator-Algebra wird auch SQuAl genannt (steht für Stream Query Algebra) [9]. Dieser Ansatz, auch Boxes-and-Arrows-Paradigma genannt, wird in vielen Prozessfluss- und Workflowsystemen verwendet. Aurora selbst versteht sich grundsätzlich als Datenfluss-System [5]. Die Rolle des Menschen ist dabei passiv, die des Datenbanksystems aktiv (DAHP-Modell) [4]. Der Grund, sich gegen Ansätze wie CQL zu entscheiden, war, dass man annahm, die Eliminierung gemeinsamer Unterausdrücke für eine große Zahl solcher Anfragen sei zu aufwändig und schwierig [6]. Grundstruktur des Systems:

Ein einfacher Strom (*simple stream*) ist eine potentiell unendliche Folge von Tupeln, welche die gleiche Strom-ID besitzen. Eine Anfrage ist ein Subnetzwerk, dass an einer einzigen Ausgabe endet und eine beliebige Anzahl von Eingaben hat. Zyklen sind dabei nicht erlaubt. Ströme können aufgespalten (entspricht dem Erzeugen zweier identischer Ströme) und zusammengefügt (entspricht einem Join) werden [6] (für diese Operationen gibt es eigene Boxen).

Ein Subnetzwerk besteht aus einem System von Boxen und Pfeilen, welche die Daten der Ströme manipulieren und steuern, um den Ausgabestrom mit Daten gemäß der Anfrage zu erzeugen. Der Ausgabestrom wird einer Anwendung zur Verfügung gestellt. Zu jeder Anfrage gehört auch eine QoS-Spezifikation, welche angibt, welche Eigenschaften die Ausgabe auch im Überlastfall noch haben muss (dazu später mehr).

Anfrage-Modell:

Aurora bietet drei Arten von Anfragen: *kontinuierliche Anfragen*, Sichten *(Views*) und *Ad-Hoc* Anfragen [4]. Kontinuierliche Anfragen leiten die Ströme durch die Boxen, in denen sie verarbeitet werden bis hin zum Ausgang. Nachdem alle erreichbaren Pfade durchlaufen wurden, werden die Daten aus dem Netzwerk entfernt, es findet keine Zwischenspeicherung statt.

Sogenannte Verbindungspunkte (*connection points*) bieten die Möglichkeit, das Netzwerk dynamisch zu ändern. An sie lassen sich neue Boxen und Pfeile anschließen um neuen Anwendungen eine Ausgabe zu bieten (z.B. Ad-Hoc Anfragen). Sie bieten auch die Möglichkeit, Daten vorübergehend persistent zu speichern, etwa um den später gestellten Anfragen historische Daten zur Verfügung zu stellen. Dazu ist es notwendig zu spezifizieren, wie lange die Daten gespeichert werden sollen. Zum Beispiel bewirkt der Ausdruck *keep 2 hrs* eine Speicherung der Daten der letzten zwei Stunden.

Sichten sind Pfade (Anfragesubnetzwerk), die von einem Verbindungspunkt ausgehen, an die jedoch keine Anwendung fest angeschlossen ist. Letztere verbinden sich bei Bedarf dynamisch und erhalten je nach Last des Systems die im Verbindungspunkt und auf dem Pfad gespeicherten Daten ganz oder teilweise.

Operatoren (Boxen):

Es gibt in Aurora acht [4] Operatoren ([5] nennt 9 Operatoren).

Dazu gehören vier verschiedene Fenster-Operatoren (engl. windowed operators). Diese Fenster-Operatoren arbeiten auf einer Menge von aufeinanderfolgenden Tupeln, wobei das, was genau mit den Tupeln geschieht, durch eine vom Benutzer anzugebende Funktion bestimmt wird. Die vier Operatoren selbst bestimmen nur das Fenster bzw. dessen Daten.

Der Operator *Slide* durchläuft den Datenstrom indem es ein Fenster angegebener Größe immer um ein angegebenes Stück verschiebt.

Bsp: Tupel seien von 1 an durchnumeriert, die Fenstergröße sei 10, die Schiebegröße 1:

Bearbeitete Fenster: [1-10],[2-11],[3-12]...

Der Operator *Tumble* funktioniert so ähnlich wie Slide, jedoch enthält ein Fenster nie Tupel eines vorherigen Fensters (die Schiebegröße ist also mindestens so groß wie das Fenster):

Bsp: [1-10],[11-20],[21-30]...

Der Operator *Latch* ähnelt Tumble, er kann jedoch intern den Status zwischen zwei Fenstern speichern (z. B. für unendliche Berechnungen [1-10],[1-20],[1-30]...).

