Data Streams:
Gegenüberstellung existierender Systeme und Architekturen
Betreuer: Jürgen Göres

Marius Renn

17. Juni 2005

Inhaltsverzeichnis

Kapitel 1

Einführung

Ein *Datenbankverwaltungssystem* (*DBVS*) ist ein Softwaresystem zur Verwaltung großer Mengen strukturierter Daten (*Datenbank*). Ein DBVS kann Anfragen mehrerer Benutzer verarbeiten, und in Operationen auf der Datenbank umsetzen. Während früher nur große Firmen genügend Kapazitäten und Leistungen besaßen, um ein DBVS zu verwenden, bilden sie heute einen zentralen Bestandteil der Infrastruktur jeder Organisation [WikiDB].
Ein DBVS wird oft als *HADP*-System (*Human-Active, DBMS-Passive*) bezeichnet [CCCC02], da es nur auf Anfragen des Benutzers reagieren kann, und nicht selbständig arbeitet. Es gibt jedoch eine wachsende Zahl an Anwendungen, für die ein solcher Ansatz ungeeignet ist. Beispielsweise dienen *Monitoring*-Applikationen der aktiven Überwachung von kontinuierlichen Datenströmen. Solche Applikationen finden ihre Anwendung vor allem in der Medizin (zur Überwachung wichtiger Körperfunktionen), beim Militär (zur Überwachung von Soldaten und Feindbewegungen), in der Finanzanalyse (zur Beobachtung von Aktienkurse), und in der geographischen Überwachung von Gegenständen [CCCC02]. Letzteres Szenario soll im folgenden Beispiel veranschaulicht werden:
Bei einem Verleihservice sollen alle wichtigen Geräte mit GPS-Sensoren ausgestattet werden. So kann der aktuelle Aufenthaltsort eines beliebigen Gerätes jederzeit bestimmt werden. Weiterhin soll ein aktives Monitoring-System eine Meldung ausgeben, sobald ein Gerät einen vorgesehenen Bereich verlässt. Das System muss also in der Lage sein, einen Datenstrom mit einer Vielzahl von erfassten Positionsdaten in Echtzeit zu verarbeiten. Anfragen zu Daten aus der Vergangenheit sollen ebenfalls möglich sein, sodass beispielsweise die Aufenthaltsorte eines Gerätes während der vergangenen Woche bestimmt werden können. Sobald ein Gerät die Datenübertragung unterbricht (beispielsweise durch einen technischen Fehler), darf das Monitoring-System nicht durch das Warten auf Daten blockieren. Stattdessen sollte lediglich die letzte Position des Gerätes vermerkt werden, und eine Meldung über das Ausbleiben der Daten ausgegeben werden.
Sicherlich wäre es nicht praktikabel, solche Monitoring-Systeme in einem herkömmlichen DBVS zu implementieren. Die passive Funktionsweise eines solchen Systems würde nicht ausreichen das kontinuierliche Beobachten, und das Melden von Sonderwerten effizient zu implementieren. Zwar wäre es möglich alle Tupel eines Stromes in einer Datenbank abzulegen, jedoch sind diese Systeme nicht auf eine solch hohe Einfügelast ausgelegt. Ohnehin ist es oft gar nicht erwünscht alle Daten aus einem Strom aufzuheben, da nur gewisse Werte auf Dauer von Interesse sind. So wäre es im obigen Szenario sicherlich sinnvoller lediglich die Positionsänderungen der Geräte abzuspeichern, als die gesamten Positionsdaten zu jedem Zeitpunkt. Eine solche Beschränkung auf die relevanten Daten macht diese nicht nur übersichtlicher, sondern spart zudem an Speicherplatz.
Trotz der Charakterisierung als passive Systeme, besitzen zahlreiche moderne DBVS aktive Mechanismen. So führte der SQL99-Standard das Konzept der *Trigger* ein, welche dem DBMS erlauben selbständig auf bestimmte Ereignisse zu reagieren. Prinzipiell wäre damit ein aktives Reagieren auf Datenstromwerte möglich. Die geringe Skalierbarkeit der gängigen Trigger-Implementierungen machen jedoch ihren Einsatz in größeren Mengen, wie es für die Stromverarbeitung nötig ist, nicht praktikabel [WiCC00].
Seit einiger Zeit beschäftigt sich eine neue Klasse von Systemen mit der Verarbeitung kontinuierli-

cher Daten. Diese *Datenstromverwaltungssysteme* (*DSVS*) zielen darauf ab, Ströme mit hohen und fluktuierenden Datenraten in angemessener Zeit zu verarbeiten. Man bezeichnet sie auch als *HP-DA*-Systeme (*Human-Passive, DBMS-Active*), da sie Datenströme kontinuierlich verarbeiten und automatisch Anfrageergebnisse an die Anwendung weiterleiten, ohne dass der Benutzer die Anfrage erneut stellen muss. In relativ kurzer Zeit sind auf diesem Gebiet eine Vielzahl von Systemen entstanden, die auf unterschiedlicher Weise versuchen die Anforderungen von Datenstromverarbeitung zu realisieren.
Im Folgenden sollen einige dieser Systeme untersucht und miteinander verglichen, sowie ihre jeweiligen Besonderheiten hervorgehoben und erläutert werden.

1.1 Vergleichskriterien

Um Anfragen auf Datenströmen über die Zeit hinweg zu stellen, werden in DSVS *kontinuierliche Anfragen* (*continuous queries*) verwendet. Diese analysieren fortlaufend den Eingabestrom und geben die Daten aus, welche den Kriterien der Anfrage entsprechen. Im Gegensatz zu ihnen stehen die *einmaligen Anfragen* (*one-time queries*), welche genau zu einem Zeitpunkt ausgeführt werden, und dem Benutzer ein Ergebnis präsentieren. Diese aus DBVS bekannten Anfragen werden in gängigen DSVS ebenfalls unterstützt.
Eine weitere Unterscheidung ist die zwischen *vordefinierten* (*predefined*) und *Ad-hoc-Anfragen* (*ad hoc-queries*). Im Gegensatz zu vordefinierten Anfragen, welche dem DSVS vor der Datenankunft bereits bekannt sind, werden Ad-hoc-Anfragen erst zur Laufzeit gestellt. Diese können einmalige oder fortlaufende Anfragen sein. Optimierungsmaßnahmen, die nun zur Laufzeit durchgeführt werden müssen, werden folglich schwieriger zu implementieren.
Noch problematischer wird es, wenn die Anfrage Datensätze benötigt, welche das System schon verlassen haben. Dies könnte beispielsweise bei einer statistischen Anfrage über alle vergangene Daten der Fall sein. Einige Implementierungen umgehen dieses Problem, indem sie solche Anfragen gar nicht erst zulassen. Diese Einschränkung mag schwerwiegend erscheinen, ist jedoch in der Praxis für viele Fälle ausreichend [BBDM02].
Die beiden vorgestellten Anfragekriterien sind zueinander orthogonal, können also miteinander kombiniert werden. Abbildung 1.1 zeigt die möglichen Kombinationen.

	einmalig	kontinuierlich
vordefiniert	-	✓
Ad-hoc	✓	✓

Abbildung 1.1: Kombination der Anfragekriterien

Ein wichtiges Kriterium der Datenstromverarbeitung ist der Umgang mit *blockierenden Operatoren* (*blocking operatosr*). Diese sind Anfrageoperationen, welche zum Erzeugen eines Ergebnisses *alle* Datensätze der Eingabe verarbeiten müssen. Zu ihnen gehören die Aggregationsoperatoren, wie `SUM`, `COUNT` und `AVG`, jedoch auch der Verbund (*Join*). Da Datenströme potentiell unendlich gross sind, würden blockierende Operatoren nie ein Ergebnis präsentieren. Um dennoch solche Operatoren in DSVS einsetzen zu können, verwenden viele Systeme eine *Fenstersemantik* (*window semantics*). In dieser werten blockierende Operatoren nicht den gesamten Strom, sondern lediglich einen Ausschnitt (ein *Fenster*) des Stroms, aus. Welche Tupel in einem Fenster enthalten sind, definiert der jeweilge *Fensteroperator*. Häufig wird ein *Sliding Window* eingesetzt, welches neue Tupel aus einem Strom in das Fenster hinzufügt, und alte Tupel aus dem Fenster entfernt. Man unterscheidet zwischen *mengenbasierten* (*tuple-based*), und *zeitbasierten* (*time-based*) Sliding Windows. Letztere enthalten all diejenigen Tupel, die in einen vorgegebenen Zeitabschnitt fallen. So könnte beispielsweise ein solches Fenster genau die Tupel der letzten drei Tage enthalten. Ein mengenbasiertes Fenster enthält hingegen eine vorgegebene Anzahl von Tupel (z.B. die letzten 1000 Tupel). Über den dargestellten Zeitausschnitt eines mengenbasiertem Fenster lässt sich

generell keine Aussage machen. Insbesondere deckt ein solches Fenster bei hoher Datenlast eine kürzere Zeitspanne ab, als bei niedriger Last. Daher ist es eher unüblich, mengenbasierte Fenster auf Strömen mit stark schwankender Datenlast zu verwenden.
Offensichtlich benötigen Fensteroperationen einen gewissen Vorrat an Speicher, in denen sie den Stromausschnitt ablegen können. Man nennt solche Operatoren *zustandsbehaftete* Operatoren (*stateful operators*), da ihre Funktionsweise von dem Inhalt der zwischengespeicherten Daten abhängt. So braucht der Verbund (*join*) für jeden Eingabestrom eine Zwischenspeichereinheit, welche ankommende Tupel im Speicher ablegt und mit den anderen Einheiten auf die Verbundbedingung prüft. Im Gegensatz zu solchen Operatoren benötigen *zustandslose Operatoren* (*stateless operators*) keinen Zwischenspeicher. Zu ihnen gehören besipielsweise die Selektion und Projektion. Die hier beschriebenen Systeme behandeln das Einrichten solcher Speichereinheiten auf sehr unterschiedliche Weise, und bieten verschiedene Optimierungsmöglichkeiten an.
Schließlich bleibt noch die viel verwendete Datenstruktur der *Warteschlangen* (*Queues*) zu erwähnen. Um Operatoren voneinander zu entkoppeln, werden zwischen ihnen Warteschlangen eingesetzt. So kann ein vorhergehender Operator ein Ergebnis ausgeben, selbst wenn der nachfolgende noch keine neuen Eingaben annehmen kann.

