Seminar Data Streams - Anfragesprachen
Betreuer: Jürgen Göres

Markus Wörz

14. Juni 2005

Vollautomatische Verkehrsüberwachung, der Supermarkt ohne Kassiererinnen und die Jacke, welche biometrische Daten des Trägers erfasst, alle diese Zukunftsvisionen produzieren enorme Mengen von Sensordaten, welche verarbeitet werden müssen. Diese Datenströme sind mit herkömmlichen Datenbankverwaltungssystemen (DBVS) nur schwer handzuhaben. Neue sogenannte Datenstromverwaltungssysteme (DSVS) übernehmen zunehmend diese Aufgaben. Bei den DBVS hat sich die Anfragesprache SQL weitestgehend durchgesetzt. Bei den DSVS gibt es noch keinen solchen Standard, jedoch hat die Forschung schon verschiedene Lösungen hervorgebracht, die sich aber zum Teil erheblich voneinander unterscheiden. Der folgende Artikel stellt einige dieser Anfragesprachen vor und setzt diese zueinander in Beziehung.

Inhaltsverzeichnis

1 Einführung und Überblick **4**

2 Continuous Query Language (CQL) **6**
2.1 Merkmale und Konzepte 6
2.2 Syntax und Semantik 7
2.3 Beispielanfragen 8
2.4 Bewertung 9

3 Hancock **10**
3.1 Merkmale und Konzepte 10
3.2 Syntax und Semantik 11
3.3 Beispielanfragen 13
3.4 Bewertung 15

4 Aurora **16**
4.1 Merkmale und Konzepte 16
4.2 Syntax und Semantik 18
4.3 Beispielanfragen 19
4.4 Bewertung 20

5 Weitere Anfragesprachen **21**
5.1 Expressive Stream Language (ESL) 21
5.2 GSQL 21
5.3 AQuery 22

6 Fazit **23**

1 Einführung und Überblick

Konventionelle DBVS speichern große Datenmengen, welche vom Benutzer selbst aktualisiert und mittels Anfragen analysiert werden; Hier spielt also der Mensch die aktive Rolle, während die Datenbank passiv bleibt. Im Kontext von Sensordatenströmen, die dem Menschen bestimmte Informationen liefern, ist es genau umgekehrt: Er nimmt hier eine passive Rolle ein und das System selbst verarbeitet die ankommenden Daten, indem es vom Anwender zuvor spezifizierte Anfragen ausführt, welche für ihn nützliche Informationen sichtbar machen. Bei der Formulierung dieser Anfragen verwendet man häufig *blockierende* Operationen, die nicht mit jedem ankommenden Stromtupel sofort ein Ergebnis liefern, sondern welche dazu erst eine gewisse Menge von Stromtupeln, also Stromausschnitte, konsumiert haben müssen. Diese spezifiziert man mit Hilfe von sogenannten *Fenstern*.

Außerdem ist man gezwungen angesichts von möglichen Systemüberlastungen, sei es durch steigende Datenraten in den Eingangsströmen oder durch situationsbedingte Veringerung der Betriebsmittel, auf exakte Anfrageergebnisse zu verzichten und sich stattdessen mit Näherungen zu begnügen. In diesem Zusammenhang wäre es durchaus interessant, wenn man dem System durch eine Art Priorisierung mitteilen könnte, an welchen Stellen eine Näherung die Ergebnisse weniger negativ beeinflusst und wo es auf Genauigkeit ankommt.

All diesen Anforderungen muss sich eine Sprache zur Anfrageformulierung im Datenstromkontext stellen und dabei möglichst intuitiv, leicht verständlich, ausdrucksmächtig und universell einsetzbar sein. Ferner müssen auch die Resultate von gestellten Anfragen nach Möglichkeit schnell zur Verfügung stehen. Mittlerweile gibt es dazu eine Reihe von unterschiedlichen Ansätzen für solche Sprachen, die man verschiedenen Kategorien zuordnen kann. Zum einen gibt es *deklarative* Sprachen, bei denen man in erster Linie angibt, *was* das Ergebnis einer Anfrage sein soll, und das System den besten Weg zur Lösung der Aufgabe ermitteln lässt. In dem *imperativen* Paradigma hingegen steht im Vordergrund, *wie* man zu der Lösung kommt, dabei gibt man dem System also detailiert die einzelnen Verarbeitungsschritte vor, welche zur Lösung nötig sind. Die dritte Kategorie bilden Sprachen, bei denen man Anfragen *graphorientiert* spezifizieren kann, was einen Mittelweg zu den beiden vorhergehenden Paradigmen darstellt.

In den folgenden Kapiteln soll von jeder Gattung ein prominenter Vertreter exemplarisch präsentiert werden. Der Rest dieser Ausarbeitung ist folgendermaßen gegliedert: Abschnitt 2 gibt es einen Überblick über die deklarative Sprache *CQL*, welche SQL aufsetzt. Im Anschluss folgt eine Vorstellung der Konzepte von *Hancock*, welche eine Erweiterung von C darstellt, und wie dieses eine imperative Sprache ist. Danach wird noch die Anfragestellung im *Aurora*-System behandelt, welches Ströme und Operatoren in einem Graphen modelliert.

Jedes dieser Systeme wird zunächst grob vorgestellt mit seinen Konzepten und Merkmalen, worauf dann eine genauere Darstellung der syntaktischen Elemente und deren Bedeutung folgt. Daraufhin wird der Einsatz der Sprache anhand eines durchgehenden Beispielszenarios verdeutlicht. Es handelt sich dabei um eine Anwendung aus dem Mobilfunkbereich, dem Spezialgebiet von Hancock, und findet sich in leicht abgewandelter Form in [7] wieder. Betrachtet wird ein Datenstrom `CallStream`, der eine Überwachung von Telefongesprächsdaten darstellt und dessen Elemente jeweils drei Attribute besitzen. In dem Attribut `origin` ist die

Nummer des Anrufers vermerkt, `destination` beinhaltet die angerufene Telefonnummer und `isInternational` sagt etwas darüber aus, ob es sich um ein Gespräch über Ländergrenzen hinweg gehandelt hat. Hier nochmals ein Tupel des Stroms schematisch dargestellt:

```
CallStream
(
  origin: string                // Nummer des Anrufers
  destination: string           // Nummer des Angerufenen
  isInternational: boolean      // internationales Gespräch?
)
```

Die Aufgabe, welche nun jedes System leisten muss, ist anhand der ankommenden Stromtupel nach Anrufernummern (`origin`) gruppiert internationale Telefongespräche zu zählen. Das Ergebnis soll also eine Liste aller Nummern von Anrufern sein zusammen mit der Anzahl der von ihr aus geführten Ferngespräche. Da diese Anfrage nicht alle Möglichkeiten der Sprachen ausnutzt, werden für die einzelnen Sprachen zusätzlich ergänzende Beispiele vorgestellt. Nach den drei Vertretern der verschiedenen Sprachkonzepte wird ein Überblick über weitere Anfragesprachen gegeben und kurz auf etwaige Besonderheiten eingegangen. Ein übergreifendes Fazit bildet dann den Abschluss dieser Ausarbeitung.

2 Continuous Query Language (CQL)

Wie in [1] beschrieben ist CQL fast vollständig im Datenstromverwaltungssystem *STREAM*[1] umgesetzt und stellt lediglich eine Weiterentwicklung der weitverbreiteten Sprache SQL dar. Damit soll das Erlernen der Continuous Query Language zumindest für Kenner des Relationenmodells möglichst einfach sein, ohne dass man auf Ausdrucksmächtigkeit verzichten müsste. Denn auch wenn CQL als Ganzes komplex ist, so wird zumindest die Syntax und Semantik des relationalen Teils schon allgemein gut verstanden.