Der Operator *Resample* produziert einen teilweise synthetischen Strom indem er Tupel zwischen aktuellen Tupeln eines Eingabestroms interpoliert.

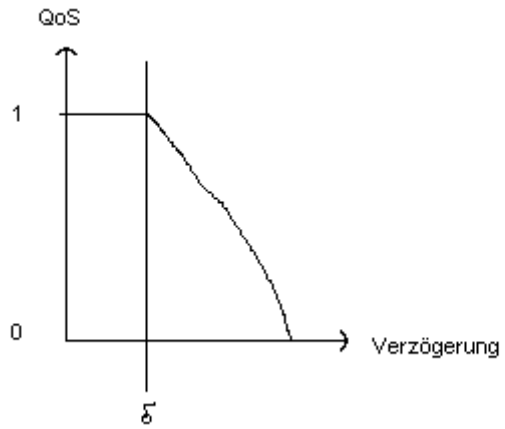
Weitere Operatoren sind *Filter* und *Drop*, welche Daten nach Auswahlprädikaten ausfiltern respektive Daten nach einer Auslassrate weglassen. Dann gibt es noch *Map*, welches eine zu spezifizierende Funktion auf die Eingabetupel anwendet. Der *Join*-Operator paart Tupel von Eingabeströmen aufgrund einer spezifizierten Distanz, die nicht überschritten werden darf (z.B. der Differenz der Zeitstempel) [4]. Zuletzt gibt es noch den Fenster-Sortier-Operator *WSort* und den *Union* Operator [5].

Anzumerken ist, dass viele andere Operatoren, die nicht in Aurora vorhanden sind, sich mit den zu Verfügung stehenden erzeugen lassen..

Quality of Service (QoS)

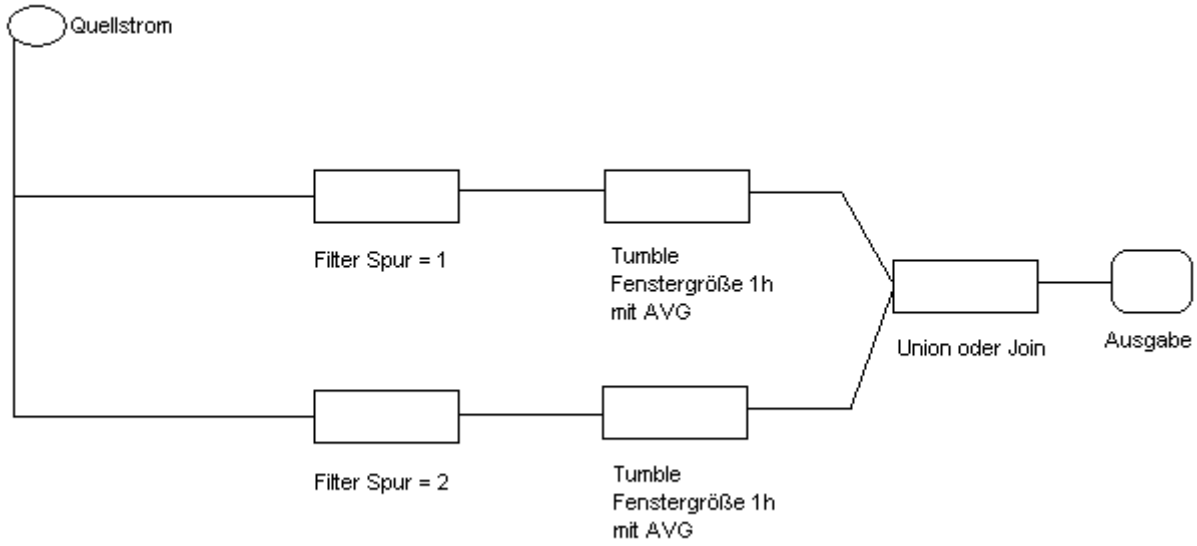
Eine präzise Antwort auf eine Anfrage ist oft nicht erreichbar oder zumindest nicht erforderlich (wenn z. B. nur Tendenzen oder Durchschnitte etc. verlangt sind). Deshalb ist es nicht das Ziel von Aurora, absolute Präzision zu erreichen. Im Allgemeinen gibt es eine zusammenhängende Menge von Antworten, mit einer messbaren Abweichung von der perfekten Antwort, die dennoch akzeptabel sind. Die QoS-Spezifikation gibt an, welche (applikationsabhängige) Abweichung tolerierbar ist [5].