1.2 Optimierungs- und Adaptierungsmaßnahmen

Damit die Verarbeitung der Datenströme möglichst effizient abläuft, sind Strategien zur Optimierung von Anfragen in der Datenstromverwaltung unumgänglich. Um einen optimalen Anfrageplan zu garantieren, ist es erforderlich diesen zu optimieren. Wie in relationalen Datenbanksystemen üblich kann eine Anfrage oft noch bevor sie ausgeführt wird *algebraische optimiert* werden. Algebraische Optimierungen suchen zu einer gegebenen Anfrage eine möglichst effiziente algebraisch äquivalente Anfrage. So ist es in der Regel effizienter, Selektionen möglichst früh in der Anfrage zu bearbeiten, da nachfolgende Operationen auf weniger Tupeln arbeiten müssen [Haer04].
Anpassung und Neuoptimierung von Anfrageplänen zur Laufzeit sind in der Datenstromverarbeitung besonders wichtig, da Datenströme über die Zeit hinweg in der Last stark schwanken können. Es ist daher erforderlich, dass sich das System den aktuellen Verhältnissen anpasst. Die Möglichkeiten auf einem laufenden System Veränderungen durchzuführen sind begrenzt, da der Datenfluss nicht unterbrochen werden kann. Im Gegensatz zu Optimierungen vor der Laufzeit liegen jedoch bereits statistische Daten über die Auslastung der Operatoren vor, die zur Optimierung genutzt werden können. Man spricht daher bei Systemen, die Optimierungen zur Laufzeit durchführen, von *adaptiven Systemen* (*adaptive systems*).
Adaptierung ist aus einem weiteren Grund für die Stromverarbeitung wichtig: Da Daten aus einem Strom schneller ankommen könnten, als das sie verarbeitet werden könnten, sind Maßnahmen zum Vermeiden eines Überlaufs erforderlich. In solch einem Fall wird ein *Load Shedder* eingesetzt, welcher zur Laufzeit eine gewisse Anzahl von Tupel aus der Verarbeitung auslässt. Der Load Shedder versucht dabei so wenig Tupel wie möglich auszulassen, da die fehlenden Werte das Ergebnis verfälschen oder zumindest ungenauer machen.
Gerade bei Echtzeitsystemen ist es wichtig, dass Ergebnisse aus der Datenstromverarbeitung mit möglichst wenig Verzögerung ausgegeben werden. Eine absolute Genauigkeit der Daten wird hier nicht gefordert. Andererseits gibt es Anwendungen, bei denen eine Verzögerung der Daten vernachlässigbar ist, jedoch eine hohe Datenpräzision gefordert wird. Diese zwei Kriterien, Verzögerung (*latency*) und Vollständigkeit (*drop*), werden oft als *Dienstgütekriterien* (*Quality of Service, QoS*) zusammengefasst. Es gibt bisher nur wenige DSVS, die eine Angabe von Dienstgütekriterien erlauben.

Kapitel 2

STREAM

Ohne Zweifel hat sich die Sprache SQL als Standard im Bereich der relationalen Datenbanken durchgesetzt, und wurde von zahlreichen namhaften Unternehmen wie IBM, Oracle und Microsoft als Anfragesprache implementiert und eingesetzt [WikSQL]. Prinzipiell wäre es möglich, SQL auch in der Datenstromverarbeitung einzusetzen: Referenzen auf Relationen könnten durch Referenzen auf Ströme ersetzt und die aus den Anfragen ermittelten Ergebnisse dann wieder in einen Ausgabestrom geleitet werden. Mit der Einführung einer Fenstersemantik (*windowed table functions*) in SQL-2003 [Kulk03], ist auch diese Möglichkeit im Standard verankert.
Mit der Hinzunahme von weiteren Konzepten, wie die Vermischung von Relationen und Strömen, kommt es bei der relationalen Anfragesprache jedoch schnell zu Unklarheiten. Ein einfaches Beispiel zeigt die auftretenden Probleme:

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

`Items` sei hier ein unendlicher Datenstrom, und `PriceTable` eine Relation (Tabelle). `[Rows 5]` spezifiziert ein Sliding Window mit fünf Elementen. Unklar bleibt, ob die Ausgabe ein Datenstrom oder eine Relation ist. Weiterhin ist unklar was passiert, wenn ein Preis im Fenster sich nachträglich ändert [ArBW03].
Das STREAM-Projekt der University of Stanford ist ein DSVS welches sich darauf konzentriert, mehrfache, hochfrequentierte, fortlaufende Anfragen in einer SQL-ähnlichen Sprache namens *CQL* (*Continuous Query Language*) zu verarbeiten. *CQL* ist eine relativ kleine Erweiterung zu SQL, und nutzt für alle relationalen Operationen die aus SQL bekannten Konstrukte, erweitert diese jedoch um Operationen, die Relationen in Ströme und umgekehrt transformieren können. Das STREAM-System wurde dazu entwickelt, schnelle und schwankende Datenströme auf Plattformen mit wenig Systemressourcen bearbeiten zu können.

2.1 Anfrageerstellung

Im STREAM-System wird zwischen *Relationen* (*relations*) und *Strömen* (*streams*) unterschieden. Ströme sind zeitliche Abfolgen von Tupel, an denen zu jedem Zeitpunkt weitere Tupel angehängt werden können. Relationen hingegen enthalten zu jedem Zeitpunkt eine potentiell von anderen Zeitpunkten verschiedene Menge von Tupeln. Folglich enthält eine Relation im Gegensatz zu einem Strom zu einem späteren Zeitpunkt womöglich nicht mehr Tupel eines früheren Zeitpunkts. Daher bezeichnet STREAM Relationen auch als *zeitvariierende Multimenge* (*time-varying bag*), und die Multimenge einer Relation zu einem Zeitpunkt t als die *momentane Relation* (*instantaneous relation*) zum Zeitpunkt t. Alle SQL-typischen Operatoren, wie die Selektion, Projektion und Verbunde (*Joins*) sind *Relation-zu-Relation-Operatoren* (*relation-to-relation operators*), also solche, die Relationen als Eingabe nehmen und wieder eine Relation ausgeben. Um Datenströme

in Relationen umzuwandeln werden *Strom-zu-Relation-Operatoren* (*stream-to-relation operators*) verwendet. Insgesamt gibt es drei solche Fensteroperatoren in STREAM. Der *mengenbasierte* Fensteroperator erzeugt Relationen, welche die letzten n Tupel des Eingabestroms enthalten. Er wird in CQL mit der Angabe von `[Rows n]` aufgerufen. Der *zeitbasierte* Fensteroperator erzeugt Relationen, die nur Tupel einer angegebenen Zeitspanne enthalten. Dieser wird mit dem Ausdruck `[Range t]` aufgerufen. Der *partitionierende* Fensteroperator nimmt als Eingabe einen Strom s, eine Anzahl von Tupel n, und eine Menge von Attributen $A_1...A_k$. Die Tupel aus s werden basierend auf der Gleichheit ihrer Attributwerte $A_1...A_k$ gruppiert, wobei jede Gruppe maximal n Einträge hat. Die Ausgaberelation des Operators enthält die Vereinigung dieser Gruppen. Der partitionierende Fensteroperator wird in CQL durch den Ausdruck `[Partition by` $A_1...A_k$ `Rows` n`]` aufgerufen.
Um Relationen wieder in Ströme zu verwandeln, gibt es drei *Relation-zu-Strom-Operatoren* (*relation-to-stream operators*): *Istream* (*insert stream*) fügt ein Tupel T in einen Strom immer dann ein, wenn T auch der Eingaberelation hinzugefügt wurde. *Dstream* (*delete stream*) hingegen fügt ein Tupel T ein, wenn T aus der Eingaberelation entfernt wurde. *Rstream* (*relation stream*) fügt zu jedem Zeitpunkt alle Tupel der Eingaberelation in den Strom ein.
Folgende Beispielanfrage soll die Operatoren von CQL veranschaulichen:

```
Select Istream(Oven.temperature, Meal.name)
From Oven [Range 3 Minutes], Meal
Where Meal.inOvenId = Oven.id
```

Diese Anfrage ermittelt die Temperatur und das enthaltene Gericht derjenigen Datensätze, die in den letzten drei Minuten aus dem Datenstrom `Oven` gelesen wurden. Das Beispiel zeigt die unterschiedliche Behandlung von Relationen und Strömen. Hier muss der Datenstrom `Oven` erst in eine Relation umgewandelt werden, um im Verbund verwendet werden zu können. Die Ausgabe wird wieder durch den `Istream`-Operator in einen Datenstrom geleitet.
Bei der Entwicklung des STREAM Systems wurde Wert darauf gelegt, dass Administratoren das System zur Laufzeit inspizieren und einstellen können. Daher wurde eine grafische Benutzeroberfläche entwickelt, welche das System, und alle zum aktuellen Zeitpunkt aktiven Anfragen grafisch darstellen kann. Abbildung 2.1 zeigt einen Screenshot des Systems zur Laufzeit. Es können so auch Anfragen zur Laufzeit gestellt werden, jedoch ermöglicht STREAM keine Anfragen zu vergangenen Daten, die das System schon verlassen haben.

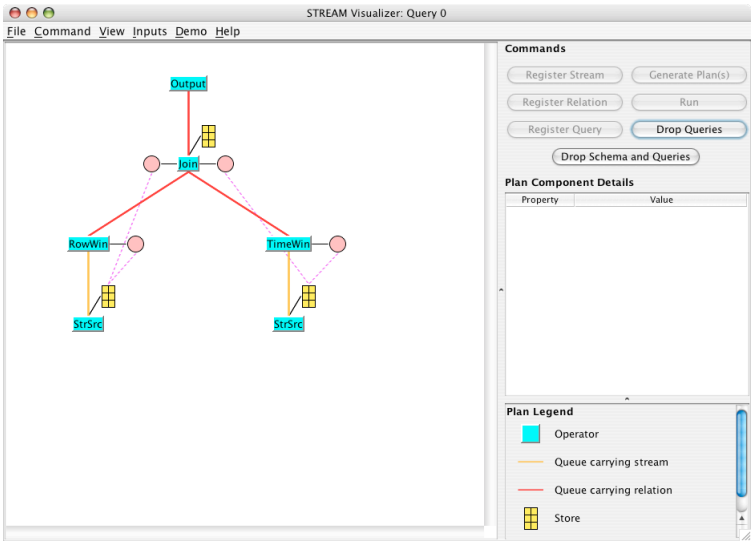

Abbildung 2.1: Visualisierung des STREAM-Systems zur Laufzeit

2.2 Implementierung und Optimierung

STREAM trifft in der Implementierung keine Unterscheidung zwischen Strömen und Relationen. Beide werden zu einer Sequenz von Tripeln zusammengefasst, wobei jedes Tripel aus einem Tupel, einem Zeitstempel und einem Binärwert besteht. Dieser hinzugekommene Wert gibt an, ob ein Tupel ein *Einfügen* oder ein *Entfernen* darstellt. Im Folgenden bezeichnen wir die beiden Arten von Elementen als *positive* und *negative* Elemente. Während Elemente aus Relationen entweder positiv oder negativ sein können, besitzen Ströme nur positive Elemente.
Um Operatoren voneinander zu entkoppeln, werden zwischen ihnen Warteschlangen eingesetzt. Operatoren lesen Operanden von ihren Eingabeschlangen und geben ihre Ergebnisse in die Ausgabeschlange weiter. Damit sehen Implementierungen wie die der *Relation-zu-Strom*-Operatoren wie folgt aus: Der Istream Operator muss alle eingetroffenen positiven Elemente weiterleiten, während der Dstream Operator alle negativen Elemente als Positive ausgeben muss. Schwieriger wird es bei zustandsbehafteten Operatoren wie den Fensteroperatoren. Da diese Ströme als Eingabe nehmen, erhalten sie nur positive Elemente. Diese können dann zwar weitergeleitet werden, müssen jedoch nach einer gewissen Zeit (bzw. nach einer bestimmten Anzahl von erhaltenen Tupeln) wieder entfernt werden, sprich, als negative Elemente ausgegeben werden. Um diese Elemente vermerken zu können benötigen solche Operatoren Zwischenspeichereinheiten, welche in STREAM *Synopsen* genannt werden. So benötigt beispielsweise der Verbund für jeden Eingabestrom eine Synopse. Beim Eintreffen eines Elementes wird dieser in die zugehörige Synopse eingetragen, und mit den Elementen der anderen Synopsen auf die Verbundbedingung geprüft. Abbildung 2.2 zeigt eine Anfrage, ähnlich wie sie auch in der STREAM-Oberfläche samt Synopsen dargestellt wird.