2.1 Merkmale und Konzepte

In CQL gibt es neben den bekannten Relationen nun auch Ströme, welche ebenfalls eine Menge von benannten Attributen haben. Diese werden im sogenannten Stromschema definiert, welches ganz analog zum Schema einer Tabelle im Relationenmodell aufgebaut ist. Die Zeit wird in CQL mittels einer diskreten, geordnete Menge modelliert. Dabei muss man als Elemente nicht unbedingt der Systemuhr entnommene Zeiten wählen, obwohl das häufig (wie auch im Folgenden) so gemacht wird. Man könnte auch genauso gut die natürlichen Zahlen als Zeitstempel verwenden und beispielsweise mit der Null den frühestmöglichen Zeitpunkt repräsentieren. Einen Strom definiert man damit als eine potentiell unendliche *Multimenge* von Tupeln aus der Attributmenge des Stroms, welche jeweils mit einem Zeitstempel versehen sind. Diesen vergibt das System bei Ankunft des Stromtupels, wobei die vergebenen Zeitstempel mit Ablauf der Zeit nie kleiner werden. Zu beachten ist dabei, dass dieser nicht Teil des Stromschemas ist und durchaus mehrere Elemente des Stroms den gleichen Zeitstempel haben können. Es wird lediglich verlangt, dass ein Zeitstempel von maximal endlich vielen Stromelementen getragen wird.

Die Definition einer *Relation* in CQL weicht erheblich von der Variante ab, wie man sie im normalen Relationenmodell antrifft. Sie ist hier zu sehen als eine zeitvariante Multimenge (time-varying bag), genauer gesagt als eine *Funktion* vom diskreten Zeitbereich in eine Multimenge von Tupeln aus dem Schema der Relation. Das bedeutet, dass wir jedem Zeitstempel eine mit Einträgen gefüllte Tabelle zuordnen können. Damit ist man in der Lage von der Relation zum Zeitpunkt t_n zu sprechen, deren Inhalt sich möglicherweise gegenüber dem Zeitpunkt t_{n-1} geändert hat. Das ist auch wichtig im Datenstromkontext, denn hier kommen in der Regel ständig neue Tupel zur Verarbeitung und ändern damit potenziell auch die Inhalte der Relationen. Eine klassische Relation R des bekannten Relationenmodells, welche sich nicht mit der Zeit ändert und auch keine Multimenge darstellt, lässt sich in CQL durch eine konstante Funktion nachbilden, welche für jede Eingabe – also jedes Element des diskreten Zeitbereichs – R zurückgibt, wobei R selbst wieder eine Multimenge ist

[1] STREAM steht für STanford stREam datA Manager und ist ein Projekt der Universität von Standford. Mehr Informationen dazu finden sich unter http://www-db.stanford.edu/stream

2.2 Syntax und Semantik

In CQL gibt es drei Klassen von Operatoren: Die sogenannten Strom-zu-Relation-Operatoren (*stream-to-relation* operators) nehmen als Eingabe einen Strom und liefern eine Relation als Ausgabe; Die Relation-zu-Strom-Operatoren (*relation-to-stream* operators) leisten die Konvertierung in die umgekehrte Richtung. Außerdem existieren viele Operatoren zwischen Relationen (*relation-to-relation* operators), deren Syntax wie Semantik sich an den aus SQL bekannten orientiert; Auf diese soll daher im Folgenden nicht mehr näher eingegangen werden. Man beachte, dass keine Operatoren existieren, welche direkt auf Strömen arbeiten, diese können aber mittels der anderen drei simuliert werden. Abbildung 2.1 veranschaulicht den Zusammenhang.

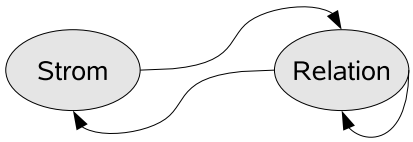

Abbildung 2.1: Die Konvertierungsoperationen in CQL

Die Strom-zu-Relation-Operatoren arbeiten allesamt nach einer Fenstersemantik, was bedeutet, dass sie stets einen Ausschnitt des Stroms als Ergebnis in Form einer Relation zurückgeben. Dabei gibt es wiederum drei verschiedene Varianten: Beim *zeitbasierten* Fensteroperator `[Range T]` spezifiziert man die Dauer, über die eingehende Daten in der Ausgaberelation gehalten werden. Zum Beispiel legt man mit `MyStream [Range 10 Seconds]` fest, dass jeweils nur die Tupel des Stromes `MyStream` in der Ausgabe erscheinen sollen, die innerhalb der letzten 10 Sekunden im System angekommen sind.

Alternativ dazu lässt sich in CQL ein Fenster aber auch *tupelbasiert* definieren. Mittels `[Rows N]` erreicht man, dass jeweils die letzten maximal `N` Tupel in der Ausgabe auftauchen sollen, also die mit den größten Zeitstempeln. Diese Operation ist nützlich bei Strömen, deren Volumen zeitabhängig stark schwankt, denn so ist man sicher, dass man immer genügend aber auch nicht zu viele Einträge in der Ausgaberelation hat, die man vielleicht zur sinnvollen Weiterverarbeitung benötigt. Allerdings kann eine tupelbasierte Fenstergröße auch zu starken Schwankungen in der durch das Fenster repräsentierten Zeitspanne führen, was zu Verwirrung führen könnte. An dieser Stelle ist es wichtig zu erkennen, dass derartige tupelbasierte Operationen nicht immer deterministisch sind. Man betrachte folgendes Szenario für `N=5`, wobei S ein Strom und s_i ein Element von S, also ein Tupel des Stroms sei:

$$S = [\ldots \langle s_0, 6\rangle, \langle s_1, 12\rangle, \langle s_2, 12\rangle, \langle s_3, 13\rangle, \langle s_4, 13\rangle, \langle s_5, 14\rangle, \langle s_6, 16\rangle]$$

Hier ist nicht eindeutig bestimmt, ob s_1 oder s_2 im Ausgabefester erscheinen soll, denn beide haben den gleichen Zeitstempel 12. Der Konflikt wird in CQL durch Zufall entschieden und die Ergebnisrelation R könnte beispielsweise so aussehen:

$$R = [\langle s_1, 12\rangle, \langle s_3, 13\rangle, \langle s_4, 13\rangle, \langle s_5, 14\rangle, \langle s_6, 16\rangle]$$

Die letzte Fensteroperatorvariante ist etwas komplexer. Sie wird angewendet, wenn aus einem Strom eine Relation erzeugt werden soll, die bezüglich der vorkommenden Wertekombinationen einer Attributmenge ausgewogen ist. Ein Anwendungsszenario hierfür wäre ein Strom von Personendaten `PersonStream`, bei dem unter anderem der Familienstand (`maritalStatus`) und das Geschlecht (`gender`) festgehalten ist.

Der Wertebereich von `maritalStatus` ({ledig, verheiratet, geschieden, verwitwet}) hat die Kardinalität vier, der von `gender` zwei, also gibt es potenziell acht verschiedene Kombinationsmöglichkeiten von Werten. Möchte man nun eine Relation erstellen, in der maximal zwei Repräsentanten jeder Kombination vorhanden sind, dann kann man dies durch ein sogenanntes *partitioniertes* Fenster erreichen, nämlich mit `PersonStream [Partition By maritalStatus, gender, Rows 2]`. Anschaulich gesprochen legt dieser Operator also für jede Kombination der spezifizierten Attributwerte einen Behälter der in der `Rows`-Klausel angegebenen Größe an. Neu ankommende Stromtupel werden entsprechend in die Behälter einsortiert und dabei gegebenenfalls das Element mit dem ältesten Zeitstempel verdrängt. Das eigentliche Ergebnis des Operators ist dann die Vereinigung der Behälterinhalte. Dabei können wir nicht garantieren, dass jeder Behälter voll ist; Das passiert nur dann, wenn jede Attributkombination wenigstens so oft im Strom vorkam, wie die fiktiven Behälter Werte fassen.

Nun gibt es wie bereits erwähnt, nicht nur Operatoren, welche Ströme in Relationen umwandeln, sondern auch die dazu komplementären. Hier bietet CQL wiederum drei verschiedene Varianten an: `Istream` (insert stream) fügt immer dann ein neues Tupel in den Ausgabestrom ein, wenn es in der Relation selbst neu hinzugekommen ist. Rufen wir uns dabei nochmals die Definition der Relation als Funktion in Erinnerung. Dann bedeutet ein Hinzukommen dabei, dass das Element in der Relation zum vorhergehenden Zeitstempel noch nicht enthalten war und in der Relation zum aktuellen Zeitpunkt vorhanden ist.