Quality of Service kann in Aurora mit drei Graphen spezifiziert werden. Dies sind der Verzögerungsgraph (*latency-graph*), der Auslassungsgraph (*loss-tolerance-/drop-graph*) und der Wertegraph (*value-based-graph*). Zu jeder Anfrage im System muss mindestens der Verzögerungsgraph angegeben werden. Dieser Graph sieht etwa folgendermaßen aus:



Alle Werte der Verzögerung kleiner (also links von) δ liegen in der „good zone“, sind also akzeptabel. Wird der Wert überschritten, so muss eine Lastverkleinerung (engl. `load shedding`) stattfinden.
Der Auslassungsgraph hingegen gibt an, wie sich der Quality of Service ändert, wenn die Anzahl der ausgelieferten Tupel im Verhältnis zur Anzahl der eingegangenen Tupel abnimmt (spezifiziert als der Prozentsatz, der sich aus dem Quotienten #ausgehende Tupel / #eingehende Tupel ergibt). Mit anderen Worten also wird angegeben, wie viele Eingabetupel verworfen werden dürfen ohne die Signifikanz der Ausgabe zu gefährden.
Der Wertegraph gibt an, welche Werte der Ausgabe am wichtigsten sind. So kann Aurora feststellen, ob die produzierten Ausgabewerte wichtig sind oder nicht. Damit kann dann entschieden werden, ob es angebracht, ist bestimmte Tupel eher auszulassen als andere. Beispielsweise könnten bei Überwachungsdaten eines Reaktors, die Werte, die nahe am Grenzwert liegen, besonders wichtig sein, während Werte im unkritischen Bereich unwichtig sind und bei Überlast eher verworfen werden dürfen.
Aurora geht davon aus, dass die angegebenen Graphen konvex sind (so wie der obige). Dies ist in den meisten Fällen (außer in bestimmten Anwendungen, wie z. B. mit dem Reaktor) nicht nur naheliegend, sondern für das Funktionieren der Algorithmen der Laststeuerung auch erforderlich (Aurora verwendet `Gradient Walking` [4]). Außerdem ist Auroras Verständnis von QoS nicht grundsätzlich auf diese drei Typen von Graphen beschränkt, auch andere Kriterien (z.B. Durchsatz) wären denkbar [4].
Eine Lastverteilung findet dann statt, wenn Aurora feststellt, dass die mit den Graphen spezifizierten QoS-Kriterien nicht mehr erfüllt werden können, etwa aufgrund zuvieler Eingaben oder zu weniger Ressourcen. Überlicherweise wird bei der Dimensionierung des Systems statisch, also vor Laufzeit, eine Reserve einkalkuliert, so dass zur Laufzeit noch Ressourcen zur Verfügung stehen, wenn etwa die Eingaben zunehmen oder Ad-Hoc-Anfragen mit ins System aufgenommen werden. Kommt es dennoch zu einer Überlastsituation, so betrachtet man die QoS-Spezifikation um festzustellen, wie die Last zu verringern ist, ohne die Aussagekraft oder Korrektheit der Ausgabe zu beeinflussen. Zieht man dabei nur die Verzögerungsspezifikation zu Rate, so besteht eine Strategie darin, so früh im Netzwerk wie möglich beliebige Tupel auszulassen, möglichst gerade so viele, dass die QoS-Spezifikation noch eingehalten wird. Dabei kann es aber passieren, dass je nach Anwendung wichtige oder zu viele Tupel

ausgelassen werden. Um dem entgegen zu wirken ist es möglich den Auslassungsgraphen und/oder den Wertegraphen heranzuziehen um die Anzahl und Auswahl der auszulassenden Tupel der Anwendung entsprechend anzupassen.
Ein anschauliches Beispiel: Gegeben seien die Anwendung und das Schema vom CQL-Beispiel mit der Autobahnbrücke. Jedoch wollen wir jetzt die Durchschnittsgeschwindigkeit aller Fahrzeuge der letzten Stunde getrennt nach Spur 1 und Spur 2 erhalten:
Erläuterung:
Der Eingabestrom wird in zwei Filter-Operatoren geleitet, in denen die Daten jeweils einer Spur ausgefiltert werden (Filter-Attribut ist Spur). AVG sei die zu definierende In-Fensterfunktion, die den Durchschnitt berechnet. Es sei angenommen, dass Tumble so lange blockiert, bis die Daten der letzten Stunde empfangen sind, die Ausgabe sei je Stunde ein Tupel mit der Durchschnittsgeschwindigkeit. Zur Vereinigung der beiden Tupel sind Union oder Join (ohne Prädikat) denkbar (Ausgabe zwei oder ein Tupel). Sinn machen würde auch ein Verbindungspunkt, der die Daten der letzten Stunde speichert und sie dieser Anfrage zur Verfügung stellt.