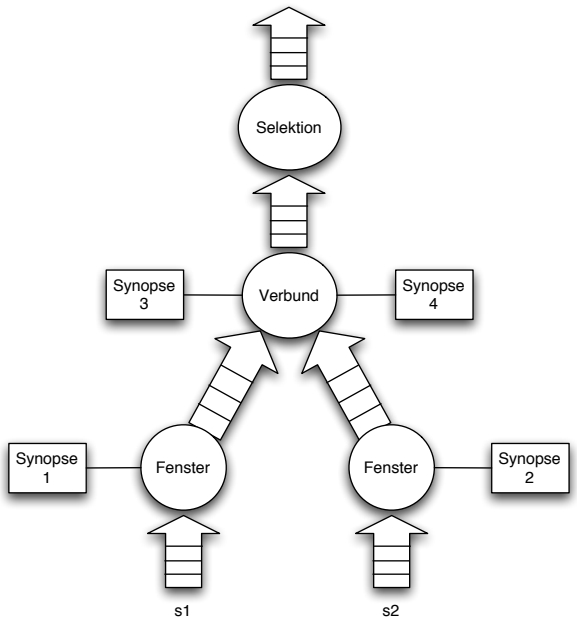

Abbildung 2.2: Eine einfache Anfrage mit Synopsen im STREAM-System

Die Abbildung zeigt ausserdem ein weiteres auftretendes Phänomen in den Synopsen. So besitzen hier die Synopsen 1 und 3 (respektive 2 und 4) den nahezu identischen Inhalt. Um solch eine hohe Datenredundanz zu vermeiden, wurde das Synopsenkonzept um *Synopsen-Sharing* erweitert. Hierbei werden zwei ähnliche Synopsen ersetzt durch zwei *Stubs*, und einem *Store*, welcher die eigentlichen Daten enthält. Über die Stubs kann nun auf den Store-Speicher zugegriffen werden, und da die Stubs die gleiche Schnittstelle implementieren, wie die Synopsen, sind diese beliebig miteinander austauschbar.
Zur Systemlaufzeit kann es vorkommen, dass die eintreffenden Daten in der Last stark schwanken.

STREAM implementiert einen *adaptiven* Ansatz, der eine dynamische Anpassung des Systems realisiert. Dazu gibt es eine Monitoring-Einheit, *StreaMon*, welche in der Lage ist, bestehende Anfragen zu reorganisieren und zu optimieren.
StreaMon besteht aus drei Komponenten: Dem *Executor*, welcher Anfragepläne ausführt, dem *Profiler*, der statistische Daten über Performanz zur Laufzeit sammelt, und dem *Reoptimierer*, welcher sicher stellt, dass die aktuellen Anfragepläne auch die effizientesten darstellen. StreaMon kann beispielsweise erkennen, wann gewisse Synopsen überflüssig sind. Als Veranschaulichung diene ein Verbund über zwei Attribute A und B bei dem sichergestellt ist, dass kein Wert in B vor demselben aus A auftritt. Somit könnte die Synopse für den Strom der B-Attribute vollständig weggelassen werden, da ein späterer Verbund eines B-Elements mit einem aus A ohnehin nie stattfinden würde. Um einen solchen *Constraint* festzustellen, beobachtet der Profiler ständig die einfliessenden Datenströme und informiert bei Bedarf den Reoptimierer über mögliche Constraints. Ebenso meldet der Profiler sobald ein Constraint nicht mehr gültig ist, also dann, wenn ein Element die Constraint-Eigenschaft verletzt.
Weiterhin verfolgt StreaMon einen *Greedy-Ansatz* zur Reorganisation von aufeinanderfolgenden Verbunden. Hierbei werden die Operatoren so angeordnet, dass Verbunde mit einer geringen Anzahl von Ausgabetupeln vor Verbunden mit hoher Ausgabelast ausgeführt werden.
Um Überläufe bei hohen Datenraten zu vermeiden, wird in STREAM *Load Shedding* eingesetzt. Dazu gibt es eine Vielzahl von *Sampling*-Operatoren, welche nach verschiedenen Verteilungsfunktionen Elemente aus dem Datenstrom auslassen. So können auch Plattformen mit begrenzter Rechenleistung große Datenmengen in Echtzeit bewältigen. Zur Vermeidung von Speicherüberläufen wurden weitere Mechanismen in das System eingebaut. So werden Synopsen in speicherkritischen Situation auf Kosten der Genauigkeit verkleinert. Ist Synopsen-Sharing aktiv, so kann eine solche Verkleinerung natürlich mehrere Operatoren negativ beeinflussen [ABBC04].

Kapitel 3

Aurora

Datenstromverarbeitung findet häufig seine Anwendung in der Signalverarbeitung. Dazu müssen Daten von Sensoren in Echtzeit ausgewertet werden. Eine Vielzahl von Sensoren (insbesondere mobile Sensoren) dürfen nur wenig Strom verbrauchen, und erzeugen daher oft stark verrauschte Daten. Eine Bereinigung bzw. Aufarbeitung der Daten ist also von Nöten. Wie soll man jedoch solche Aufarbeitungsmodule sinnvoll in eine Anfrage kapseln? Sicherlich ist es umständlich in jeder Anfrage an die Sensordaten eine vorhergehende Bereinigung mit anzugeben. Eine andere Möglichkeit wäre die Aufbereitung in ein unabhängiges System vor dem Eintreffen der Daten in das DSVS zu implementieren. Allerdings ist auch eine solche Trennung der Verarbeitungsschritte nur suboptimal [ZCSB03].
Das Aurora-System, welches gemeinsam von den U.S. Universitäten Brown, Brandeis und dem MIT konzipiert und als Prototyp implementiert wurde, geht daher von der typischen Anfragestruktur weg und stützt sich stattdessen auf ein flexibles *Datenflusssystem*. Datenströme, die durch ein solches System fließen, können schrittweise in Modulen aufbereitet und verarbeitet werden. Diese Verarbeitungssysteme konstruiert der Benutzer als ein Netzwerk von Kästchen und Pfeilen (*Boxes and Arrows*). Eine weitere Besonderheit von Aurora ist die Möglichkeit, die Einhaltung von Dienstgüte-Eigenschaften zu gewähren. So ist das System in der Lage, Daten je nach Wichtigkeit unterschiedlich zu gewichten. Aurora beinhaltet zusätzlich zu seinem Laufzeitsystem und der grafischen Benutzeroberfläche ein umfangreiches *Monitoring*-Programm [ACCC02].

3.1 Anfrageerstellung

Um Anfragen in Aurora zu modellieren, kann der Benutzer eine Vielzahl von Operatoren zu einer komplexen Anfrage grafisch zusammenbauen. Operatoren heißen in Aurora *Boxes*, und werden in der grafischen Oberfläche als Kästchen dargestellt. Um die Ausgaben der Operatoren an andere Operatoren weiterzuleiten, werden diese in der grafischen Oberfläche mit Pfeilen (*Arrows*) verbunden, wobei eine zyklische Weiterleitung nicht erlaubt ist [CCCC02]. So entsteht für eine Anfrage ein azyklischer gerichteter Graph, dessen Knoten die einzelnen Operatoren darstellen, und die Kanten die Weiterleitung der Ergebnisströme. Ein solcher Anfragegraph darf dabei beliebig viele Eingaben, und beliebig viele Ausgaben enthalten. Eingabeknoten stellen die Schnittstelle zu den unbearbeiteten Datenströmen dar, die in das Aurora-System gelangen. Die Ausgabeknoten sind die Schnittstellen zu den Programmen, die auf die erhaltenenen Daten reagieren können [ACCC02].
Aurora besitzt eine Vielzahl von eingebauten Operatoren, die als Basis der Anfrageerstellung dienen. Die Fensteroperatoren ermöglichen es Operationen auf Ausschnitte der Datenströme zu berechnen [Prok03]. Die Funktionsweise eines solchen Operators kann in drei Schritten zusammengefasst werden:

1. Eingabetupel werden in das Fenster (in Aurora eine Warteschlange) eingefügt.
2. Eine Operation wird auf die Tupel des Fensters ausgeführt.

3. Ein neuer Fensterausschnitt wird berechnet.

Welcher Ausschnitt berechnet wird, hängt vom jeweiligen Fensteroperator ab. Der *Slide*-Operator arbeitet nach dem Prinzip eines Sliding Windows. Mit ihm könnte man beispielsweise die Anzahl der Besucher einer Website in den letzten zwei Stunden bestimmen. Der Operator *Tumble* erreicht Ähnliches, versetzt das Fenster jedoch so, dass die erzeugten Tupelmengen von Fenster zu Fenster disjunkt sind. Dieser Operator eignet sich z.B. um die Anzahl der Besucher einer Webseite pro Tag zu bestimmen. Der *Latch*-Operator funktioniert wie der Tumble-Operator, kann jedoch Zwischenergebnisse zwischen Fenstern hinweg speichern. Somit ist es mit ihm möglich, Aggregatfunktionen wie die Summe, den Durchschnitt, das Maximum oder Minimum aller bisherigen Daten zu bestimmen. Der *Resample*-Operator erzeugt einen teils künstlichen Strom, indem er interpolierte Werte zwischen den Tupeln einfügt.
Zu den Fensteroperatoren gibt es zusätzlich die Filteroperatoren, welche auf den Strömen tupelweise arbeiten. Der *Filter*-Operator leitet nur solche Tupel an die Ausgabe weiter, die das Filterprädikat erfüllen. *Drop* ist ein spezieller Filter, welcher einen zuvor definierten Prozentsatz von zufällig ausgewählten Tupeln nicht weiterleitet. *Map* bildet jedes Tupel durch eine angegebene Funktion auf ein anderes ab. *GroupBy* nimmt mehrere Eingangsströme und gruppiert die enthaltenen Tupel zu neue Ströme, welche jeweils eine bestimmte Eigenschaft besitzen. So könnte man z.B. Personendaten in Ströme aufteilen, die jeweils nur Personen einer Altersgruppe enthalten. Der *Join*-Operator führt den Verbund auf diejenigen Tupel aus, welche einen gewissen *Abstand* (wie etwa einen zeitlichen Abstand) zueinander besitzen, der nicht über eine angegebene obere Schranke hinausgeht [CCCC02].
Zusätzlich zu den vordefinierten Anfragen erlaubt Aurora Ad-hoc-Anfragen, also solche, die zur Laufzeit gestellt werden. Selbst Anfragen zu Daten aus der Vergangenheit sind dabei möglich. Dazu müssen vorab im Anfragegraph *Verbindungspunkte* (*connection points*) gesetzt sein, an denen neue Operatoren zur Laufzeit angehängt werden können. Um vergangene Daten in den Anfragen mit aufnehmen zu können, speichern Verbindungspunkte Daten aus den Strömen für eine gewisse Zeit. Wenn also ein Datenstrom durch einen Verbindungspunkt fließt, so wird er dort für eine (vom Benutzer) vordefinierte Zeit in einem internen Cache gespeichert. So kann man z.B. bei einem Verbindungspunkt die Daten der vergangenen drei Stunden speichern. Ad-hoc-Anfragen an diesem Punkt können somit zu einem Zeitpunkt beginnen, der bis zu drei Stunden in der Vergangenheit liegt.