Bei `Dstream` (delete stream) hingegen sorgt die Löschung eines Tupels in der Eingaberelation für das Auftauchen im Ausgabestrom. Die letzten Variante `Rstream` (relation stream) speist einen Strom aus der Eingaberelation derart, dass sie alle Tupel, die sich zum augenblicklichen Zeitstempel in der Relation befinden in die Ausgabe leitet. Veranschaulicht an einer Relation, die immer drei Einträge besitzt, für jeden Zeitstempel also gleich aufgebaut ist, bedeutet dies, dass zu jedem Zeitpunkt in der diskreten Menge aus Zeitstempeln eben diese drei Elemente in den Ausgabestrom fließen.

2.3 Beispielanfragen

Kommen wir nun auf das in der Einleitung geschilderte Beispiel zurück. In CQL würde man den Eingabestrom `CallStream` zunächst mittels eines Fensters unendlicher Größe in eine Relation umwandeln. Dies geschieht durch `[Range Unbounded]`. Dabei will man jedoch nur internationale Telefonate im Ergebnis haben, was sich durch eine Selektion erreichen lässt, in der man `isInternational = 1` als Bedingung stellt. Dann gruppiert man die Relation noch mittels `GROUP BY` anhand der Ausgangsnummer, also von `origin` und gibt diese zusammen mit der Anzahl ihrer Telefonate zurück, welche man mit der Aggregatfunktion `COUNT` ermittelt. Die vollständige Anfrage sieht so aus:

```
SELECT origin, COUNT(destination)
FROM CallStream [Range Unbounded]
WHERE isInternational = 1
GROUP BY origin
```

Dabei ist klar, dass die Ergebnisrelation nach dem Modell von CQL auch von der Zeit abhängig ist und sich mit dem Ankommen neuer Tupel verändert. Schwierig dabei ist, dass das Datenstromverwaltungssystem zwar nicht den kompletten Strom, aber dennoch eine potenziell unbeschränkte Menge von Telefonnummern speichern muss wegen des unendlichen Fensters.

Im nächsten Beispiel ist dies nicht mehr so, hier wird gezeigt wie man Ströme in Form von Relationen bearbeitet und wieder als Ströme ausgibt. Gesetzt den Fall, dass wir unseren `CallStream` nach Inlandsgesprächen filtern wollten, ihn aber dennoch als Strom weiterleiten wollten, so könnten wir das mit dieser Anfrage erreichen:

```
SELECT Rstream(origin, destination)
FROM CallStream [Range Now]
WHERE isInternational = 0
```

Hier spezifizieren wir mit `[Range Now]` lediglich die Tupel mit dem aktuellen Zeitstempel, die wir auch sofort mittels `Rstream`, welcher ja einen Strom aus den in der Relation aktuell enthaltenen Elementen macht. An dieser Stelle fällt auf, dass es recht umständlich ist, in CQL lediglich einen Filter auf einem Strom zu definieren, denn man muss in Ermangelung von Operatoren auf Strömen immer den Umweg über Relationen gehen.

Auf der anderen Seite bringt die Unterstützung von Relationen, die man als Spezialisierung der Vertreter im Relationenmodell sehen kann, auch große Vorteile mit sich. Beispielsweise könnte man obige Anfrage mit Hilfe einer Relation, die statische Telefonbenutzerinformationen speichert, etwa nach dem Schema `PhoneUser (phoneNumber, name)`, derart erweitern, dass man den Strom mit dem Namen des jeweiligen Anrufers anreichert, wie in folgender Anfrage:

```
SELECT Rstream(origin, destination, name)
FROM CallStream [Range Now], PhoneUser
WHERE isInternational = 0
AND origin = phoneNumber
```

2.4 Bewertung

Anhand der Beipiele wird deutlich, dass Leute, die SQL beherrschen, mit CQL ohne große Einarbeitungszeit, ja fast intuitiv, ihre eigenen Anfragen formulieren können. Die Erweiterungen beschränken sich auf eine Minimum und fügen sich nahtlos in die bestehende Syntax ein. Die Unterstützung von Relationen ist von großem Vorteil, insbesondere wenn man Ströme mit statischen Daten verknüpfen möchte wie im letzten Beispiel.

Das deklarative Paradigma hat aber auch seine Schwächen: Die Anfragen sind relativ einfach zu stellen und der Benutzer muss sich kaum Gedanken dazu machen, wie das System diese letztendlich umsetzt. Dieser Komfort kann jedoch dazu führen, dass Benutzer Anfragen stellen, die viele Betriebsmittel in Anspruch nehmen, wie etwa die Verwendung eines unendlich großen Fensters. Zusätzlich stellt CQL enorme Anforderungen an den Optimierer, denn die Anfragen haben zum Teil auch enormes Einsparungspotenzial, welches man unbedingt nutzen sollte.

3 Hancock

Die nächste Sprache schafft hier deutlich mehr Transparenz auf Seite des Benutzers und überträgt diesem dadurch auch mehr Verantwortung. Die Ausdrucksmächtigkeit, welche der Anwender dadurch erhält, hat aber auch eine deutlich höhere Komplexität zu Folge.

3.1 Merkmale und Konzepte

Wie in [6] nachzulesen, ist Hancock eine imperative Sprache zur Formulierung von Anfragen auf Datenströmen, welche im Telekommunikationsbereich von der Firma AT&T entwickelt worden ist. Zuvor hatte man versucht die enormen Datenmengen, wie sie bei der Erfassung von Mobilfunkdaten anfallen, mit hocheffizienten *C-Programmen* zu verarbeiten. Die sehr gute Leistungsfähigkeit hatte man jedoch mit hoher Komplexität und somit schlechter Wartbarkeit des Quelltextes erkauft, sodass die teueren Änderungen und Erweiterungen, wie sie zum Beispiel durch Gesetzesänderungen erforderlich werden, die Entwicklung einer neuen Lösung nötig gemacht haben. So hat man das bestehende C erweitert und damit Hancock[1] geschaffen.

Mittels dieser Sprache lassen sich Konstrukte, die im Telekommunikationsbereich häufig vorkommen, einfacher formulieren als mit herkömmlichem C-Programmen, wobei man keinesfalls auf die gute Performance verzichten muss. Dabei geht es also allein darum, den Programmierer zu entlasten und ihn nicht jedesmal das Rad neu erfinden zu lassen, sondern ihm eine Auswahl von Sprachelementen zur Verfügung zu stellen, die es ihm erlauben typische Aufgaben bei der Datenstromverarbeitung elegant und mit wenig Aufwand zu lösen.

Was man hier beispielsweise häufig benötigt, sind sogenannte *Signaturen*. Dabei handelt es sich um Datenobjekte, welche dazu verwendet werden, Daten über einen Strom hinweg zu aggregieren. Sie sagen somit etwas über diesen Datenstrom aus, beispielsweise könnte man in einer Signatur den Durchschnittswert und das Maximum eines Stromattributs festhalten, wobei klar ist, dass diese Signatur unter Umständen mit jedem Ankommen eines neuen Stromtupels, ihren internen Zustand – in diesem Fall wahrscheinlich den Durchschnittswert – aktualisieren muss.

Signaturen haben verschiedene *Repräsentationen*. Die *logische* Sicht ist die für den Programmierer am einfachsten verständliche, jedoch ist sie nicht immer platzsparend und effizient. Um dieses zu erreichen, hat man zusätzlich noch eine *approximative* Darstellung einer Signatur, in der beispielsweise eine als Gleitkommazahl gespeicherte Wahrscheinlichkeit[2] mit nur vier Bits kodiert wird. Desweiteren existiert noch eine *physikalische* Repräsentation, in welche man die Daten zur Verringerung des Eingabe-/Ausgabe-Aufwands transformiert, um Signaturen auf der Platte oder sonstigen Medien abzuspeichern. Im Wesentlichen erfolgt hier eine im Gegensatz zur approximativen Darstellung verlustfreie Komprimierung. Die Algorithmen zur Konvertierung zwischen den einzelnen Darstellungen lassen sich auch in Hancock spezifizieren und sind im Grunde genommen C-Funktionen.

