# Seminar
# Data Streams

## Scheduling und Speicherverwaltung in Data Streams

von

Jörg von Blon

Betreuer: Jürgen Göres

**18. Mai. 2005**

# Inhaltsverzeichnis

# 1. Grundlagen der Datenstromverarbeitung

Die Verarbeitung von Datenströmen hat in den letzten Jahren im Bereich der Datenbankforschung verstärkte Aufmerksamkeit erfahren. Der effiziente Umgang mit potentiell unendlichen Strömen bringt neue Anforderungen an die Datenbankverarbeitung mit sich, was die Forschung in diesem Bereich besonders motivierte. Anwendungsszenarien solcher Systeme sind z.B. Verkehrsmanagement, Datenverarbeitung in Sensoren und Netzwerken, um nur einige zu nennen.
Im Gebiet der Sensortechnik sind Anfragen auf Datenströmen heute kaum noch wegzudenken. Ein in diesem Bereich wichtiges Einsatzgebiet sind Wetterstationen, welche viele Parameter wie Temperatur, Luftströme und Unwetterdaten sehr schnell und präzise verarbeiten müssen. Diese Daten sind für die Luft- und Raumfahrttechnik von enormer Bedeutung. Eine typische Anfrage wäre: „Liste die Wetterstationen auf, an denen in den letzten 15 Minuten eine durchschnittliche Windgeschwindigkeit von über 100km/h gemessen wurde“. Eine solche Anfrage wird kontinuierliche Anfrage genannt [13]. Die Verarbeitung dieser Datenströmen bedeutet eine neue Herausforderung und verlangt nach anderen Verarbeitungstechniken wie die aus traditionellen Datenbank-Managementsystemen (DBMS). Dort sind ausschließlich Anfragen über persistent in der Datenbank abgelegte Daten möglich, d.h. es erfolgt eine Auswertung über dem festen Datenbestand und es wird *eine* Antwort geliefert. Kontinuierliche Anfragen hingegen werden einmal definiert, im System abgelegt und fortlaufend über den Eingangsdatenströmen ausgewertet. Die Ergebnisse einer Anfrage können dann optional als Relation oder ebenfalls als Strom modelliert werden, je nachdem wie es die aktuelle Anwendung verlangt. Eine Möglichkeit solche kontinuierlichen Anfragen in einem DBMS zu simulieren ist die Verwendung von Triggern. Problem hierbei ist, dass diese in ihren Möglichkeiten sehr beschränkt sind. Es ist ihnen z.B. nicht möglich Antworten als Strom zu modellieren. Außerdem sind Triggern die Verwendung von Approximationstechniken vorenthalten und der Gebrauch von sehr vielen Triggern wirkt sich negativ auf die Ausgabezeit der Ergebnisse aus. Diese beiden komplementären Konzepte zwischen DBMS und DSMS sind in Abbildung 1 graphisch dargestellt.
Ein ankommender Datenstrom kann in einem DBMS als ein ständiges Einfügen von Daten in Relationen verstanden werden. Ein traditionelles DBMS ist jedoch nicht dafür ausgelegt Daten in einer hohen Geschwindigkeit zu laden. Deswegen spricht in diesem Zusammenhang von passiven Datenbeständen, welche sich durch eine lange Lebensdauer und wenige Änderungen auszeichnen. In einem DSMS hingegen wird mit aktiven Datenbeständen gearbeitet, d.h. man arbeitet auf einem Abschnitt eines Stroms, in welchem sich die Daten auf denen gearbeitet wird ständig ändern. Dies stellt einen fundamentalen Unterschied zum traditionellen Konzept der DBMS dar und erfordert neue Techniken der Verarbeitung. Die ankommenden Datenströme müssen in Echtzeit verarbeitet werden, da man nur einen begrenzten Speicher zur Verfügung hat. Problem hierbei ist die potentiell unbegrenzte Größe dieser Datenströme, insbesonders wenn im Anfragegraph zustandsbehaftete Operatoren existieren. Blockierende Operatoren, welche eine Teilmenge der zustandsbehafteten Operatoren darstellen, müssen alle ihre Eingaben komplett gesehen haben, bevor sie eine korrekte Ausgabe liefern. Ein typischer Vertreter dieser Gruppe ist der Join-Operator. Operatoren wie Projektion und Selektion zählen zu den zustandslosen Operatoren und weisen diese Problematik bei der Verarbeitung nicht auf.
In einem DBMS wird zur Beantwortung von Anfragen stets der aktuelle Zustand der Datenbank betrachtet. In einem DSMS hingegen wird auf die Daten sequentiell in der Ankunftsreihenfolge zugegriffen, welche unvorhersehbar ist und Auswirkungen auf die Qualität der Antwort haben kann.