3.2 Implementierung und Optimierung

Abbildung 3.1 zeigt die wesentlichen Elemente des Aurora-Laufzeitsystems. Alle Tupel, die entweder aus den Datenquellen kommen oder Ausgaben von Operatoren sind, werden dem *Router* übergeben. Dieser leitet die Tupel an externe Anwendungen weiter oder übergibt sie dem *Speicher-Manager*, der dafür verantwortlich ist, die Tupel an weitere Operatoren zu leiten und ggf. in eine Warteschlange vor dem Operator einzufügen. Der *Scheduler* bestimmt welcher Operator auszuführen ist und übergibt dem *Box-Prozessor* (welcher in mehreren Threads laufen kann) einen Pointer auf diesen. Nach der Ausführung des Operators wird das Ergebnis wieder an den Router geleitet, und der Zyklus beginnt erneut.
Eine Besonderheit von Aurora ist die Beachtung von benutzerspezifizierten Dienstgütekriterien durch den *QoS-Monitor*. Dieser beobachtet fortlaufend die Systemleistung und aktiviert bei zu hoher Auslastung den *Load Shedder*, der dann gewisse Tupel auslässt. Aurora unterstützt folgende drei QoS-Kriterien:

1. *Aktualität*: Ergebnisse sollten mit so geringer Verzögerung ankommen, dass sie für den Benutzer noch brauchbar sind. Dies ist besonders wichtig für Echtzeitsysteme, welche innerhalb kurzer Zeit auf Sensordaten reagieren müssen.

2. *Vollständigkeit*: Die Ergebnistupel sollten möglichst wenig Lücken aufweisen. Gerade bei präzisen Analysen von Strömen mit hoher Datendichte (wie z.B. Audiodaten) ist dies von Bedeutung.

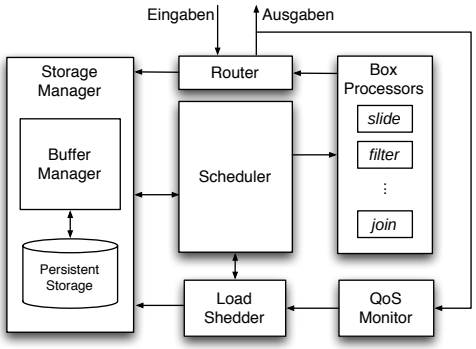

Abbildung 3.1: Das Aurora-Laufzeitsystem

3. *Wertegewichtung*: Manche Werte sind wichtiger als andere. Unwichtige Werte sollten beim Load-Shedding eher weggelassen werden, als die wichtigen. So sollten beispielsweise kritische Werte eines Kernreaktors auf keinen Fall ausgelassen werden.

Aurora ermöglicht es dem Benutzer diese drei Kriterien als zweidimensionale Funktionen anzugeben. Das Kriterium der Aktualität kann der Benutzer durch eine Funktion angeben, in der die akzeptable Verzögerung der Ausgabe spezifiziert ist. Abbildung 3.2 zeigt eine solche Funktion. Ist die Verzögerung geringer als der Schwellwert δ, so ist die QoS maximiert. Ist hingegen die Verzögerung größer als δ, so nimmt auch die QoS mit zunehmender Verzögerung ab. Der Benutzer kann weiterhin eine Funktion angeben, die auf der Anzahl der auszugebende Tupel basiert (Abbildung 3.3). Hier wurde angegeben, dass eine hohe QoS erreicht wird, wenn die Anzahl der ausgegebenen Tupel bei fast 100% liegt. Zuletzt gibt es noch die Möglichkeit eine Funktion anzugeben, die auf der Wichtigkeit der Werte basiert (Abbildung 3.4). Diese zeigt an, welche Werte vom Load-Shedder eher weggelassen werden können, als andere. Es bleibt noch zu erwähnen, dass Aurora von den angegebenen Funktionen erwartet, dass sie zur Laufzeit eingehalten werden können. Unrealistische Anforderungen an das System können zu einem Fehlverhalten der Optimierungsalgorithmen führen.

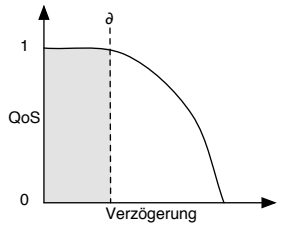

Abbildung 3.2: Die Verzögerungs Funktion

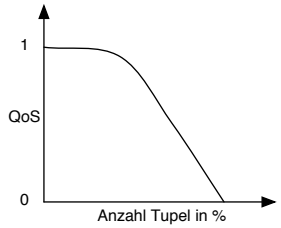

Abbildung 3.3: Die tupelbasierte Funktion

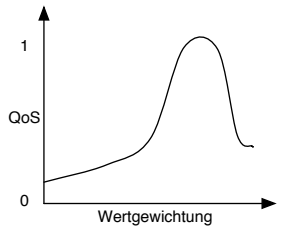

Abbildung 3.4: Die wertbasierte Funktion

Aurora kennt weiterhin eine Vielzahl von Optimierungsmöglichkeiten, welche alle zur Laufzeit stattfinden, ohne dass das System dabei heruntergefahren werden muss (*online optimization*). Dazu werden zur Laufzeit Daten über die aktuelle Performanz und Auslastung der Operatoren gesammelt. Es ist nicht sinnvoll den gesamten Operatorgraphen als eine einzige große Anfrage zu behandeln, und dann auf konventionelle Weise zu optimieren. Eine solche Optimierung wäre NP-vollständig [SeGh90] und damit zur Laufzeit nicht realisierbar. Stattdessen erfolgt eine Optimierung auf Ebene von durch Verbindungspunkte umschlossene Teilgraphen. Die Verbindungspunkte lassen dazu keine Tupel mehr hindurchfließen (sondern behalten diese in einer Warteschlange), sodass der Teilgraph vollständig leer laufen kann. Nachdem der Teilgraph keine Aktivitäten mehr aufweist, können Optimierungen innerhalb dieses Graphen vorgenommen werden. Eine davon ist das aus klassischen DBMS bekannte *Einfügen* oder *Vorziehen von Selektionen*. Um möglichst wenige Tupel verarbeiten zu müssen, werden Selektionen an den frühstmöglichen Stellen eingefügt. Diese entfernen alle Attribute, die im weiteren Anfragegraph nicht mehr benötigt werden. Eine weitere Optimierungsmöglichkeit ist das *Zusammenfügen von Operatoren*. Da jeder Operator eine gewisse Last an Verwaltung erzeugt, ist es wünschenswert die Anzahl von Operatoren zu minimieren. Besonders Teilgraphen mit vielen primitiven Operatoren erzeugen durch den entstehenden Overhead unnötig viel Last. Aurora versucht hier zu optimieren, indem es solche Operatoren zu einem grösseren und komplexeren Operator zusammenfasst. So können z.B. zwei aufeinanderfolgende Filteroperatoren durch Konjunktion ihrer Prädikate zu einem einzigen komplexeren Filter zusammengefasst werden. Eine weitere Möglichkeit der Optimierung ist das *Umsortieren von Operatoren*. Wie auch bei relationalen Datenbanksystemen üblich, kann man so (z.B. indem man einen Filter vor einen Verbund zieht) Anfragen deutlich optimieren. Aurora führt dazu eine aktuelle Kostenanalyse der Operatoren durch. Ob das Vertauschen zweier Operatoren zu einem effizienteren Graphen führen würde, kann anhand der jeweils ausgegebenen Anzahl von Tupel, und den Kosten pro Tupel berechnet werden. Die Effizienz der einzelnen Permutationen werden daraufhin miteinander verglichen, und die bestmöglichste Lösung ausgewählt. Nachdem der Optimierer alle sinnvollen Veränderungen zur Optimierung eines Teilgraphen gefunden hat, wird dieser neu konstruiert und an den Graphen wieder angeschlossen. Die umschließenden Verbindungspunkte lassen dann wieder Tupel hindurchfließen.

Kapitel 4

PIPES

In Umgebungen, in denen eine Vielzahl von Datenströmen mit hohen Durchsatzraten verarbeitet werden müssen, sind leistungsstarke Plattformen von zentraler Bedeutung. Es liegt nahe, dass gerade hier Multiprozessor-Systeme bzw. ganze Rechner-Cluster ideal einzusetzen wären. Voraussetzung ist jedoch, dass die Software mehrere Prozessoren sinnvoll einsetzen kann. Leider sind viele heutige DSVS-Implementierungen noch weit von einer idealen Thread-Aufteilung entfernt. Stattdessen wird meistens einer der folgenden beiden Lösungen verwendet:

1. *operatorbasiertes Scheduling*: Jeder Operator läuft in einem eigenen Thread. Bei kleineren Anfragegraphen kann man mit einer solchen Lösung durchaus hohe Performanz erzielen. Allerdings erzeugt jeder Thread auch einen gewissen Overhead, sodass komplexe Anfragesysteme möglicherweise nicht vom Threading profitieren, sondern im schlechtesten Fall sogar dadurch belastet werden.

2. *graphbasiertes Scheduling*: Hier läuft der gesamte Graph einer Anfrage in einem einzigen Thread. In diesem Thread wählt der Scheduler aus, welche Operation als nächstes auszuführen ist. So wird zwar kaum Overhead erzeugt, jedoch kann auch pro Anfrage nur ein Operator gleichzeitig ausgeführt werden. Dies wird insbesondere dann problematisch, wenn ein Operator ein hohes Maß an Zeit benötigt um eine Eingabe zu verarbeiten. Im schlimmsten Falle könnte dies sogar zu einem Überlauf von Eingangspuffern oder zu einem Stagnieren der Ausgabe führen. Weiterhin kann so eine einzelne Anfrage nicht von mehreren im System vorhandenen Prozessoren profitieren.

Das *PIPES*-System verfolgt einen anderen Ansatz. Entwickelt von der Universität Marburg ist PIPES kein ausführbares Programm, sondern eine Java-Bibliothek. Diese enthält eine Vielzahl von generischen Operatoren, die speziell für die Verarbeitung von Datenströmen konzipiert worden sind. Als Basis wurde die Bibliothek *XXL* verwendet, welche komplexe Methoden und Datenstrukturen zur Datenverarbeitung bereitstellt. PIPES kann weiterhin durch objektorientierte Paradigmen erweitert und konfiguriert werden. Um Threads unter den Operatoren sinnvoller aufzuteilen verwendet PIPES das sog. *hybrid multi-threaded scheduling* (*HMTS*). Dieses versucht die Vorteile der beiden zuvor beschriebenen Ansätze miteinander zu verbinden.