[1] Gültige C-Programme sind also auch Hancock-Programme. Es gibt einen Präprozessor, der Hancock-spezifische Konstrukte in C-Code umwandelt.

[2] Diese kann ja nur Werte zwischen Null und Eins annehmen

Wichtig im Umfeld der Telekommunikation ist auch die Behandlung von bestimmten *Ereignissen*, die bei der Analyse eines Stromes auftreten können. Als Beispiele seien hier das Aufdecken von Betrug oder sonstigen Anomalien im Mobilfunkverkehr genannt; Auch hierfür bietet Hancock Mechanismen. Im Folgenden werden all diese Konstrukte in Hancock genauer vorgestellt und anhand konkreter Beispiele erläutert.

3.2 Syntax und Semantik

Signaturen definiert man wie von C gewohnt mittels eines `structs`, welches man mit `typedef` benennt, so würde beispielsweise

```
typedef struct
{
  float average;
  int maximum;
}
mySignature;
```

eine neue Signatur mit dem Bezeichner `mySignature` einführen, die zwei Felder beinhaltet, um etwa den Durchschnitt und das Maximum eines Stromes aus Integer-Werten zu speichern. Natürlich benötigt man nicht immer ein `struct`; Wollte man lediglich einen einfach Wert, wie den Durchschnitt speichern, so könnte man auch einen einzelnen `float` nehmen. Signaturen werden in Hancock oft in sogenannten *Maps* gespeichert, das sind Datenstrukturen, in denen man Objekte (in unserem Fall Signaturen) unter einem bestimmten Schlüssel ablegen kann, um anschließend schnell darauf zugreifen zu können. Da die Werte in der Map eventuell auf den Externspeicher geschrieben werden, will man genau hier bei Bedarf eine Komprimierung der Daten vornehmen, bevor das passiert. Dehalb lassen sich bei der Definition einer Map auch optional Funktionen angeben, welche die Kompression der Datensätze beim Schreiben auf den Externspeicher und auch die Dekompression beim Zugriff leisten. Die Map beinhaltet also Mechanismen, um Daten von der logischen in die physikalische Sicht zu konvertieren und umgekehrt. Folgender Beispielcode illustriert die Deklaration einer Map, die zu einer gegebenen Telefonnummer den Namen des jeweiligen Besitzers speichert:

```
map PhoneNumberToOwnerMap
{
  key (MINVALPN .. MAXVALPN); // Die Telefonnummer ist Schlüssel zu einem
  value phoneOwner;           // String, dem Namen des Besitzers.
  default '\0';               // Wenn zu einer Nummer kein Eintrag existiert,
                              // wird der leere String zurückgegebn.
  compress stringSqueeze;     // Komprimierungsfunktion
  decompress stringUnsqueeze; // Dekomprimierungsfunktion
};
```

Hier wird also ein Map-Typ namens `PhoneNumberToOwnerMap` deklariert, welcher Datensätze vom Typ `phoneOwner` unter einem Schlüssel zugreifbar macht, der wie alle Schlüssel in Hancock vom Typ `long long` ist. Die Konstanten `MINVALPN` und `MAXVALPN` definieren dabei den Bereich der gültigen Schlüssel und neben `compress`, bzw. `decompress` sind Bezeichner von

C-Funktionen angegeben, welche bei Bedarf das Packen und Entpacken der Datensätze vom Typ `phoneOwner` vornehmen.

Um auf die Map zuzugreifen, benutzt man den `<: :>`-Operator. Er funktioniert im Prinzip genauso wie der aus C bekannte Arrayzugriff mit `[ ]`. Die oben definierte Map könnte man etwa so verwenden:

```
PhoneNumberToOwnerMap map;
phoneNumber number = 63122708;
phoneOwner owner = "Schmidt, Klaus";

// Lege die Daten in der Map ab.
map<:number:> = owner;

...

// Hole sie mittels des Schlüssels wieder heraus.
phoneOwner restoredOwner = map<:number:>;
```

Aber nicht nur in der Map finden sich bei Hancock Unterstützung für Konvertierungsfunktionen, sondern auch im `view`-Konstrukt. Hierin findet sich ein Mechanismus zur Transformation zwischen der logischen und der approximativen Sicht. Dabei spezifiziert man die Datentypen der verschiedenen Repräsentationen und die Funktionen, welche zur Umwandlung benötigt werden. Im folgenden Beispiel wird eine Sicht definiert, die es erlaubt eine Zeitspanne einmal genähert (als `bin` bezeichnet) und einmal exakt zu betrachten.

```
view time(bin, minute)
{
  char <=> int;
  bin(m) { return minutesToBinary(m); }
  minute(b) { return binaryToMinutes(b); }
}
```

Um nun diese Sicht richtig zu verwenden, setzt man den `view`-Operator `$` ein. Im untenstehenden Ausschnitt werden zunächst zwei Variablen deklariert, einmal die für die approximative Darstellung, also letztendlich ein `char`, und im zweiten Fall die genaue Repräsentation als `int`; Die erste Variable wird mit `3` initialisiert. Daraufhin wird der Inhalt dieser Variablen mittels `$` umgewandelt und das Ergebnis an die zweite Variable zugewiesen. Später folgt das Gleiche in umgekehrter Richtung. Vorausgesetzt die Konvertierungsfunktionen sind geeignet definiert, so hat die Variable `b` am Ende den gleichen Wert wie nach ihrer Initialisierung.

```
bin b = 3;      // Approximierte Darstellung
minute m;       // Exakte Darstellung
m = b$minute;
...
b = m$bin;
```

Eine weitere wichtige Zielsetzung für Hancock ist die Ereignisbehandlung. Dafür schreibt man eine sogenannte *Ereigniserkennungsfunktion*, welche die Tupel eines Stroms in einem definierbaren Fenster analysiert und die dabei aufgetretenen Ereignissen als Menge zurückliefert.

Diese Ereignismenge wird durch eine sogenannte *multi-union* modelliert, welche ähnlich einer Struktur in C eine Menge von mit Marken versehenen Ereignistypen beinhaltet.

```
munion PhoneEvents
{:
  CallStreamElem call,          // Ein Telefonanruf wurde getätigt.
  PhoneNumber newOriginNumber, // Die jetzige Telefonnummer unterscheidet
                                // sich von der vorhergehenden.
  PhoneNumber endOriginNumber  // Die jetzige Telefonnummer unterscheidet
                                // sich von der nachfolgenden.
:};
```

Dieser Quelltextausschnitt zeigt die Definition eines multi-union-Typs, der zur Ereignisbehandlung bei dem in der Einleitung vorgestellten Telefonbeispiel dient. Die Annahme ist hierbei, dass bei der Verarbeitung des Stroms `CallStream` ein Anruf selbst ein Ereignis ist und auch die Änderung der Nummer des Anrufers (`origin`). Ein Ereignis vom Typ `PhoneEvents` speichert also die *Vereinigung* der möglichen Ereignisse. Gesetzt den Fall eine Ereigniserkennungsfunktion wollte durch ihren Rückgabewert mitteilen, dass ein Anruf passiert ist und sich (gleichzeitig) die jetzige Nummer des Anrufers von der vorangehenden unterscheidet, dann würde sie ein Objekt vom Typ `PhoneEvents` konstruieren, in welchem lediglich die entsprechenden Attribute gesetzt sind. Das geschähe mittels `{: call = myCall, newOriginNumber = myNumber :}`, wobei `myCall` vom Typ `CallStreamElem` und `myNumber` vom Typ `PhoneNumber` ist. Man erlaubt deshalb eigene Typen bei Attributen der multi-union, damit man zusätzliche Informationen in einem Ereignis speichern kann, die dessen Behandlung erleichtern. Bei einem `call` wird im `CallStreamElem` beispielsweise das ganze Stromtupel, welches den Anruf repräsentiert, gespeichert.