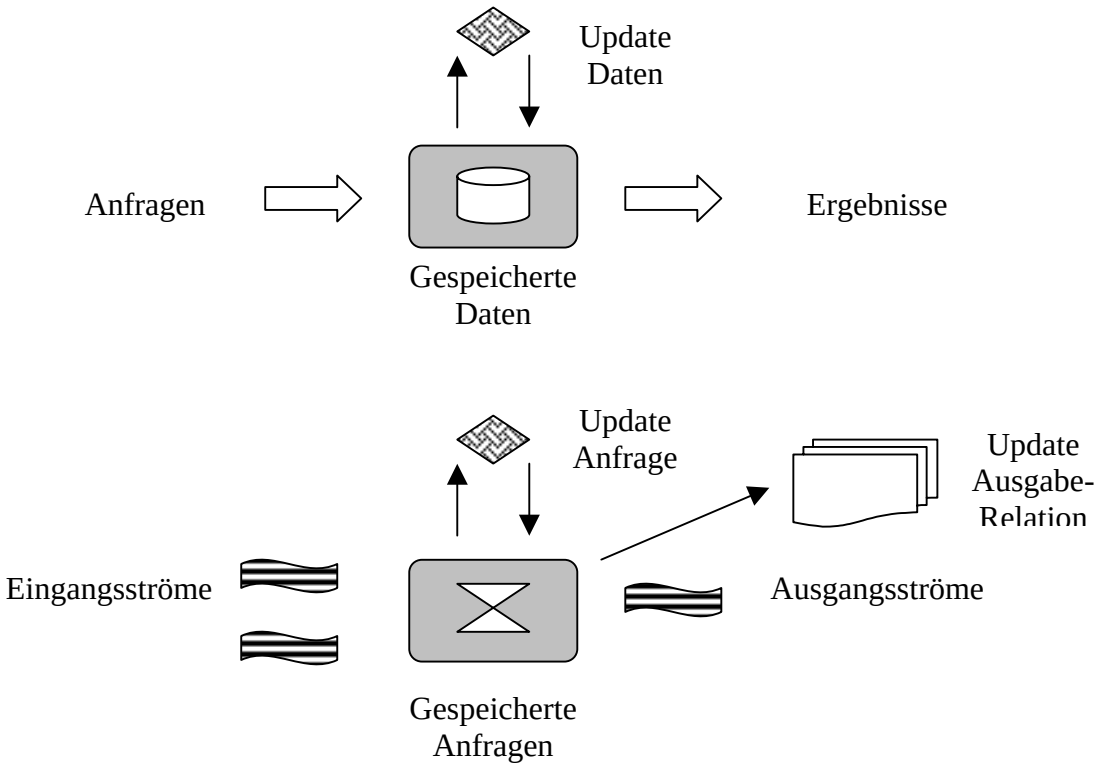


**Abbildung 1: Datenverarbeitung in DBMS (oben) und DSMS (unten)**

Da die Datenströme wie schon erwähnt potentiell unendliche Länge haben, sind Techniken erforderlich, die blockierende Operatoren so anpassen, dass sie auch ohne komplette Eingabe Ergebnisse liefern können. Hier kommen Zeitfenster zum Einsatz, die das blockierende Verhalten von Operatoren auflösen , indem nur noch ein begrenzter Ausschnitt des Datenstroms betrachtet wird. Die beiden bekanntesten Techniken der Fenstersemantik sind *gleitende* und *feste* Fenster. In gleitenden Fenstern erhält jedes Element eines Datenstroms die gleiche Gültigkeitsdauer. Dadurch ist zu jedem Zeitpunkt ein Teil jedes Datenstroms gültig. Dieses Gültigkeitsintervall verschiebt sich somit mit der Zeit über den gesamten Datenstrom. Bei festen Fenstern wird die Zeitachse in gleich lange Intervalle unterteilt. Liegt der Startzeitstempel innerhalb eines Intervalls, so wird dessen Endzeitstempel auf das Ende des Fensters gesetzt (Abb.2a & 2b).