4.1 Anfrageerstellung

Wie üblich besteht ein Anfragegraph in PIPES aus einer Menge von Datenquellen, aus denen Datenströme über Operatoren (hier *Pipes* genannt) zu den Schnittstellen der Anwendungen (*Sinks*), fließen. Abbildung 4.1 zeigt die Veranschaulichung eines Anfragegraphs in PIPES.
Drei Java-`Interfaces` bilden hierzu das Grundgerüst der Bibliothek. Das Interface `Source` wird von allen Klassen implementiert, welche als Quellen im Anfragegraph operieren. Sie sind dafür verantwortlich, erzeugte Elemente an verbundene Knoten weiterzuleiten. Das Interface `Sink` wird von den Klassen implementiert, welche die Schnittstelle zur Anwendung bilden. Sie müssen die

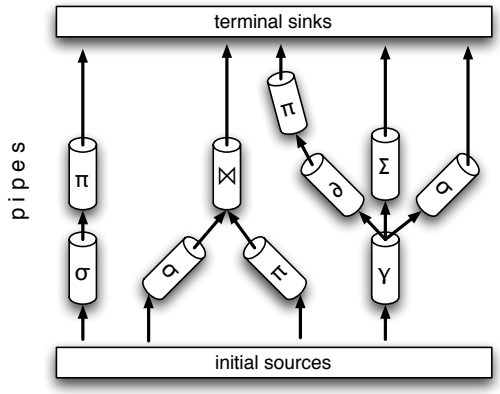

Abbildung 4.1: Anfragegraph in PIPES

Annahme von Elementen zur weiteren Verarbeitung implementieren. Das `Pipe`-Interface erweitert sowohl das `Source`- wie auch das `Sink`-Interface, und erlaubt somit die Verarbeitung von Eingaben und die Weiterleitung der Ausgaben. Damit der Benutzer nicht bei jeder Erweiterung das gesamte Interface implementieren muss, werden abgeleitete Klassen bereitgestellt, welche die Grundfunktionalität der Knotentypen implementieren. Diese sind `AbstractSource`, `AbstractSink`, und `AbstractPipe`. `AbstractSource` wie auch `AbstractPipe` aggregieren eine beliebige Anzahl von `Sink`-Objekten [XXLb2]. Dadurch wird erst eine Graphbildung ermöglicht. Abbildung 4.2 zeigt die eben vorgestellte Klassenhierarchie.

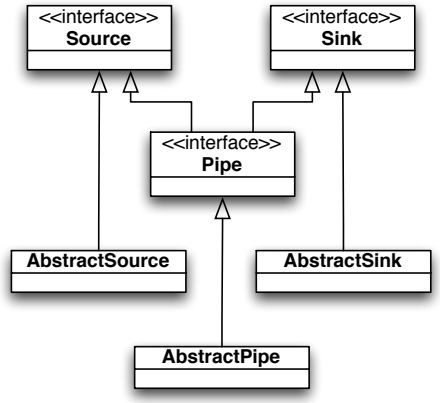

Abbildung 4.2: Die PIPES-Klassenhierarchie

Um Datenverarbeitungslogik der Operatoren zu implementieren, überschreibt der Benutzer die `Sink`-Methode `process`. Man erhält in ihr als Parameter das zu verarbeitende Objekt (vom Typ `Object`) und eine eindeutige ID der Quelle des Objekts. Subklassen von `Source` (und damit auch jede `Pipe`) können über eine Methode `transfer` Ergebnisse an weitere Knoten leiten. Diese Methode ruft rekursiv die `process()` Methode der verbundenen `Sink`-Objekte auf. Dies ist ein ungewöhnlicher Ansatz, denn es werden keine Warteschlangen zwischen den Knoten eingesetzt. Weiterhin wird kein explizites Scheduling nötig, da dieses bereits implizit von der Operatoren erledigt wird. PIPES nennt diese Art der direkten Weiterleitung *direkte Interoperabilität* (*direct interoperability*). Folgender Beispielcode zeigt die Implementierung eines Filters welches nur Zahlenwerte der Klasse Integer weiterleitet:

```
public class Filter extends AbstractPipe {

  public void process(Object o, int ID) {
    if(o instanceof Integer)
    transfer(o);
  }

}
```

PIPES bringt schon eine Vielzahl von Implementierungen der vorgestellten Interfaces mit sich. Um Anfragen sinnvoll testen zu können liegen einige synthetische Quellen bereit. Diese erzeugen zufällige Daten in entweder relativ gleichmäßigem Abstand, oder aber in unregelmäßigen Schüben. So lässt sich die Skalierbarkeit und Belastbarkeit des Systems testen. Einige `Sink`-Subklassen sind implementiert, welche die eintreffenden Daten entweder textuell darstellen, oder in einen Java-PrintStream zur weiteren Verarbeitung ausgeben. Selbstverständlich bietet PIPES auch eine Vielzahl an vorprogrammierten Operatoren, darunter gängige wie *filter*, *map*, und *join* [CHKM03].

4.2 Implementierung und Optimierung

Um zustandsbehaftete Operatoren wie den Verbund zu realisieren, verwenden diese Zwischenspeicher, welche in PIPES *SweepAreas* genannt werden. So hat der Verbundoperator zu jeder Eingabe eine SweepArea. Beim Eintreffen eines neuen Elements wird dieses in die zugehörige SweepArea eingefügt, und gegen die Elemente der anderen SweepAreas getestet. PIPES bietet eine Reihe von SweepArea-Typen an, welche jeweils eine unterschiedliche Fenstersemantik implementieren.
Um eine optimale Verteilung der Operatoren in Threads zu erreichen, benutzt PIPES das eigens entwickelte *Hybrid Multi-Threaded Scheduling*, kurz *HMTS*. Einerseits, wird durch dieses Konzept ermöglicht, Teile des Anfragegraphs in eigene Threads laufen zu lassen, sodass selbst langsame Operatoren, nicht zu einem Blockieren des ganzen Systems führen können. Andererseits, wird nicht *jedem* Operator ein eigener Thread zugeteilt, sodass nur ein minimaler Overhead entsteht.
Sobald Teile des Graphen in verschiedenen Threads laufen, wird ein explizites Scheduling nötig, sodass die Operatoren voneinander entkoppelt werden müssen. Dazu werden zwischen den Operatoren verschiedener Threads Warteschlangen eingerichtet. In PIPES implementiert die Klasse `BufferPipe` eine solche Warteschlange. Diese können zur Laufzeit beliebig in den Graphen eingefügt und entfernt werden. BufferPipes sind, wie der Name schon verrät, von `Pipe` abgeleitet. Im Gegensatz zu vielen anderen `Pipe`-Subklassen führen BufferPipes ihre `transfer`-Methode nicht selbständig aus, da dies vom Scheduler übernommen wird.
Um HMTS auf einen Anfragegraphen anzuwenden, wird dieser in drei Ebenen unterteilt. Auf der ersten Ebene besteht der Graph aus *physischen* und *virtuellen* Operatoren. Physische Operatoren sind die eben beschriebenen Subklassen von `Pipe`. Die Verkettung der physischen Operatoren über direkte Interoperabilität führt zur Bildung eines virtuellen Operators. Dieser darf keine Warteschlangen enthalten. Abbildung 4.3 zeigt die Verkettung physischer Operatoren zu virtuellen Operatoren.

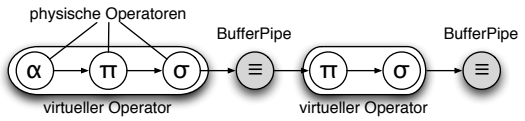

Abbildung 4.3: Physikalische und virtuelle Operatoren

Auf der zweiten Ebene, wird der gesamte Graph in Teilgraphen unterteilt. Vor jedem Teilgraph wird eine BufferPipe angelegt. Um die BufferPipes zu steuern verfügt jeder Teilgraph über einen `BufferPipeScheduler`. Dieser wählt zur Laufzeit eine BufferPipe aus (z.B. die mit den meisten

Elemeneten), und ruft zur Weiterleitung der Elemente ihre `transfer`-Methode auf. Welche Strategie zur Auswahl eines Operators verwendet wird, hängt von der jeweiligen Implementierung des BufferPipeSchedulers ab. Abbildung 4.4 veranschaulicht die Funktionsweise eines BufferPipeSchedulers.

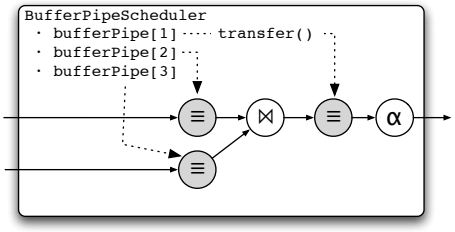

Abbildung 4.4: Aufbau eines BufferedPipeSchedulers

Die dritte Ebene realisiert schließlich das gleichzeitige Abarbeiten der Teilgraphen in Threads. Dazu wird ein `ThreadScheduler` konstruiert, welcher in einem eigenen Thread läuft, und die Prioritäten der anderen Threads steuert. Ziel ist es diejenigen Threads zu priorisieren, welche die größte Datenlast zu verarbeiten haben [CHKM03]. Abbildung 4.5 zeigt noch einmal den Aufbau der drei Ebenen.

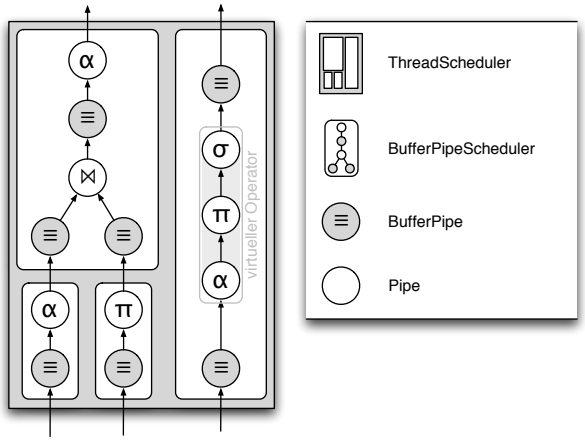

Abbildung 4.5: Die drei Ebenen von HMTS

Kapitel 5

SPEX

Die bisher vorgestellten Systeme betrachteten einen Datenstrom als eine potenziell unendliche Sequenz von Tupeln. Die Möglichkeit von komplexeren nicht-flachen Datenstrukturen wurde bisher nicht berücksichtigt. Mit der wachsenden Beliebtheit von *XML* werden jedoch komplexer strukturierte Daten immer häufiger verwendet.
XML-Ströme erlauben es Datensätze zu spezifizieren, deren Größe und Schachtelungstiefe möglicherweise unbegrenzt sind, und denen eine rekursiv definierte Struktur zu Grunde liegen kann. Wenn diese Datensätze unterschiedlich groß sind, in unterschiedlicher Weise geschachtelt sind oder eine unbestimmte Schachtelungstiefe haben, dann stellt ihre Verarbeitung eine besondere Herausforderung dar [BrFO04].
Es gibt inzwischen zahlreiche Anwendungen, welche XML-Datenströme erzeugen. Ein konkretes Beispiel hierfür ist der Prozess-Monitor (`ps -elfH`[1]) des UNIX-Betriebssystems. Dieser erlaubt es die aktuellen Prozessdaten als XML-Datei oder als kontinuierlichen Datenstrom auszugeben. Dieser Strom ist in der Größe und Tiefe unbegrenzt, da die aktuellen Prozessdaten kontinuierlich ausgegeben werden, und jeder Prozess eine beliebige Anzahl von Kindprozessen besitzen kann. Sicherlich interessieren einen Administrator nicht alle ausgegebene Werte, sondern nur solche, die beispielsweise in einem kritischen Bereich liegen (hier: eine hohe Prozessorauslastung) oder nur solche, die Teilprozess eines bestimmten Programms sind. Um diese Informationen zu extrahieren reichen tupelbasierte DSVS nicht aus.
Ein weiteres aktuelles Beispiel von XML-basierter Datenstromverarbeitung ist das Filtern von *RSS*-Strömen. RSS wird dazu verwendet, Kurzbeschreibungen von Web-Inhalten (wie z.B. Nachrichten) zusammen mit einem Verweis auf die Quelle zu erstellen. Diese Informationen werden als XML-Daten ausgegeben, genannt RSS-*Feeds* oder RSS-*Streams*. Hier wäre es z.B. sinnvoll genau die Feeds herausfiltern zu können, die ein bestimmtes Schlagwort enthalten.
SPEX ist ein DSVS, entwickelt von der Technischen Universität München, welches auf XML-Datenströmen arbeitet. Durch die Verwendung der *XPath*-Anfragesprache, erlaubt SPEX komplexe und vielseitige Anfragen auf unbegrenzten Datenströmen [BCDF05].