Durch die graphische Verwendung der Operatoren lässt sich eine Anfrage in Aurora also auf andere Weise als bisher zusammenstellen. Eine gewisse Ähnlichkeit mit deklarativen Ansätzen ist dennoch vorhanden. Die Spezifikation der Anfrage entsteht nämlich durch vordefinierte Operatoren, nur geschieht dies nicht verbal/textuell sondern graphisch/anschaulich. Die Entstehung eines Flusses aus Operatoren, verknüpft durch die Pfeile, darf jedoch nicht als benutzerseitige Optimierung missverstanden werden. Die system-interne Anfrageverarbeitung und -optimierung geschieht auf andere Art und Weise. Außerdem scheint Aurora einer der wenigen Ansätze zu sein, der sich überhaupt Gedanken um QoS macht. In den anderen hier vorgestellten Systemen ist dies nicht der Fall.

d) GSQL [7]

GSQL ist die Sprache, die im DSMS Gigascope von AT&T verwendet wird. Gigascope wird eingesetzt zur Überwachung und Auswertung von Netzwerkdaten. GSQL ist eine eingeschränkte Variante von SQL mit Erweiterungen für Datenströme. Im Gegensatz zu CQL, welches letztendlich auch eine Erweiterung von SQL ist, setzt GSQL, wie auch StreaQuel, voll auf das Stream-Konzept. Es gibt keine Relationen, alle Eingaben und Ausgaben von Operatoren und Anfragen sind Ströme. Diese Entscheidung erlaubt eine präzise Anfragesemantik, ermöglicht eine komplexe Verschachtelung von Anfragen [7] und vereinfacht die Implementation von schnellen Operatoren (es muss keine Umwandlung von Strömen in Relationen vorgenommen werden oder umgekehrt, es gibt keine sich mit der Zeit verändernde Relationen). Das Anfragemodell einer kontinuierlichen Anfrage über ein Sliding-Window, welche als Zwischen- oder Endresultat eine Relation liefert, wie es in vielen anderen Systemen Anwendung findet, erschien hier (trotz einiger Vorteile, z. B. die Präsentation der Ergebnisse für den Endanwender) für die Analyse von Netzwerkdaten unangebracht. Nachteilig sind nämlich die Schwierigkeiten bei der Zusammenfügung von Anfragen (Ströme müssen bei geschachtelten Operatoren mitunter zu Relationen umgewandelt werden oder umgekehrt). Auch die teilweise schwierige Semantik einer Anfrage bzw. umgekehrt das Erzeugen einer Anfrage, welche die gewünschte Semantik enthält, sind von Nachteil. Ebenso auch die aufwendigere Implementierung [7].
Um das Problem der blockierenden Operatoren in Angriff zu nehmen (in anderen System verwendet man hierzu die eben angesprochenen Sliding-Windows) werden hier die in jedem Tupel vorhandenden Zeitstempel oder Sequenznummern, die in der Regel monoton fallen oder steigen, betrachtet. Sie werden definiert als geordnete Attribute mit Ordnungseigenschaften (z.B. `strikt/monoton zunehmend/abnehmend, monoton nichtwiederholend, in Gruppe zunehmend`). Diese geordneten Attribute werden genutzt um die blockierenden Operatoren in Stromoperatoren umzuwandeln. Zum Beispiel beim Join-Operator mit der Ordnungseigenschaft `monoton zunehmend`. Das Join-Prädikat muss eine Bedingung (Constraint) auf einem geordneten Attribut enthalten, um damit ein Join-Fenster zu definieren. Beispiel:
Seien B und C Ströme und ts ein Attribut, welches in beiden vorkommt, so könnte die Join-Bedingung lauten:

1. B.ts = C.ts oder
2. B.ts >= C.ts – 1 und B.ts <= C.ts + 1

Betrachtet man nun beispielsweise folgende Daten für B.ts
(si:n bedeute i. Tupel der Ordnung, Wert des Attributs ts = n):
[s1:1, s2:1, s3:2, s4:3, s5:3, s6:3, s7:4, s8:4, s9:5] und für C.ts:
[s1:0, s2:2, s3:2, s4:2, s5:3, s6:4, s7:5, s8:6] (also monoton steigende Ordnung).
Die Join-Fenster eines Join mit Bedingung 1 wären folgende:

[B.s3, C.s2, C.s3, C.s4], [B.s4, B.s5, B.s6, C.s5], [B.s7, B.s8, C.s6], [B.s9, C.s7]

Sobald also Tupel mit höherer Sequenznummer ankommen, wird das Fenster der vorherigen Sequenznummer für beendet erklärt, das genaue Verfahren hängt von der Implementierung und von der Ordnungeigenschaft des Attributes ab).