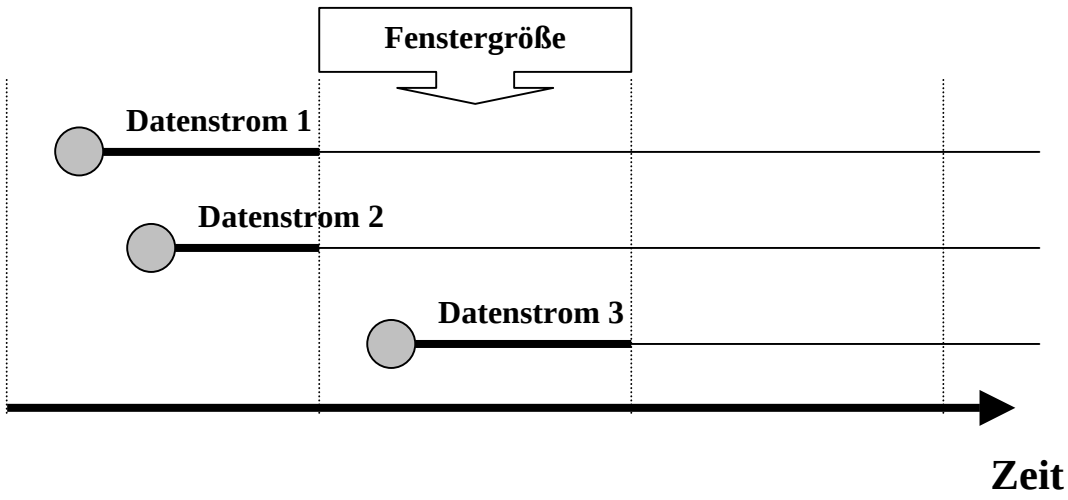

**Abbildung 2a: Festes Fenster (Fixed window)**

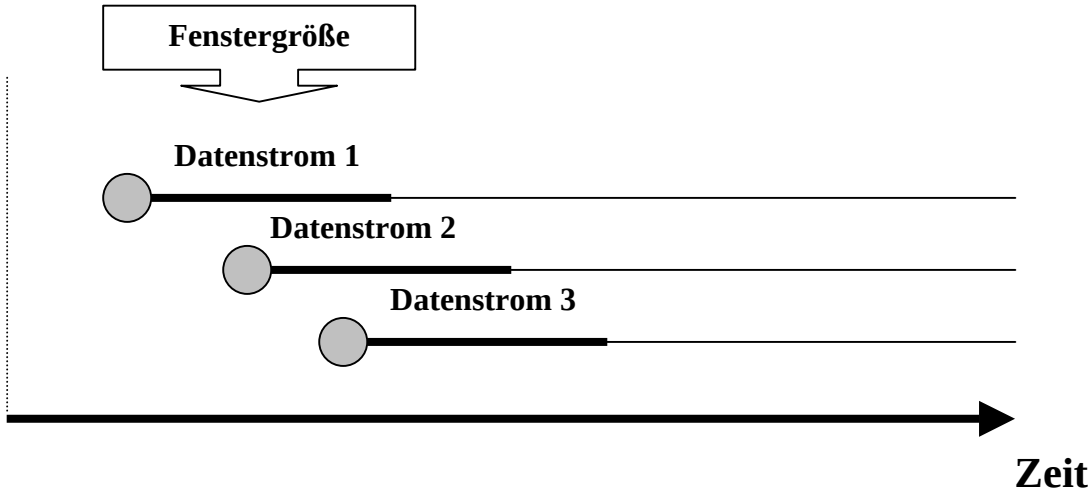


**Abbildung 2b: Gleitendes Fenster (Sliding window)**

Genau die Tupel; DIE ZUM AKTUELLEN Zeitpunkt gültig sind, werden nun zur Anfragebeantwortung herangezogen. Um nun den Speicherbedarf eines Anfragegraphen anzupassen, kann man die Größe der Fenster skalieren. Diese Funktion übernimmt in vielen Systemen die sogenannte Speicherverwaltung, die dies unter Berücksichtigung der Dienstgüte (Quality of Service) steuert. Darunter zählt z.B. die Einstellung von Latenzzeiten durch den Benutzer. Die optimale Zuteilung von Ressourcen wie CPU-Zeit, Haupt-, Externspeicher und Netz-Bandbreite gestaltet sich in der Praxis als äußerst schwierig. Aufgrund des begrenzten Speicherbedarfs lassen sich oft nur approximierte Antworten geben. Adaptivität und Skalierbarkeit spielen in DSMS somit eine entscheidende Rolle, da die Datenrate zur Laufzeit stark variiert. Einige Techniken und Möglichkeiten werden in den folgenden Kapiteln näher erläutert und bewertet. Es werden in Kapitel 2 auch existierende Systeme vorgestellt, in welchen besondere Strategien erfolgreich verwendet werden. Allgemein kann man jedoch fast immer sagen, dass mit sinkendem Speicherbedarf auch die Qualität der approximierten Beantwortung der Anfrage sinkt [1].