5.1 Anfrageerstellung

Die Verwendung von XPath erlaubt die Adressierung und Lokalisierung von Teilen eines XML Dokumentes. Es wird dabei eine Syntax verwendet, die der Spezifikation eines Dateipfades unter UNIX angelehnt ist [W3C99]. XPath findet seinen Einsatz vor allem in den Standards XSLT, XPointer und XQuery.
Eine XPath-Anfrage besteht aus einer beliebigen Anzahl von *Pfadschritten*, welche durch einen *Slash* (/) getrennt werden. Ein Pfadschritt besteht aus den folgenden Komponenten:

[1] Diese Funktionalität wird nicht in allen Implementierungen des `ps`-Befehls unterstützt. So reichte auf der Darwin-Plattform die Funktionalität nicht aus um dieses Beispiel zu rekonstruieren.

1. Eine *Achse* (*axis*): Achsen treffen eine weitere Auswahl von Knoten, die in einem bestimmten Verhältnis zum aktuellen Knoten (*Kontextknoten*) stehen. So bezeichnet beispielsweise die `child`-Achse alle direkten Nachkommen des Kontextknotens. Eine wichtige Unterscheidung ist die in *Vorwärts-* und *Rückwärtsachsen* (*foward and reverse axes*). Vorwärtsachsen enthalten ausschließlich Knoten, die in der XML-Datei in der Dokumentordnung *nach* dem Kontextknoten (also Richtung Ende des Dokuments) stehen. Die `child`-Achse ist somit eine Vorwärtsachse. Rückwärtsachsen enthalten entsprechend ausschließlich Knoten, die im XML-Dokument *vor* dem Kontextknoten (also Richtung Anfang des Dokuments) stehen. Ein Beispiel für eine Rückwärtsachse ist die `ancestor`-Achse, welche alle übergeordnete Knoten (Elternknoten) des Kontextknotens auswählt.

2. Ein *Knotentest* (*node test*): Knotentests (geschrieben `Achse::Knotentest`) schränken die Elementauswahl einer Achse ein. So würde `child::Buch` all die Nachkommen des Kontextknotens auswählen, die den Namen "Buch" haben.

3. Optional ein *Prädikat* (*predicate*): Durch Angabe von Prädikaten, kann das Ergebnis weiter eingeschränkt werden. Prädikate werden in eckige Klammern eingeschlossen und können in beliebiger Zahl hintereinander geschrieben werden. Prädikate können Funktionen und geschachtelte XPath-Ausdrücke enthalten [WikiXP].

Die Anfrage *"Finde alle Buch-Knoten, die Kind des Wurzelknotens Buchladen sind"* wird in XPath folgendermaßen gestellt:

```
/child::Buchladen/child::Buch
```

Da die `child`-Achse die Standardachse ist, kann diese Angabe auch weggelassen werden. Die Anfrage könnte dann verkürzt aufgeschreiben werden zu:

```
/Buchladen/Buch
```

Folgender Code zeigt eine XPath-Anfrage mit Vorwärtsachse und Prädikat:

```
/Buchladen/desendant::Buch[Preis < 35]
```

Dies entspricht der Anfrage *"Finde alle Nachkommen von Buchladen, welche Buch-Knoten sind, und einen Preis größer als 35 haben"*. `Preis` muss dabei ein Kind von `Buch` sein.
XPath-Anfragen können in einem Prototyp einer grafischen Benutzeroberfläche, namens *SPEX Viewer*[2] gestellt werden. Dieser ermöglicht die Verfolgung eingegebener Anfragen, Zwischenschritte zu analysieren, und Ausgaben zu ermitteln. Der SPEX Viewer soll vor allem als Debugger und Präsentationsplattform zum Einsatz kommen und soll im Gegensatz zur Aurora-Oberfläche nicht zum grafischen Erstellen einer Anfrage dienen. Anfragen in SPEX können ausschliesslich über XPath gestellt werden.

5.2 Implementierung und Optimierung

Um eine XPath-Anfrage in ein für SPEX verwendbares Format zu transformieren sind insgesamt drei Schritte notwendig. Der erste Schritt ist die Transformation der Anfrage in eine *Vorwärtsanfrage* (*forward query*). Anfragen mit Rückwärtsachsen würden ein Speichern aller bisher erfassten Elementen erfordern, was bei einem potenziell unendlichem Strom nicht realisierbar wäre. Daher ist es nötig, alle Rückwärtsachsen der XPath Anfrage in semantisch äquivalente Anfragen mit Vorwärtsachsen umzuschreiben. Da es eine solche Anfrage zu jeder beliebigen Anfrage mit Rückwärtsachsen gibt, ist ein solches Umschreiben immer möglich. Abbildung 5.1 zeigt die Konvertierung einer Rückwärtsanfrage in eine Vorwärtsanfrage.
Im zweiten Schritt wird die Anfrage in einen logischen azyklischen Graphen verwandelt. Dieser enthält zwei Arten von Knoten. Solche, die Prädikate darstellen, welche von den hindurchfließenden

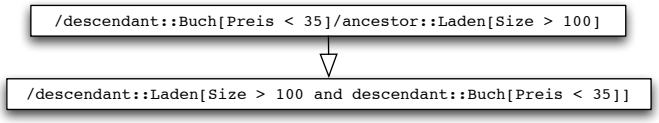

Abbildung 5.1: Umwandlung der Anfrage *"Finde alle Läden mit einer Größe über 100, die ein Buch enthalten, welches billiger als 35 ist."* in eine Vorwärtsanfrage.

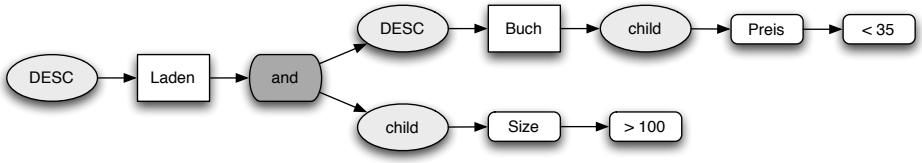

Abbildung 5.2: Der logische Anfrageplan des Beispiels

Elementen erfüllt sein müssen, und solche, welche die Antworten extrahieren, die der Benutzer sucht. Abbildung 5.2 zeigt den Anfragegraph für das oben genannte Beispiel.
Im dritten und letzten Schritt wird der logische Anfrageplan in einen physischen Plan transformiert. Dies geschieht in zwei Teilschritten. Zunächst werden alle Knoten des logischen Anfragegraphs in einen entsprechenden *Modifikationsoperator* verwandelt. Diese modifizieren oder annotieren ankommende Daten, und schicken diese an verbundene Operatoren weiter. So annotiert der *and*-Operator Elemente mit Bedingungen, die zu erfüllen sind. An den Anfang des Graphes wird ein *in*-Operator eingefügt, der die Daten aus dem XML-Strom in den Anfragegraphen leitet. Weiterhin werden die Ergebnisse in einem sogenannten *Trichter* am Ende der Anfrage gesammelt. Dieser besteht in den meisten Fällen zunächst aus verifizierende Operatoren, wie dem *cd-and*, welcher überprüft ob die annotierten Bedingungen erfüllt sind oder nicht. Das Ende des Trichters bildet ein *out*-Operator, der die Ergebnisse ausgibt [OlFB04]. Abbildung 5.3 zeigt die Beispielanfrage als physischen Anfrageplan.

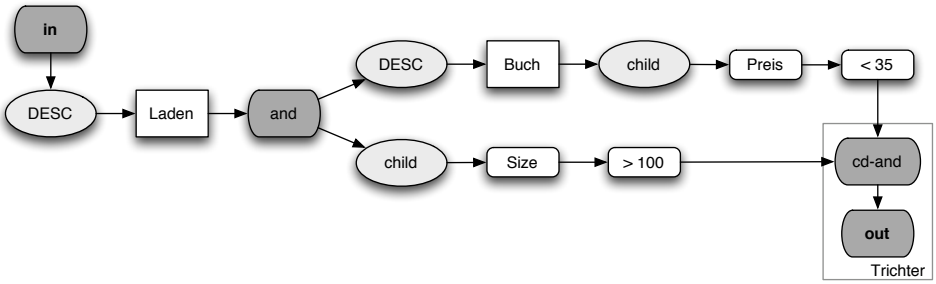

Abbildung 5.3: Der physische Anfrageplan des Beispiels

Um die einzelnen Transformationsschritte der Anfrage genau verfolgen zu können, kann der SPEX Viewer verwendet werden. Dieser stellt jeden Schritt der Transformation grafisch dar. Zusätzlich kann die Auswertung des XML-Stromes jederzeit angehalten oder verlangsamt werden, damit eine sinnvolle Analyse und Fehlerbehebung möglich ist [BCDF05].