Zur Zeit unterstützt GSQL Selektion, binären Join, Aggregation und Strömemischung (engl. stream merge). Es gibt zwei Arten von Strömen, *Protokolle*, so genannt wegen der Herkunft des Systems, und *Ströme*. Ein Protokoll ist ein Strom, der durch Interpretation der Eingabepakete mit Hilfe von Interpretationsfunktionen entstanden ist. Ein Strom ist das Ergebnis einer Anfrage. Ein Protokoll wird an ein Interface gebunden, ein symbolischer Name, der vom Laufzeitsystem mit einer Quelle von Paketen verbunden wird. Hier eine Beispielanfrage welche Ziel_IP, Port und einen Zeitstempel von TCP-Paketen auf eth0 (der ersten Ethernet-Interface Karte) liefert:

```
DEFINE {query name tcpDest0; }
Select destIP, destPort, time From eth0.TCP
Where IPVersion = 4 and Protocol = 6
```

(hierbei ist eth0.TCP ein Protokoll)

Die *DEFINE* Sektion einer Anfrage ermöglicht es, Eigenschaften zu setzen, hier also der Name der Anfrage als tcpDest0. Eine andere Anfrage oder Anwendung kann nun diese im *FROM* – Teil referenzieren, somit ist, nach der obigen Definition, ein Strom entstanden.

Eine Erweiterung zu SQL ist der Merge-Operator, der dem herkömmlichen Union-Operator ähnlich ist, allerdings die Ordnungseigenschaften eines Attributes beibehält.

Beispielanfrage (tcpDest1 sei analog zu tcpDest0 definiert):

```
DEFINE {query name tcpDest; }
Merge tcpDest0.time : tcpDest1.time
From tcpDest0, tcpDest1
```

Auf diese Weise unterstützt Gigascope auch geschachtelte Unteranfragen.

Es ist auch möglich, benutzerdefinierte Funktionen zu erstellen und in GSQL verfügbar zu machen (Funktionsbibliotheken, -Registrierung). Parameter einer Funktion können auch als *Handle* übergeben werden, sie sind gewissermaßen externe Parameter, z. B. eine Datei, die vom Laufzeitsystem beim ersten Aufruf der Funktion eingelesen und in den Speicher gelegt werden.

```
Select peerid, tb, count(*) FROM tcpDest
Group by time/60 as tb,
getlpmid(destIP,'peerid.tbl') as peerid
```

Das Attribut time ist ein Feld mit der Granularität 1 Sekunde, time/60 erzeugt eine Granularität von 1 Minute. Die Funktion getIpmid hat einen Attributparameter (destIP) und einen Handle-Parameter (hier eine Datei mit Routingeinträgen) und liefert Subnetzinformationen (hier unwichtig).

An den Beispielen wird deutlich, dass Gigascope und GSQL speziell für die Bedürfnisse von Netzwerkanalysten entwickelt wurden. Das Datenbanksystem kann um eigene Funktionen erweitert werden, mehrere Anfragen können gleichzeitig im System angelegt und von anderen referenziert

werden. Besonders auffällig ist der Mechanismus für die Umsetzung blockierender Operatoren, der auf definierbarer zeitlicher Ordnung beruht und somit sein Zeitfenster anders als die bisher vorgestellten Ansätze bestimmt. Zu Performance und QoS lässt sich sagen, dass es auch hier nicht unbedingt auf die Schnelligkeit der Antwort ankommt, sondern vielmehr darauf, wie hoch die Eingaberate sein kann ohne Tupel verwerfen zu müssen. In der Praxis hat sich gezeigt, dass Gigascope sehr schnell ist. (In Experimenten mit mittlerer Hardware und hohem Input ließen sich Verlustraten von weit unter 2% erzielen. Diese Rate war gewissermaßen die QoS-Anforderung. Ein anderes System mit Dual 2,4 GHz CPU und Eingaberaten von bis zu 1,2 Millionen Paketen pro Sekunde läuft seit drei Monaten ohne Unterbrechung.)

e) StreaQuel [8]