An der Aufrufstelle der Ereigniserkennungsfunktion werden die Ereignisse abgefangen und eigens dafür geschriebenen Funktionen übergeben, die eventuell weitere Aktionen durchführen. Es gibt also für jedes mögliche Ereignis eine Routine, die es behandelt. Auf eine genauere Beschreibung sei an dieser Stelle verzichtet, im nächsten Abschnitt macht das Eingangsbeispiel mit der Zählung der Telefongespräche die Zusammenhänge deutlicher.

Um dieses Beispiel zu verstehen, benötigt man noch das `iterate`-Konstrukt, welches den mittels `over` angegebenen Stromausschnitt durchläuft und dabei auf jedes Tupel die Funktion zur Ereigniserkennung anwendet, welche mit `withevents` angegeben wird. Zusätzlich lassen sich einzelne Stromtupel herausfiltern, das bedeutet sie kommen gar nicht in die Ereignisbehandlung. Dies geschieht durch `filteredby`. Da es sich lediglich um einen Ausschnitt eines Stromes handelt, ist man in der Lage hier diesen mittels einer `sortedby`-Klausel nach einer Menge von Attributen zu sortieren.

3.3 Beispielanfragen

In dem Telefonbeispiel in Abbildung 3.1 durchlaufen wir den Stromausschnitt von Anrufen und sortieren diese nach den Nummern der Anrufer. Dabei berücksichtigen wir nur internationale Telefongespräche. Die Ereigniserkennungsfunktion `originDetect` nimmt dabei ein Fenster des Eingabestromausschnitts und sorgt dafür, dass Events generiert werden, die dann hier im Beispiel dazu benutzt werden, die Statistik, also die Signaturen in der Map, zu aktualisieren. Man beachte, dass die Sortierung des Stromausschnitts und die Definition der Ereignisbehandlung

```
map PhoneNumToFrequencyMap {
  key (MINVALPN .. MAXVALPN-1);
  value int;
  default 0;
};

void sig_main( CallStream stream <currentCall:>,
               PhoneNumToFrequencyMap map <I:>)
{
  // Variable, die die Anzahl der Anrufe der aktuellen Nummer speichert.
  int numCurrentCalls;

  iterate ( over stream
            sortedby origin
            filteredby(currentCall) (currentCall->isInternational)
            withevents originDetect )
  {
    // Hat sich mit dem aktuellen Anruf die Anrufernummer geändert?
    // => Zähle dessen internationale Anrufe.
    event newOriginNumber(PhoneNumber phoneNumber) {
      numCurrentCalls = 0;
    }

    // Ein internationaler Anruf passiert? => Zähler inkrementieren.
    event call() {
      numCurrentCalls++;
    }

    // Sind wir in dem Strom vorerst fertig die Anrufe von dieser
    // Nummer aus zu zählen?
    // => Addiere die Anzahl der Anrufe zum Eintrag in der Map.
    event endOriginNumber(PhoneNumber number) {
      map<:number:> = numCurrentCalls + map<:number:>;
    }
  };
}
```

Abbildung 3.1: Die Umsetzung der Telefonanfrage in Hancock

in dieser Weise lediglich vorgenommen wird, um den Zugriff auf die Map zu optimieren. Die Map beansprucht unter Umständen nämlich sehr viel Platz und kann daher nicht vollständig im Hauptspeicher gehalten werden. Folglich müssten bei wahllosem Zugriff auf einzelne Werte der Map mit zunehmender Häufigkeit Seiten von der Platte in den Speicher geladen werden.

3.4 Bewertung

Die Verwandtschaft von Hancock mit der Programmiersprache C ist gut zu erkennen. Zwar hat man die Programmierung von grundlegenden Datenstromverarbeitungen sehr vereinfacht, indem man Konstrukte eingeführt hat, die man in diesem Anwendungsbereich häufig braucht. Jedoch wurde bei Weitem nicht der Abstraktionsgrad erreicht, den CQL bietet; Ein Laie wird erhebliche Probleme haben zu verstehen, wie geschriebene Hancock-Programme arbeiten, wohingegen er bei einer in CQL formulierten Anfrage viel schneller eine Idee von deren Funktionsweise bekommt. Und genau diese Tatsache ist ein Nachteil, der bei interdisziplinären Projekten besonders zum Tragen kommt, wenn beispielsweise Verkehrsplaner mit einem DSVS den Straßenverkehr analysieren wollen. Auf der anderen Seite muss man natürlich auch die Vorteile sehen: Nicht nur die Geschwindigkeit und Leistungsfähigkeit von Hancock, sondern auch die Möglichkeiten C-Programme und Bibliotheken einzubinden, schaffen grenzenlose Freiheit bei der Lösung beliebig komplexer Probleme.

4 Aurora

Die Sprache des *Aurora*-Systems hat genau hier ein gewisses Defizit gegenüber Hancock, jedoch bietet das zunächst einfach scheinende Konzept auch genügend Möglichkeiten, die für die meisten Anwendungen ausreichen dürften.

4.1 Merkmale und Konzepte

Genau genommen bezeichnet Aurora keine Anfragesprache, sondern ein Datenstromverwaltungssystem, welches in einer Kooperation der *Brown Universität*, der *Brandeis Universität* und des *M.I.T.* entwickelt worden ist. Eine knappe Einführung in Aurora bietet [5]. Es besteht aus einem Laufzeitsystem und einer grafischen Oberfläche, die es erlaubt, Anfragen auf Datenströmen visuell mittels Diagrammen zu spezifizieren. Man formuliert also weder SQL-ähnlichen Text noch schreibt man anderweitigen Programmcode, sondern definiert Anfragen allein durch die Anordnung und das Verbinden einzelner Komponenten zu einem gerichteten akzyklischen Graphen. Dessen Knoten stehen zum einen für Eingabeströme in das System, die sogenannten *Datenquellen*. Sie besitzen wie alle Ströme in Aurora eine eindeutige Identifikationsnummer und repräsentieren etwa einen Netzwerkport oder einen Sensor. Um die daraus entstehenden Daten sinnvoll nutzen zu können, leitet man diese Ströme in einen Teilgraphen von Operatoren, welche auf diesen Daten arbeiten und durch Kästchen in der Benutzeroberfläche dargestellt werden. Operatoren sind also auch Knoten des Graphen. Die Kanten dazwischen werden mit Pfeilen dargestellt und stehen für den Datenfluss zwischen den Knoten.

Zusammengefasst ist das so gebildete Stromnetzwerk in Aurora ein azyklischer Graph bestehend aus Datenquellen, die mit Pfeilen über Operatorknoten in Ausgabeströme münden. Unter einer Anfrage versteht man dabei einen Teilgraphen des Stromnetzwerks, welcher einen einzigen Ausgabestrom hat - das Anfrageergebnis. Dabei unterscheidet [2] drei Klassen: *Dauerhafte* Anfragen sind von Anfang an im System und liefern Ergebnisse. Tupel müssen nicht zwischengespeichert werden und können aus dem System gelöscht werden, sobald sie den kompletten Anfragepfad durchlaufen haben. Diese Art von Anfrage wird in Aurora mittels eines Teilgraphen aus Operatoren realisiert, welcher von ein oder mehreren Eingabestömen gespeist wird und in eine angeschlossene Anwendung mündet. Dagegen sind *Sichten* Zweige eines Netzwerks, an die noch keine Anwendungen angeschlossen sind, die aber trotzdem schon Ergebnisströme liefern und eventuell auch eine *Dienstgüte*-Spezifikation besitzen, die Auskunft darüber gibt, was für den Systemadministrator Leistung in diesem Teilnetz ausmacht.