[2]Der SPEX-Viewer wird bisher nur intern benutzt, und ist noch nicht über das Internet verfügbar

Kapitel 6

MAIDS

In Datenstromsystemen mit extrem hohen Datenraten ist es oft nicht möglich alle Datensätze ohne Lücken zu erfassen. So fallen in manchen Telekommunikationssystemen mehrere hundert Gigabyte Verbindungsdaten am Tag an [RaGG02]. Bei solch hohen Datenraten sind Rückfragen des Systems an die Datenquellen oft nicht möglich, sodass Daten teilweise lückenhaft abgelegt werden müssen. Bei solchen Systemen ist es jedoch potentiell möglich die fehlenden Daten mit einer hohen Wahrscheinlichkeit korrekt zu "erraten". Denn die Vielzahl an Werten, die durch das System bereits geflossen sind, geben einen Hinweis darauf welcher Wert wahrscheinlich korrekt ist. So könnte beispielsweise eine statistische Erfassung der Netzwerknutzung aus den Datenströmen eines Netzwerkbetreibers erstellt werden, und aus diesen Werten die Fehlenden mit einer gewissen Wahrscheinlichkeit vorausgesagt werden.
Eine solche Erkennung von Mustern in großen Mengen von Daten bezeichnet man als *Data Mining* [IWEF03]. Dieses eignet sich nicht nur zur Wertergänzung, sondern auch um Daten anhand ihrer Attributwerte zu *klassifizieren*. Hierzu wird eine Relation mit einem Gruppenattribut versehen, und mit *Trainingsdaten* gefüllt. Auf die gleiche Weise wie Werte ergänzt werden, kann man so eine Zugehörigkeitsgruppe eines neuen Attributs berechnen. Ein Versandhaus könnte damit neue Kunden anhand ihrer persönlichen Daten als Viel- oder Wenigbesteller einstufen, und ihnen ggf. einen Warenkatalog zuschicken.
Data Mining findet weiterhin seine Anwendug in der Erkennung von wiederkehrenden Mustern in Daten. So könnten beispielsweise auftretende Muster in Netzwerkaktivitäten analysiert werden, um etwa die Aktivitäten von Computer-Viren aufzudecken.
Gemeinsam entwickelt von der Automated Learning Group, NCSA und der University von Illinois verfolgt das MAIDS-System (Mining Alarming Incidents for Data Streams) das Ziel *Data Mining* auf Datenströmen zu ermöglichen. Dazu müssen Data-Mining-Algorithmen so angepasst werden, dass sie in Echtzeit bearbeitet werden können und nicht zu übermäßigem Speicherkonsum führen. Das MAIDS-System befindet sich noch in der frühen Entwicklung, und kann daher hier nicht im Detail beschrieben werden. Es sollen jedoch die grobe Funktionsweise und einige interessante Aspekte des Systems herausgegriffen werden.

6.1 Anfrageerstellung

Datenströme, die in das MAIDS-System einfließen, werden zunächst formatiert, normalisiert und transformiert, bevor sie an die in Abbildung 6.1 dargestellten vier Modulen weitergeleitet werden. Die *Anfrage-Engine* (*Stream Query Engine*) erlaubt es, eine Vielzahl von möglichen Anfragetypen, darunter fortlaufende und Ad-hoc-Anfragen, exakte oder approximierte Anfragen, an das System zu stellen. Der *Datenklassifizierer* (*Stream Data Classifier*) konstruiert Klassifikationsmodelle anhand der vorhandenen Daten des Zeitfensters. Damit wird es möglich einströmende Daten aufgrund ihrer Attributwerte in eine Klasse bzw. Gruppe einzuteilen. Die *Musterkennung* (*Stream Pattern Finder*) versucht, wiederkehrende Muster in dem Datenstrom zu erkennen und

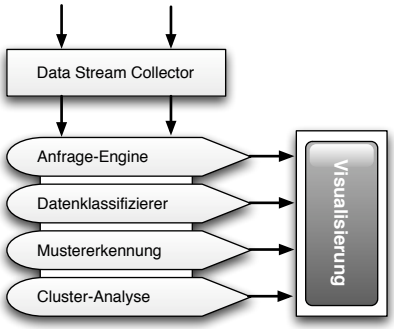

Abbildung 6.1: Aufbau des MAIDS-System

herauszufiltern.
MAIDS nutzt zur Datenspeicherung das Prinzip eines *geneigten Zeitfensters* (*tilted time window*). Dieses funktioniert ähnlich zu einem herkömmlichen Zeitfenster mit dem Unterschied, dass Daten in verschiedenen Granulaten gespeichert werden. MAIDS geht davon aus, dass Daten, die weit in der Vergangenheit liegen, weniger präzise sein müssen als die aus der jüngeren Vergangenheit. Daher werden weit zurückliegende Daten nur in großen Zeitabständen gespeichert, sodass die Anzahl der Tupel pro Zeiteinheit abnimmt, je weiter man in die Vergangenheit schaut. MAIDS erreicht dadurch ein Zeitfenster mit relativ stabiler Grösse, ohne dabei die Möglichkeit aufzugeben, vergangene Daten zu analysieren. Abbildung 6.2 veranschaulicht den Aufbau eines geneigten Zeitfensters.

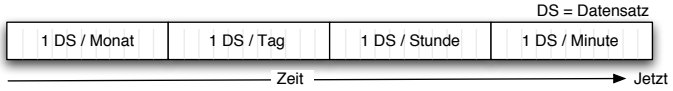

Abbildung 6.2: Granularitätsstufen eines geneigten Zeitfensters

6.2 Implementierung und Optimierung

Aufgrund des großen Umfangs und des frühen Entwicklungsstadiums von MAIDS sollen hier lediglich einige Konzepte des Systems herausgegriffen und erläutert werden.
Einer der grundlegenden Datenstrukturen des MAIDS-Systems ist die des geneigten Zeitfensters. Es war den Entwicklern wichtig, dass dieses möglichst effizient implementiert wird.
Ein geneigtes Zeitfenster besteht aus einer Menge von *zyklischen Listen* (*circular queues*). Jeder dieser Listen entspricht einer zeitlichen Granularität. Abbildung 6.3 zeigt ein solches Zeitfenster mit vier Granularitätsstufen: Minute, Viertelstunde, Stunde, Tag.

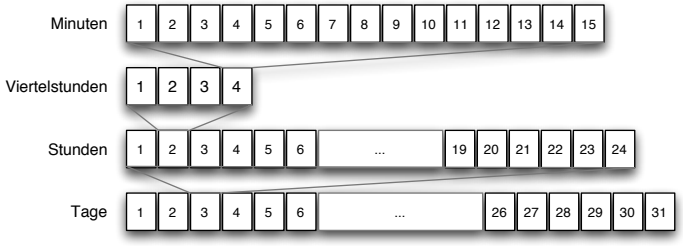

Abbildung 6.3: Die vier zyklischen Listen des Zeitfensters

Jede Liste muss genug Einträge besitzen, um insgesamt die Zeitspanne der nächsthöheren Stufe ausfüllen zu können. So umfasst die Minutenliste in unserem Beispiel 15 Einträge, die Viertelstundenliste vier Einträge, die Stundenliste 24 Einträge, und so weiter. Jede Liste besitzt zusätzlich einen *Zeiger* (*pointer*) der zunächst auf das erste Element der Liste zeigt. Zu Beginn sind alle Listen mit Nullwerten gefüllt. Wird nun ein erster Datensatz zum Zeitpunkt t gelesen, so wird dieser in das erste Feld der Liste mit der feinsten Granularität eingetragen. Wird ein weiterer Datensatz zum Zeitpunkt $t + \delta$ gelesen, so wird der Zeiger um δ Schritte verschoben, und der Datensatz an die neue Stelle eingetragen. Sobald der Zeiger über den Rand der Liste hinausläuft, springt er wieder an den Anfang der Liste (daher der Name "*zyklische* Listen"). Noch bevor er den neuen Wert schreibt, werden die Daten der Liste zusammengefasst und in die nächsthöhere Granularitätsstufe geschrieben. Die Liste wird dann wieder mit Nullwerten gefüllt, und der neue Wert an die Zeigerposition geschrieben [MTTW05].

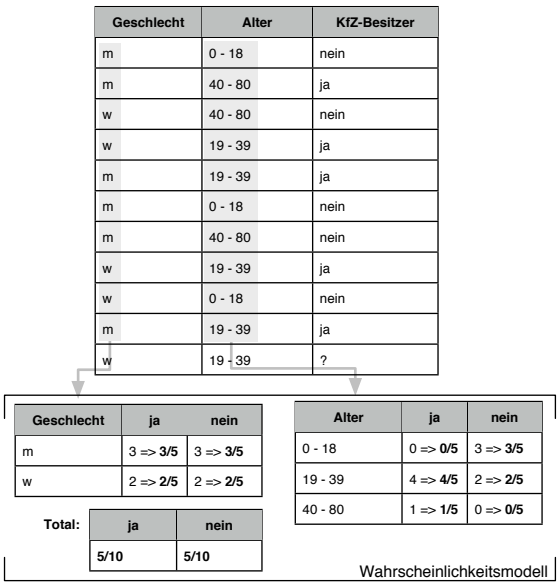

Abbildung 6.4: Die Erstellung eines Modells nach *Naive Bayes*

Zur Datenklassifikation verwendet MAIDS den *naiven Bayes-Algorithmus*. Dieser ist relativ einfach zu berechnen, und verbraucht zudem wenig Speicher. Um einen Wert prognostizieren zu können, muss zunächst ein *Wahrscheinlichkeitsmodell* (*probability model*) erstellt werden [IWEF03]. Abbildung 6.4 zeigt, wie aus vorangegangenen Werten ein Modell zur Ermittlung der Wahrscheinlichkeit für den Besitz eines Automobils in Abhängigkeit von Geschlecht und Alter erstellt werden kann. Dazu wird zunächst gezählt wie oft ein Attributwert zu einem Wert (`ja`/`nein`) vorkommt. In der Abbildung gibt es für das Geschlecht `m` dreimal einen `ja`-Wert und dreimal einen `nein`-Wert. Die untere Tabelle der Abbildung zeigt die Werte umgerechnet zu *beobachteten Wahrscheinlichkeiten* (*observed probabilities*). So sind von den insgesamt fünf Werten bei denen `Kfz-Besitzer = ja` ist, drei `Geschlecht = m`. Das ergibt somit einen Wert von 3/5. Die Tabelle `Total` gibt an wie oft ein Wert insgesamt im Verhältnis vorkommt. So ist hier `Kfz-Besitzer = ja` für fünf der zehn Fälle. Abbildung 6.5 zeigt weiterhin, wie aus diesem Modell ein Klassifikationswert approximiert werden kann. Dazu werden die Wahrscheinlichkeitswerte aus dem Modell zu den vorhandenen Attributwerte herangezogen. Dies wird sowohl für den `ja`- als auch für den `nein`-Wert gemacht. Diese miteinander multipliziert ergeben die gewünschte Wahrscheinlichkeit für den jeweiligen Wert.
Hier hat `ja` eine höhere Wahrscheinlichkeit als `nein` und wird daher als wahrscheinlichster Wert eingesetzt. In MAIDS wird das Modell auf Anfrage erstellt. Der Benutzer legt dabei fest aus welchem Zeitraum Daten zur Wahrscheinlichkeitsberechnung herangezogen werden sollen. So könnte der Benutzer beispielsweise angeben, nur Daten der vergangenen 16 Stunden oder Daten des Zeitraums Juni - August 2001 in das Modell aufzunehmen. Diese Daten werden dann aus dem

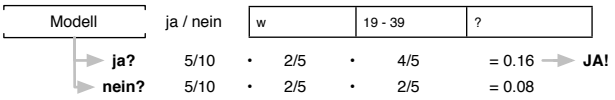

Abbildung 6.5: Die Berechnung eines Wertes aus einem Modell

angegebenen Bereich des geneigten Zeitfensters zur Berechnung herangezogen. Aus diesem Modell können nun fortlaufend Wertvorhersagen über Datenströme gemacht werden. Es können vom Benutzer beliebig viele Modelle erstellt werden, welche jeweils Werte vorhersagen [Clut03].
Die beiden Ansätze zeigen die Bedeutung auf effiziente Strukturen und Algorithmen, jedoch auch wo die Grenzen der Möglichkeiten liegen. Dem geneigten Zeitfenster liegt trotz seiner komplexeren Struktur ein einfacher Algorithmus zugrunde, der nahezu keinen Overhead erzeugt. Die Wertvorhersage ist effizient genug um fortlaufende Prognosen zu errechnen. Zu einer kontinuierliche Berechnung des Modells ist das System allerdings nicht in der Lage.