StreaQuel ist ebenfalls eine SQL-basierte Sprache, entwickelt von der Berkeley-Universität. Sie wird vorausssichtlich eingesetzt im Stream-System TelegraphCQ [8]. StreaQuel arbeitet wie GSQL nur auf Strömen, also nicht auf Relationen.
Ströme können auf zwei Arten erzeugt werden: ausgehend von einer unendlichen, zeitmarkierten Tupelfolge oder durch Umsetzung einer herkömmlichen Relation. Unterstützt wird eine logische und eine physikalische Zeitsemantik. Um sich in den Strömen fortzubewegen bzw. sie zu verarbeiten, kommt auch hier die Sliding-Window-Technologie zum Einsatz Es wird eine ganze Reihe von Fenstervarianten angeboten (s. u.). Dafür gibt es einen Cursor-Mechanismus, der es dem Benutzer erlaubt, seine eigene Fensterverarbeitung umzusetzen.
Die Syntax von StreaQuel ist noch nicht endgültig ([8]), generell sieht sie aber folgendermaßen aus:

```
SELECT projection_list                    /*Auswahlliste
FROM  from_list                           /*Quell-Strom-Liste
WHERE selection_and_join_predicates       /*wie üblich; Auswahl
ORDEREDBY                                 /*Sortierung
TRANSFORM…TO                              /*s.u.
WINDOW…BY                                 /*s.u.
```

Weitere Schlüsselwörter sind:

```
NOW         aktuelle Zeit                 /*Nutzung in Fenster-Ausdrücken
ST          Startzeit der Anfrage         /*s. o. + Transform-Klausel
```

Optional gibt die Transform - Klausel an, welche Fenster für die Anfrage erzeugt werden:

```
Transform Stream1 For ( t = ST; t < ST + 10; t++) To Stream1(t)
Window Stream1 By ST, t
```

Diese Spezifikation generiert mit jedem Schleifendurchlauf ein Fenster, welches als Startzeitpunkt *ST* und als Endzeitpunkt *t* hat. Da *t* in jedem Durchlauf um 1 erhöht wird, werden folgende Fenster generiert:
Sei *ST* (der Startzeitpunkt) = 40: [40, 40], [40, 41], [40, 42], [40, 43], .., [40, 50]

Die folgenden Sliding-Window-Arten lassen sich mit dem Transform – Konstrukt erzeugen (Tabelle 3.1):

Name	Intervallbegin	Intervallende
Snapshot	Fest	Fest
Landmark	Fest	Fortlaufend
Sliding	Fortlaufend	Fortlaufend
Reverse Landmark	Rücklaufend	Fest
Reverse Sliding	Rücklaufend	Rücklaufend
..	..	..

Einige Beispielanfragen:
1. Gegeben sei eine Netzwerküberwachungsanwendung, welche Informationen über die transportierten Pakete sammelt und verarbeitet.
Der Eingabestrom *Pakete* mit Paketinformationen sei mit Attributen wie folgt definiert*:*
Pakete (pID, länge, zeit)
Aufgabe: Erzeuge einen Strom von denjenigen Paketen deren Länge mehr als zweimal so groß ist, wie der Durchschnitt der Paketlängen der letzten Stunde:

```
AVGPAKETE:
Select AVG(length) as avlen          /*bilde den Durchschnitt
From Pakete                          /*über Pakete
Window Packets By (NOW - 1hr, NOW)   /*im Zeitfenster [jetzt – 1 Stunde,
                                      *jetzt]

GREATERTHAN2AVG:
Select *                             /*liefere alle Felder
From Pakete p, AVGPAKETE a           /*verknüpfe Pakete mit dem Durchschnitt
Where p.length > 2 * avlen;          /*wähle nur die Pakete deren Länge mehr
                                      *als 2-mal so groß ist wie die des
                                      *Durchschnitts
Window p By (NOW, NOW)               /*über dem Zeitfenster [jetzt,jetzt],
                                      *also nur aktuelle Pakete
STREAM: Open a delta-output cursor on GREATERTHAN2AVG.
```

```
                                        /*erzeuge einen Ausgabestrom mit den
                                         *Tupeln aus Anfrage GREATERTHAN2AVG
```

2. Gegeben sei eine Grundstücksüberwachungsanwendung, welche Daten von auf dem Grundstück verteilten Sensoren erfasst und verarbeitet.
Gegeben seien die beiden folgenden Eingabeströme:

```
SquirrelSensors(sID, region, time)  /*Meldungen von Erschütterungssensoren
SquirrelType(sID, type))            /*Erschütterungsklasse (zu jeder Meldung)
```

Aufgabe: Erzeuge einen Alarm wenn mehr als 20 Erschütterungen der Klasse ’A’ in Jennifers Garten sind
Option 1: Zwanzig verschiedene Erschütterungen der Klasse ‘A’ sind seit Beginn der Zeit entdeckt worden

```
Select Alert()                          /*Alarmfunktion
From SquirrelSensors ss, SquirrelType st /*Meldungen mit ihren
                                         *Erschütterungsklassen
                                         *kombinieren
Where ss.region = JGARDEN AND           /*wenn region = jennifers garten
        ss.id = st.id AND               /*id der Meldung = id der Klasse
                                         *der Meldung
        st.type = A                     /*Erschütterung vom Typ ‘A’
Having COUNT (DISTINCT(ss.id)) > 20     /*mehr als 20 mal
```