Desweiteren gibt es noch *Ad-Hoc*-Anfragen zu deren Realisierung Aurora die Möglichkeit bietet, den Graphen an so genannten *connection points* (Verbindungspunkten) jederzeit zu erweitern. Diese Verbindungspunkte besitzen lokale Warteschlangen, in denen die ankommenden Stromtupel für eine bestimmte Zeit zwischengespeichert werden können, um somit historische Daten für neue Anfragen vorzuhalten. Von einer *Ad-Hoc*-Anfrage spricht man dann, wenn man zur Laufzeit einen Verbindungspunkt um einen Pfad mit Operatoren und einem Anwendungsprogramm am Ende erweitert.

Konzeptionell gesehen ist ein Strom ähnlich wie bei CQL eine potenziell unendliche Folge von Tupeln mit gleichem Attributschema, die jeweils mit einem Element einer Indexmenge versehen sind. Diese muss eine totale Ordnung besitzen und mit einer Maßeinheit verbunden werden können. Damit ist es möglich auf der Indexmenge, die oft aus Zeitstempeln besteht, Vergleiche, arithmetische Operationen wie Modulus und Abstandsberechnungen durchzuführen.

Wie in [4] beschrieben, lassen sich für die einzelne Anfrageteilgraphen Qualitätsanforderungen festlegen, wie beispielsweise maximale Reaktionszeit oder Genauigkeit der Ausgabe. Somit erlaubt es Aurora dem Benutzer präventiv dafür zu sorgen, dass das System im Überlastfall an den richtigen Stellen Einsparungen vornimmt und für ihn wichtige Berechnungen weiterhin kompromisslos durchgeführt werden. In diesem Zusammenhang stellt [2] drei Typen von Graphen vor, welche der Administrator für jede Ausgabe eines Operators angeben kann. Mit dem Ersten teilt er dem System mit, welcher Dienstgüte er eine bestimmte *Auslassungsrate* bemisst. Der Wert liegt dabei zwischen Null und Eins, wobei höhere Werte bessere Qualität bedeuten. Würde er hier jeder Auslassungsrate Eins zuordnen, also eine Konstantenfunktion angeben, so liese das System ohne Bedenken Tupel aus, sobald die Notwendigkeit dafür bestünde. Der zweite optionale Graph dient dem System als Richtline für die Bedeutsamkeit der verschiedenen *Werte* einzelner Stromattribute. Bei einem System, welches den Herzschlag eines Patienten misst, böte es sich demnach an, den unnormalen, also krankhaften Werten eine höhere Bedeutung zuzumessen. Mittels des dritten Graphen lässt sich spezifizieren, mit welcher Dienstgüte man eine gewisse *Verzögerung* bei der Abarbeitung verbindet. Abbildung 4.1 zeigt je ein Beispiel der verschiedenen Graphen. Dabei ist der Schwellwert, an dem der Administrator eine Verzögerung als kritisch betrachtet, mit `q` eingezeichnet und der letzte Graph ist beispielsweise so zu deuten, dass die Zahlen im oberen Drittel des Wertebereichs, weniger wichtig sind und so eher verworfen werden können, hingegen ein Verlust von Stromtupeln mit Extremwerten zu erheblicher Beeinträchtigung der Dienstgüte führen würde.

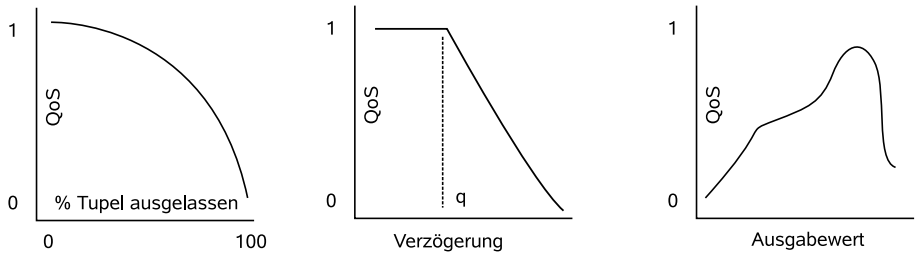

Abbildung 4.1: Die Graphen zur QoS-Spezifikation in Aurora

Selbstverständlich kann man das System nicht zu Höchstleistungen zwingen, indem man hohe (unrealistische) Qualitätsanforderungen mittels der Graphen festlegt, im Gegenteil: Die Optimierungsalgorithmen funktionieren nur dann gut, wenn man dem System genügend Spielraum lässt, es also an verschiedenen Stellen sparen kann, um dafür bei anderen wichtigeren Operationen besser zu sein.

Die Entwickler von Aurora haben einen grafischen Ansatz mitunter wegen der anschaulichen Darstellung der Anfragen und die damit einhergehende Akzeptanz bei Leuten gewählt, die sonst weniger mit Informatik und Datenbanken zu tun haben. Aber nicht nur Einfachheit hat man mit diesem Modell gewonnen, sondern auch eine Beobachtung in der Praxis war Motivationen für diesen Ansatz: Oft hat man nämlich mehrere Eingangsströme aus Sensordaten und viele Anfragen, die aber oft auf der gleichen Basis arbeiten, das heißt sie ziehen ihre Informationen daraus, dass sie die Tupel aus diesen Strömen erst einmal filtern und dann mittels Joins verbinden. Folglich hätten viele Anfragen gemeinsame Teilausdrücke, die ein System wie STREAM selbst eliminieren müsste, wenn man hier beispielsweise CQL einsetzte. In Aurora löst sich das Problem elegant dadurch, dass man zunächst ein Netzwerk aus Operatoren aufbaut, welche die Filterung und Zusammenführung übernehmen und dann über einen Verbindungspunkt die eigentlichen Anfragen anschließt.

4.2 Syntax und Semantik

Genauer betrachtet können an die Ausgabe eines Operators beliebig viele Pfeile angeschlossen werden; Durch sie fließen nachher identische Ströme. Hingegen ist die Anzahl der Pfeile, welche auf den Eingang des Operators zeigen darf, abhängig von dem Operator selbst. [2] nennt sieben elementare Operatoren: Der *Filter*-Operator wird durch ein Prädikat parametrisiert und leitet nur diejenigen Tupel in die Ausgabe, welche es auch erfüllen. *Merge* vereint zwei kompatible Ströme, also solche mit gleichem Schema und gleicher Indexmenge, zu einem einzigen. *Resample* nimmt zwei Ströme S_1 und S_2, eine Fenstergröße w und eine Funktion f. Der Operator legt sich anhand des Stroms S_1 einen Vorrat von w Werten an, die er verwendet, um für die Zeitstempel aus S_2 Werte mit Hilfe von f zu interpolieren. *Drop* ist eine Art Filter, welcher einen Teil der Tupel basierend auf den Zeitstempeln (der Indexmenge) verwirft. Man kann dazu eine Zeitspanne von k Einheiten (z.B. 5 Minuten) angeben und somit erreichen, dass alle Elemente mit dem k-ten Zeitstempel blockiert werden. Im Beipiel mit den 5 Minuten wären das unter Umständen die Zeiten 11.02 Uhr, 11.07 Uhr, 11.12 Uhr, usw.

Die *Join*-Operation wird wie im bekannten Relationenmodell durch ein Verbundsprädikat parametrisiert, anhand dessen es Partner in den Eingabeströmen ermittelt und an die Ausgabe weiterleitet. Zusätzlich jedoch wird hier ein Zeitfenster spezifiziert, in welchem Verbundpartner ermittelt werden können. Ferner ist noch eine Funktion mit anzugeben, welche angibt, wie das Ausgabetupel eines erfolgreichen Verbunds berechnet wird. Weiterhin gibt es zwei *Map*-Operationen. Die normale Variante bildet eingehende Stromtupel nach einer bestimmten Funktion ab, das heißt sie verändert einzelne Stromattribute, fügt neue hinzu oder löscht andere. Die fensterbasierte Variante macht dasselbe, nur kann sie zur Berechnung der Ausgabe alle Tupel in einem gegebenen Zeitfenster heranziehen.