Kapitel 7

Fazit

Die vorgestellten Systeme zeigen, dass es bereits eine Vielzahl von Möglichkeiten gibt, Datenströme effizient und flexibel zu verwalten. Mit der Fenstersemantik lassen sich blockierende Operatoren auf Ausschnitten der Datenströme umsetzen. Durch Load Shedding wird auch bei hoher Datenlast ein reibungsloser Ablauf ohne Überläufe garantiert. Eine Vielzahl von Adaptierungsmöglichkeiten erlauben es, das System zur Laufzeit automatisch an die Verhältnisse anzupassen.
Obwohl die hier vorgestellten Systeme ähnliche datenstromtypische Komponenten einsetzen, verfolgen sie individuelle Ansätze und Konzepte diese zu verarbeiten.
Das STREAM-System wurde zur Erforschung von Methoden zur effizienten Verarbeitung zahlreicher Datenströme auf Plattformen mit begrenzter Leistung entwickelt. Zur Anfrageerstellung wurde für das System die Sprache CQL entwickelt, welche als Erweiterung zu SQL für Nutzer klassischer Datenbanken schnell zu erlernen ist. Das STREAM-System dient heute weiterhin hauptsächlich der Forschung und steht als Prototyp samt einer grafischen Benutzeroberfläche zum kostenlosen Herunterladen[1] bereit.
Im Gegensatz zu STREAM verwendet das Aurora-System keine Anfragesprache. Stattdessen wird dem Benutzer ein Datenflusssystem geboten, welches mit einer intuitiven grafischen Oberfläche gekoppelt ist. In dieser können Anfragepläne bestehend aus Kästchen (Operatoren) und Pfeilen (Warteschlangen) zusammengestellt und ausgeführt werden. Die Möglichkeit, Dienstgütekriterien mit in die Verarbeitung und Optimierung aufzunehmen, macht Aurora besonders im Bereich der Überwachungssysteme attraktiv. So wird derzeit Aurora im Militär zur Überwachung von feindlichen Truppenbewegungen eingesetzt. Alarmierende Informationen erhalten dabei eine hohe Priorität um selbst bei hoher Systemauslastung unverzüglich gemeldet zu werden.
PIPES kann zwar als Java-Bibliothek mit der einfachen Handhabung von Aurora nicht mithalten, ist jedoch als solches auch weitaus flexibler. Durch die Vielzahl von vordefinierten Klassen kann ein Programmierer vorhandene Operatoren anpassen oder durch die Erweiterung abstrakter Klassen völlig neue Operatoren entwerfen und implementieren. Weiterhin bietet PIPES einen flexiblen Threading-Ansatz, der es erlaubt einen Anfragegraphen in mehrere Threads zu unterteilen, ohne dabei jeden Operator in einen separaten Thread laufen lassen zu müssen.
Um XML-Datenströme verarbeiten zu können, reichen die Fähigkeiten der eben genannten Systeme nicht aus. Es sind auf XML angepasste Algorithmen und Anfragesprachen erforderlich um die Verarbeitung solcher Ströme zu ermöglichen. Das SPEX-System ist ein DSVS, welches in der Lage ist, XPath-Anfragen auf XML-Datenströmen auszuführen. Die fast vollständige Unterstützung des XPath-Standards (der Version 1.0) macht das System zu einem mächtigen Werkzeug zur Analyse von XML-Strömen. Zur Zeit ist SPEX noch in der Entwicklung, und kann nicht getestet oder heruntergeladen werden.
Das MAIDS-System implementiert Data-Mining-Konzepte auf Datenströmen, und bietet dazu einen ausgereiften Visualisierer. MAIDS befindet sich noch derzeit in der Entwicklung, und kann weder getestet noch gekauft werden.

[1] http://www-db.stanford.edu/stream/

Abbildung 7.1 soll noch einmal die Merkmale und Unterschiede der vorgestellten Systeme zeigen.

	STREAM	Aurora	PIPES	SPEX	MAIDS
Autoren	University of Stanford	M.I.T., Brandeis, Brown Universities	Universität Marburg	Universität München	University of Illinois
Anfragesprache	CQL (Continuous Query Language)	Workflow-System "Boxes and Arrows"	*keine*	XPath	keine Angaben
kontinuierliche Anfragen	ja	ja	ja	ja	ja
einmalige Anfragen	ja	nein	nein	nein	ja
vordefinierte Anfragen	nein	ja	ja	ja	ja
Ad-hoc-Anfragen	ja	ja, an Verbindungspunkte	ja, durch Anbinden neuer Pipes	nein (2)	ja
Fenstersemantik	mengen-/zeitbasiert	mengen-/zeitbasiert	mengen-/zeitbasiert	keine	zeitbasiert
zustandsbehaftete Operatoren	ja, durch *Synopsen*	ja, durch interne Puffer	ja, durch *SweepAreas*	ja, durch Stacks	keine Angaben
Threading	GTS (1)	GTS (1)	HMTS	keine Angaben	keine Angaben
Optimierungs-möglichkeiten	algebraisch/adaptiv	algebraisch/adaptiv/QoS	Nur durch den Programmierer	durch Caches	keine Angaben
Entwicklungsstatus	Released (vers. 0.6.0)	Released (vers. 1.2)	Released (vers. 1.2beta / 1,0 stable)	In Entwicklung / Kein Release	In Entwicklung / Kein Release

(1) Beide Systeme beschreiben in frühren Versionen einen OTS Ansatz, später jedoch GTS.
(2) Der aktuelle SPEX Viewer erlaubt keine neuen Anfragen, sobald ein Strom analysiert wird.

Abbildung 7.1: Die Systeme im Vergleich

Die Tabelle zeigt, dass sich die meisten hier vorgestellten Systeme noch in einem frühen Stadium befinden. Jedes System, bis auf Aurora, befindet sich zur Zeit noch in der Entwicklung. Aurora hingegen, wurde bereits durch einen Nachfolger *Borealis* abgelöst, der Aurora um Neuerungen wie der *dynamischen Modifikation von Anfragen* (*dynamic query modification*) und der *skalaren Optimierung* (*scalable optimizations*) erweitert. Borealis befindet sich zur Zeit ebenfalls in der Entwicklung. Es bleibt also abzuwarten ob die Systeme ausserhalb der Forschung ihren Einsatz finden werden, und sich dort behaupten können. Es bleibt ebenfalls offen, ob sich ein System gegenüber den anderen durchsetzen wird. Die Wichtigkeit der Forschung sollte jedoch nicht unterschlagen werden: Mit Sicherheit werden Datenströme in Zukunft von zunehmender Bedeutung. Solide Konzepte und ausreichend Erfahrung mit vergangenen Systemen sind da von größter Bedeutung.

Literaturverzeichnis

[ABBC04] A. Arasu, B. Babcock, S. Babu. J. Cieslewicz, M. Datar, K. Ito, R. Motwani, U. Srivastava, J. Widom: STREAM: The Stanford Data Stream Management System, 2004

[ACCC02] D. Abadi, D. Carney, U. Cetintemel, M. Cherniackm 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, 2002

[ArBW03] A. Arasu, S. Babu, and J. Widom. The CQL Continuous Query Language: Semantic Foundations and Query Execution. Technical report, Stanford University, 2003.

[BBDM02] B. Babcock, S. Babu, M. Datar, R. Motwani, J. Widom: Models and Issues in Data Stream Systems, 2002

[BCDF05] F. Bry, F. Coskun, S. Durmaz, T. Furche, D. Olteanu, M. Spannagel: The XML Stream Query Processor SPEX, Institute for Informatics, University Of Munich, 2005

[BrFO04] F. Bry, T. Furche, D. Olteanu: Datenströme, 2004

[CCCC02] D. Carney, U. Centintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, S. Zdonik: Monitoring Setrams - A New Class of Data Management Applications, 2002

[CHKM03] M. Cammert, C. Heinz, J. Krämer, A. Markowetz, B. Seeger: PIPES: A Multi-Threaded Publish-Subscribe Architecture for Continuous Queries over Streaming Data Sources, 2003

[Clut03] D. Clutter: Implementation on Naive Bayes, Slides for MAIDS Presentation, 2003

[Haer04] T. Härder: Grundlagen Betrieblicher Informationssysteme, Vorlesungsskript Sommersemester, Technische Universität Kaiserslautern, 2004

[IWEF03] Data Mining - Practical Machine Learning Tools and Techniques with Java Implementations, Academic Press, 2000

[Kulk03] K. Kulkarni: Overview of SQL:2003, IBM Corporation, San Jose, 2003

[MTTW05] MAIDS group, NCSA and Dept. of CS, University of Illinois at Urbana-Champaign: Implementation on Natural Tilted Time Windows, 2005

[OlFB04] D. Olteanu, T. Furche, F. Bry: An Efficient Single-Pass Query Evaluator for XML Data Streams, 2004.

[Prok03] Joe Prokop: Scribe Notes on Monitoring Streams - A New Class of Data Management Applications. 2003 (Elektronisch verfügbar unter `http://www.cs.iit.edu/~yee/classes/cs595f03/103003_2.htm`)

[RaGG02] R. Rastogi, M. Garofalakis, J. Gehrke: Querying and Mining Data Streams: You Only Get One Look, Presentation for Bell Labs, 2002

[SeGh90] T. Sellis, S. Ghosh: On the Multiple-Query Optimization Problem, 1990

[W3C99] o.V.: XML Path Language (XPath) - W3C Recommendation (Elektronisch verfügbar unter: `http://www.w3.org/TR/xpath`)

[WikiDB] Wikipedia - The Free Encyclopedia: Definition Database management system. (Elektronisch verfügbar unter: `http://en.wikipedia.org/wiki/Database_management_system`)

[WikSQL] Wikipedia - The Free Encyclopedia: Defintion SQL. (Elektronisch verfügbar unter: `http://en.wikipedia.org/wiki/SQL`)

[WikiXP] Wikipedia - The Free Encyclopedia: Defintion XPath. (Elektronisch verfügbar unter: `http://de.wikipedia.org/wiki/XPath`)

[WiCC00] J. Widom, S. Ceri, R. Cochrane: Practical Applications of Triggers and Constraints: Successes and Lingering Issues, 2000

[XXLb2] XXL (the eXtensible and fleXible Library for data processing) beta 1.2 API Documentation, Automatically generated using JavaDoc. (Elektronisch verfügbar unter: `http://dbs.mathematik.uni-marburg.de/research/projects/xxl`)

[ZCSB03] S. Zdonik, U. Cetintemel, M. Stonebraker, M. Balazinska, M. Cherniack, H. Balakrishnan: The Aurora and Medusa Projects, 2003