Option 2: Zwanzig verschiedene Typ ‘A’ Erschütterungen im Garten, in diesem Moment

```
Select ALERT()                          /*Alarmfunktion

From SquirrelSensors ss, SquirrelType st /*s.o.
Where ss.region = JWGARDEN AND          /*s.o.
        ss.id = st.id AND               /*s.o.
        st.type = A                     /*s.o.
Having COUNT (DISTINCT(ss.id)) > 20     /*s.o.
Window ss by (NOW, NOW)                 /*über dem Zeitfenster von
                                         *[jetzt,jetzt], also aktuell
```

Anscheinend wurde bei StreaQuel sehr großer Wert auf das Sliding-Window Konzept gelegt. In keiner der hier vorgestellten Sprachen können so viele Arten von Fenstern erzeugt werden. StreaQuel ist in

dieser Hinsicht sehr flexibel. Der Rest der Sprache ist auch sehr nahe an SQL angelehnt, dies erleichtert den Einstieg, die Semantik ist größtenteils intuitiv. QoS-Aspekte werden in dieser Sprache derzeit nicht berücksichtigt. Insgesamt scheint die Sprache noch entwicklungsfähig.

4. Vergleich der vorgestellten Anfragesprachen

Die hier vorgestellten Anfragesprachen lassen sich grob in zwei verschiedene Kategorien einteilen:
- deklarative Sprachen wie CQL, StreaQuel, GSQL, zu denen auch Aurora gerechnet sei
- prozedurale Sprachen wie Hancock

Deklarative Sprachen sind meist von SQL abgeleitet und bieten lediglich weitere Konstrukte zur Verarbeitung von Datenströmen, z. B. Fenstertechniken. Prozedurale Sprachen ähneln in der Regel einer Programmiersprache, im Beispiel von Hancock C. Es werden keine speziellen Operatoren angeboten (für die Datenverarbeitung, z.B. Join, Aggregate), die Behandlung der Daten erfolgt auf elementarere Weise (eben in Prozeduren und Funktionen), jedoch lässt sich die Datenverarbeitung durch benutzerdefinierte Funktionen nahezu beliebig anpassen. Dies jedoch verlangt tiefere Kenntnisse über die Daten, ihre Anwendung und nicht zuletzt auch der Sprache. Der Ansatz von Aurora ist, bezogen auf die Darstellung einer Anfrage, etwas ganz anderes, jedoch genauer betrachtet zeigen die Verwendung elementarer Operatoren (einer Algebra) und die Komposition einer Anfrage, die intern vom System optimiert wird, eine Ähnlichkeit mit deklarativen Ansätzen.

Ein weiteres Unterscheidungskriterium, das zugrundegelegt werden kann ist die hinter der Sprache stehende Semantik. CQL beispielsweise erweitert die Semantik von SQL um einen Zeitbezug und um Datenströme. Verschiedene Funktionen erlauben das Konvertieren von Relationen zu Strömen und umgekehrt, wobei die SQL-typischen Operatoren nur auf Relationen ausgeführt werden können. Die Fenstermechanismen benutzt man in hier in erster Linie um aus Strömen Relationen zu gewinnen (und dann erst blockierende Operationen auszuführen). Dieser Ansatz erscheint etwas unglücklich, da für blockierende Operatoren auf Strömen mehrere Umwandlungen notwendig sind. Aurora, GSQL und StreaQuel arbeiten nur auf Strömen, sie sind sozusagen reine Datenstromsprachen- bzw. Systeme. Fenstermechanismen lösen hier in erster Linie das Problem blockierender Operationen. Hancock liegt in dieser Hinsicht etwas zwischen diesen beiden Ausrichtungen, interpretiert man die verwendeten Signaturen als persistente Relationen, so finden sich auch hier beide Konstrukte. Fenster werden verwendet um Ströme zu durchlaufen und um Events zu erkennen (gibt es in den anderen Ansätzen nicht), nicht um blockierende Operatoren zu ermöglichen.

Aurora wartet als einziger Ansatz hier mit QoS Spezifikationen auf, auch die graphische Anfragesprache hebt es von den anderen Systemen ab. StreaQuel bietet eine Fülle von Fenstertechniken während GSQL seine Fenster aus der zeitlichen Ordnung ausgewählter Attribute erzeugt.