[2] unterscheidet hier sogar noch weiter nach der Art der Fensteroperation: *Slide* verschiebt den aktuellen Ausschnitt jeweils um eine gewisse Anzahl von Elementen, während *Tumble* sicherstellt, dass die aktuellen Fensterinhalte immer komplett erneuert werden, es also nicht passieren kann, dass ein Stromtupel in zwei Fenstern vorkommt. Zusätzlich wird in [3] noch

erläutert, dass Tumble durch eine Aggregatfunktion und eine Menge von Attributen parametrisiert ist, nach denen der zu verarbeitenden Strom gruppiert wird. Dabei wird so lange ein internes Fenster aufgebaut, bis ein ankommendes Stromtupel sich in den Gruppierungsattributen von den Tupeln des bisherigen Fensters unterscheidet. In diesem Fall wird die Aggregatfunktion auf dem bisherigen Fenster aufgerufen und das Ergebnis in den Ausgabestrom geleitet.

Diese Art von Operation macht natürlich nur dann Sinn, wenn im Eingabestrom in der Regel viele Tupel hintereinander genau gleiche Werte in den Gruppierungsattributen haben, sonst fasst man immer lediglich einzelne Tupel zusammen. Um eben das zu verhindern, kann man den Strom vorher entsprechend umordnen: In [3] wird beispielsweise die Operation *WSort* (windowed sort) genannt, welche einen Strom ausschnittsweise basierend auf einer gewissen Fenstergröße sortiert, sie ist also eine Art Slide. Bei einer *Latch*-Operation, die ähnlich wie Tumble funktioniert, dürfen zusätzlich über die Verschiebung hinweg noch Daten, bzw. Zustände intern gespeichert werden, was zum Beispiel nützlich ist, wenn man das Maximum eines Stroms bestimmen will.

4.3 Beispielanfragen

Mit all diesen primitiven Operatoren lassen sich nun komplexere realisieren, wie etwa eine Stromverzweigung, die abhängig von gewissen Eigenschaften der Stromtupel diese in eine bestimmte Richtung weiterleitet. Denkbar wäre im Umfeld eines großen Versandhauses Einkäufe mit großem Wert gesondert zu behandeln und zu analysieren, diese also in einen separaten Strom umzuleiten. Geschehen könnte dies, indem man an die Ausgabe des zu teilenden Stroms zwei Pfeile anschließt, an deren Enden zwei Filter mit zueinander komplementären Prädikaten dafür sorgen, dass Tupel genau dann von ihnen weitergeleitet werden, wenn der andere Filter sie aussortiert.

Die Realisierung unseres Eingangsbeispiels mit dem Telefonstrom in Aurora zeigt Abbildung 4.2. Dabei ist es hier wegen der Fixierung auf das Stromparadigma erheblich schwerer als in den zuvor vorgestellten Sprachen, die Problemstellung mittels einer Anfrage umzusetzen. Denn die Gruppierung nach allen Nummern der Anrufer erfordert prinzipiell ein beliebig großes Fenster, was unmöglich ist, da der darauf arbeitende Operator grundsätzlich beliebig lange blockieren würde. Das Beispiel ist also nur dazu gedacht eine Vorstellung davon zu bekommen, wie man das Problem in Aurora anginge, eine vollständige Lösung ist es nicht.

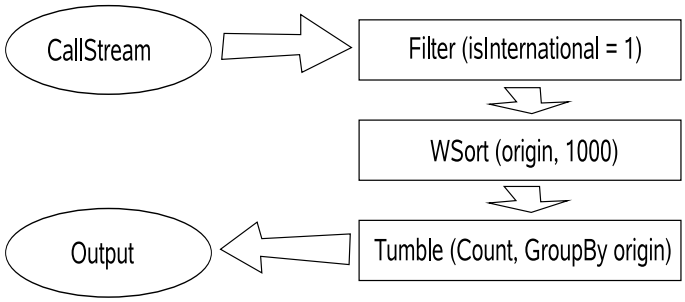

Abbildung 4.2: Das Telefonbeispiel in Aurora

Dabei werden zunächst die nationalen Anrufe mittels eines Filters aus dem `CallStream` entfernt. Anschließend werden die Elemente mittels `WSort` in einem Fenster der Größe `1000`

nach `origin` sortiert, dabei ist klar, dass der Operator erst dann Tupel weiterleitet, wenn er tausend erhalten hat. Jetzt ist der `Tumble`-Operator in der Lage den sortierten Strom zu gruppieren und mittels `Count` den Ergebnisstrom zu produzieren. Dabei ist diese Aggregatfunktion so definiert, dass sie sowohl die Tupel im aktuellen Fenster zählt als auch deren gemeinsame Telefonnummer – also das Gruppierungsattribut – mitausgibt. Diese Lösung ist insofern nicht ganz befriedigend, als dass es passieren kann, dass gleiche Nummern, die nur weit voneinander entfernt im Strom vorkommen, nicht durch den `WSort` nebeneinander einsortiert werden und im Ausgabestrom der Anfrage Tupel mit gleichen Nummern vorkommen, die eigentlich nicht hätten getrennt gezählt werden dürfen. Die Lösung hierfür wäre beispielsweise die Addition in einen gemeinsamen Datenbestand außerhalb von Aurora in der Anwendung zu machen.

4.4 Bewertung

Den Entwicklern von Aurora ist es gelungen eine interessante Alternative zu den meist SQL-verwandten Anfragesprachen im Bereich der DSVS zu schaffen, die sich insbesondere durch die auch für Fachfremde leichte Bedienbarkeit und die Möglichkeit auszeichnet, Dienstgüte für Anfragen zu spezifizieren. Benutzer, welche bereits SQL beherrschen, werden hingegen die grafische Anfragespezifikation gegenüber einer textuellen wahrscheinlich als umständlicher empfinden.

5 Weitere Anfragesprachen

Die bisher vorgestellten Anfragesprachen sind aus vielfältigen Gründen und mit teilweise sehr verschiedenen Zielsetzungen entstanden. Die Möglichkeiten, wo man hier Schwerpunkte setzen kann, sind nahezu unbegrenzt. So gibt es neben den zuvor behandelten Vertretern eine Reihe weiterer Anfragesprachen, die individuelle Ziele verfolgen.

5.1 Expressive Stream Language (ESL)

Wie in [8] beschrieben liegt bei ESL, welches im *Stream-Mill*-System Verwendung findet, beispielsweise das Augenmerk auf Ausdrucksmächtigkeit. Diese Sprache ist *Turing-vollständig* und damit genauso ausdrucksmächtig wie jede höhere Programmiersprache, was insbesondere im Kontext von komplexen Aufgaben wie der Erkennung von bestimmten Mustern in Daten nützlich ist. ESL setzt auf SQL auf und unterstützt sowohl dauerhafte Anfragen auf Datenströmen als auch Ad-Hoc-Anfragen auf Datenbanktabellen und Sichten, die auf Stromdaten definiert sind. Hier hat man also ähnlich wie in CQL die Möglichkeit aktuelle Daten aus Strömen mit herkömmlichen statischen Daten zu verknüpfen.

Erwähnenenswert sind die verschiedenen Zeitstempelvarianten, die in ESL unterstützt werden: Neben *externen* Zeitstempeln, die von der Stromanwendung selbst generiert werden, sowie *internen* Zeitstempeln, welche durch ESL an ankommende Stromtupel vergeben werden, gibt es noch *latente* Zeitstempel. Diese Zeitstempel werden erst bei Bedarf, zum Beispiel bei Fensteroperationen initialisiert, wobei das System hier genau wie bei internen Zeitstempeln diese Aufgabe selbst übernimmt.

5.2 GSQL

GSQL ist neben Hancock eine weitere Entwicklung von AT&T. Diese Sprache findet in *Gigascope* Anwendung, einem System zur Analyse von Netzwerkströmen. Wie in [10] nachzulesen ist, spielt dabei die Verarbeitungsgeschwindigkeit eine wesentliche Rolle. Um diese zu erreichen, erlaubt man nur reine Ströme, also keine Relationen wie in CQL und definiert auch die Operationen immer mit dem Ziel, diese möglichst effizient implementieren zu können.

So fordert man zum Beispiel beim Join, dass das Verbundprädikat mindestens ein geordnetes Attribut der zwei beteiligten Eingabeströme beinhalten muss. Wenn bespielsweise `B` und `C` Ströme mit dem jeweils geordneten Attribut `ts` (timestamp) wären, so könnte die Bedingung `B.ts >= C.ts-1 AND B.ts <= C.ts+1` lauten. Da die jeweiligen Attribute geordnet sind und damit im Stromverlauf monoton steigen oder fallen, ist es GSQL möglich ein dem aktuellen Bedarf ausreichend großes Fenster aufzubauen ähnlich der Join-Operation in Aurora, bei der das Fenster durch einen Zeitrahmen spezifiziert wird.

5.3 AQuery

Die Zusammenfassung von [9] macht deutlich, dass die Sprache AQuery mit dem Ziel entwickelt worden ist, den Entwurf *reihenfolgeabhängiger* Anfragen möglichst einfach zu gestalten. Darunter fällt beispielsweise die Aufgabe für eine bestimmte Aktie an einem gegebenen Datum den günstigsten Zeitpunkt für Kauf und Wiederverkauf zu ermitteln. Dabei legen wir eine Tabelle `Trades(ID, tradeDate, price, ts)` zu Grunde, in welcher der Wert einer Aktie mit der Identifikationsnummer für einen bestimmten Tag zu bestimmten Zeitpunkten `ts` gespeichert ist. Um die Aufgabestellung zu lösen, müsste man die Einträge einer bestimmten Aktie eines festen Datums in der *Reihenfolge* ihrer Zeitstempel durchlaufen und das bis dahin erfasste preisliche Minimum von dem aktuellen Wert subtrahieren. Das Ergebnis wäre das Maximum dieser Differenz, also der größtmögliche Gewinn an diesem Tag.

In SQL:1999 lässt sich diese Aufgabenstellung nur mittels komplizierten Schachtelung von Unteranfragen lösen, die zudem noch schwer zu optimieren sind. Mit AQuery lässt sich das für den Benutzer wesentlich übersichtlicher lösen. Diese Sprache setzt nämlich auf SQL-92 auf und die Erweiterung besteht hauptsächlich darin, dass man sogenannte *arrables* (array-tables) anstelle von herkömmlichen Tabellen einführt. Diese sind vereinfacht dargestellt identisch, bis auf die Tatsache, dass man auf arrables eine Ordnung definieren kann, was jedoch die Formulierung reihenfolgeabhängiger Anfragen erheblich vereinfacht.

6 Fazit

Die starken Unterschiede, welche die hier vorgestellten Anfragesprachen aufweisen, deuten darauf hin, dass das Stromkonzept doch einige Schwierigkeiten mit sich bringt, für die es keine Universallösung gibt. Der imperative Ansatz ist vergleichsweise kompliziert und eignet sich daher wenig für interdisziplinäre Projekte, an denen auch Leute mitarbeiten, welche mit Programmiersprachen nur in geringem Maße vertraut sind. Jedoch können auch bei einer solchen Anfragesprache gewisse Eigenschaften wie im Beispiel von Hancock die Kompatibilität zur C den Ausschlag dafür geben, sie doch in bestimmten Fällen einzusetzen.

Anfragesprachen nach dem deklarativen Paradigma bieten in der Regel eine Syntax, die sich stark an SQL orientiert und somit für Kenner des Relationenmodells sehr einfach zu erlernen ist. Aber gerade bei diesen Leuten besteht die Gefahr, dass sie Anfragen spezifizieren ohne sich darüber Gedanken zu machen, wie das System diese umsetzt, denn obwohl die Syntax SQL ähnelt, unterscheiden sich die zu Grunde liegenden Ströme erheblich von den bekannten Relationen.

Im Aurorasystem hat man mit den Graphen dagegen eine völlig neue Art der Anfragesstellung eingeführt, die sich ganz am Stromkonzept orientiert. Wegen der Einfachheit, der Unabhängigkeit von der Anwendung und der Möglichkeit, hier auch Dienstgüte zu spezifizieren, erfüllt Aurora in hohem Maße die in der Einleitung formulierten Anforderungen. Nachteilig ist aber die für viele Leute ungewohnte Schnittstelle und ferner ist nicht klar, wie man hier Anfragen ohne die grafische Oberfläche mit Programmen generieren und ausführen kann. Auch die fehlende Unterstützung von statischen Relationen, wie man sie beispielsweise in CQL findet, machen Aurora für manche Zwecke unbrauchbar.

So haben alle hier behandelten Vertreter ihre individuellen Vor- und Nachteile, letztendlich bestimmt der konkrete Verwendungszweck, welche Sprache sich am ehesten eignet. Bei all der Vielfalt lassen sich aber auch einige Gemeinsamkeiten in den Sprachen feststellen: So finden sich überall Konzepte zur Behandlung des zeitlichen Ablaufs, in welchem Stromtupel im System antreffen, und auch für die Problematik der potenziellen Unendlichkeit von Strömen finden sich in jeder Sprache Konstrukte zur Spezifikation von Fenstern, woraus sich schließen lässt, dass diese Phänomene bei Strömen eine zentrale Rolle spielen.

Mit zunehmender wirtschaftlicher Bedeutung von DSVS wird wie bei DBVS eine Konsolidierung einsetzen und damit die Zahl der für die Praxis relevanten Sprachen erheblich verringern. Welches Paradigma sich dabei durchsetzt, ist im Moment nicht absehbar.

Literaturverzeichnis

[1] A. Arasu, S. Babu, J. Widom: *The CQL Continuous Query Language: Semantic Foundations and Query Execution*, 2003. (Elektronisch verfügbar unter: `http://dbpubs.stanford.edu/pub/2003-67`)

[2] D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, S. Zdonik: *Monitoring Streams – A New Class of Data Management Applications.* In: Pproceedings of the 28th International Conference on Very Large Data Bases (VLDB'02), Hong Kong, China, 2002. (Elektronisch verfügbar unter: `http://dbpubs.stanford.edu/pub/2003-67`)

[3] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Çetintemel, Y. Xing, S. Zdonik: *Scalable Distributed Stream Processing.* In: Proceedings of the Conference for Innovative Database Research (CIDR), Januar 2003. (Elektronisch verfügbar unter `http://www.cs.brandeis.edu/~mfc/papers/cidr03.pdf`)

[4] D. Abadi, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, C. Erwin, E. Galvez, M. Hatoun, J. Hwang, A. Maskey, A. Rasin, A. Singer, M. Stonebraker, N. Tatbul, Y. Xing, R. Yan, S. Zdonik: *Aurora: A Data Stream Management System.* (Elektronisch verfügbar unter `http://www.mit.edu/~dna/AuroraDemo.pdf`)

[5] D. Abbadi, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. *Aurora: A New Model and Architecture for Data Stream Management.* In: Brown Computer Science CS-02-10, August 2002.

[6] C. Cortes, K. Fisher, D. Pregibon, A. Rogers, F. Smith: *Hancock: A Language for Extracting Signatures from Data Streams*, 2000. (Elektronisch verfügbar unter `http://www.research.att.com/projects/hancock/kdd.pdf`)

[7] K. Fisher, K. Högstedt, A. Rogers, F. Smith: *Hancock 2.0.1 Manual*, 2002. (Elektronisch verfügbar unter `http://www.research.att.com/projects/hancock/manual.pdf`)

[8] Y. Bai, R. C. Luo, H. Thakkar, H. Wang, C. Zaniolo: *An Introduction to the Expressive Stream Language (ESL).* (Elektronisch verfügbar unter `http://magna.cs.ucla.edu/stream-mill/doc/esl-manual.pdf`)

[9] A. Lerner, D. Shasha: *AQuery: Query Language for Ordered Data, Optimization Techniques, and Experiments.* In: Proceedings of the 29th VLDB Conference, Berlin, Germany, 2003.

[10] C. Cranor, T. Johnson, O. Spataschek, V. Shkapenyuk: *Gigascope: A stream database for network applications.* In: ACM SIGMOD Conference, 2003. (Elekronisch verfügbar unter `http://www.db.cs.cmu.edu/Pubs/Lib/sigmod03gigascope/sigmod03_gigascope.pdf`)