(Tabelle 4.1: Vergleich der wichtigsten Merkmale der vorgestellten Sprachen)

	CQL	Hancock	Aurora	StreaQuel	GSQL
Verwandte Sprache	SQL	C	- (Workflow)	SQL	SQL
Fenstertechnologien	Sliding Windows Zeit- Tupel- Partitions- basiert	Sliding Window	Windowed Operators Slide Tumble Latch Resample	Sliding Windows Umfangreiche Intervallsemantik (Transform..To Snapshop, Landmark, Sliding...)	Keine Sliding Windows (Zeitmarken- basierte "Fenster- berechnung")
Ströme + Relationen	Ja	Jein (Signaturen)	Nein	Nein	Nein
Approximation/Sampling/ Filterung	Nein	Ja	Ja	Nein	Nein
Zeitsemantik	Ja (logisch, physikalisch)	Ja	Ja	Ja (logisch, physikalisch)	Ja (logisch, physikalisch)
Blockierende Operatoren	In Verbindung mit Sliding Windows	Ja	Ja (Sliding Windows oder spezielle Operatoren)	In Verbindung mit Sliding Windows	Ja, über Zeitmarken gesteuert
QoS	Nein	Implementierbar	Ja	Nein	Nein
Persistente Speicherung	Nein	Ja	Temporär	Nein	Nein

5. Vergleich mit Anfragesprachen konventioneller DBMS (SQL)

Hier noch einmal in einem kurzen Überblick ein Vergleich der wesentlichen Merkmale von DBMS und DSMS.

(Tabelle 5.1: Unterschiede DBMS-/DSMS-Anfragesprachen)

	DBMS (SQL)	DSMS
Fenstertechnologien	Nein (erst seit SQL2003)	Ja
Ströme	Nein	Ja

Approximation/Sampling/ Filterung	Nicht erforderlich	Teilweise erforderlich
Zeitsemantik	Nein (nicht implizit)	Ja
Blockierende Operatoren	Kein Problem	Problematisch
QoS	Keine Abstufung, „Full Service“	Teilweise notwendig
„Universalität“, Verbreitung	Ja, hoch	Nein, niedrig
Persistenz	Elementarer Bestandteil	Nur selten

6. Zusammenfassung/Konklusion

DSMS erfordern eine andere Art der Datenverarbeitung, und wenn überhaupt, der Speicherung. Die hier vorgestellten Anfragesprachen bzw. Datenverwaltungssysteme zeigen verschiedene Ansätze wie mit den besonderen Anforderungen umzugehen ist. Einige Aspekte erscheinen dabei in mehreren Systemen sehr ähnlich, während sich andere sehr stark unterscheiden oder nur in einer spezifischen Lösung auftauchen. Die bisher entwickelten Systeme/Sprachen sind oftmals auf eine bestimmte Anwendung oder Klasse von Anwendungen spezialisiert, dies erklärt auch die große Vielfalt. Universelle Ansätze gibt es bisher wenige (am ehesten Aurora und STREAM).
Jede Sprache hat also ihre Besonderheiten. Viele Aspekte sind jedoch allen Systemen gemein, z. B. Fenstertechniken, Stromkonzept und Zeitbezüge. Daraus lässt sich ableiten, dass diese Aspekte für DSMS und ihre Problemstellungen elementar sind.
Ansätze, die wirklich alle Anforderungen in vollem Umfang befriedigen gibt es kaum. Insbesondere mit QoS-Aspekten wird sich sehr wenig beschäftigt (nur in Aurora), offensichtlich ein Anzeichen dafür, dass es bisher in der Praxis keine allzu große Rolle spielt.

7. Quellenangaben

[1]Models and Issues in Data Stream Systems
(Babcock, Brian et. al.)
[2] Hancock: A Language for Extracting Signatures from Data Streams
(2000, Cortes, Corinna, Fisher, Kathleen, Pregibon, Daryl, Rogers, Anne, Smith, Frederick)
[3] The CQL Continuous Query Language: Semantic Foundations and Query Execution
(Arasu, Arvind, Babu, Shivnath, Widom, Jennifer)
[4] Monitoring Streams – A New Class of Data Management Applications
(Carney, Don et. al.)
[5] Scalable Distributed Stream Processing
(Cherniak, Mitch et. al.)

[6] Aurora: A Data Stream Management System

(Abadi, D. et. al.)

[7] Gigascope: A Stream Database for Network Applications

(Cranor, Chuck, Johnson, Theodore, Spataschek, Oliver, Shkapenyuk, Vladislav)

[8] StreaQuel Overview, Language Panel 1st octennial SWiM Meeting, 9. Januar 2003

(Franklin, Mike, UC Berkeley)

[9] Managing Trajectories of Moving Objects as Data Streams
(Patroumpas, Kostas, Sellis, Timos)