Eine weitere Herausforderung ist das Stellen von Ad-hoc-Anfragen, die sich auf sogenannte *historische* Daten beziehen. Dieses sind Anfragen, welche erst gestellt werden, wenn das System schon in Betrieb ist und über längere Zeit Daten verarbeitet hat. Um solche Anfragen möglichst präzise beantworten zu können muss das System auch noch auf historische Daten zurückgreifen können. Das Maß inwieweit vergangene Daten noch betrachtet werden, kann durch bestimmte Techniken definiert werden. Eine Möglichkeit wäre z.B. die Verwendung von Zeitfenstern über den Datenströmen. Ein solches Fenster könnte z.B. die Daten der letzten zehn Minuten beinhalten, oder aber auch die Sicherung der letzten 100 Tupel ist eine durchaus denkbare Implementierung.
Der zweite wichtige Eckpfeiler dieser Ausarbeitung ist neben der Speicherverwaltung das Scheduling.
Die Zuteilung der Rechenzeit in einem DBMS geschieht über den sogenannten Scheduler. Dieser überwacht alle im System ablaufenden Anfragen und muss anhand deren gewünschter Dienstgüte (Quality of Service) und Dringlichkeit entscheiden, welche davon priorisiert wird und Rechenzeit erhält. Aufgrund der oben genannten Gründe bezüglich der Problematik mit Datenströmen wird schnell klar, dass der Umgang mit Ressourcen von zentraler Bedeutung ist. Besonders in Zeiten mit hohem Datenaufkommen ist ein geschicktes Handling mit CPU- und Speicherverwaltung von enormer Bedeutung. Es existieren grundsätzlich zwei Aspekte, nach denen man eine solche Verwaltung ausrichten kann. Zum einen könnte man eine maxi-

mal große Lastreduktion anstreben, oder zum anderen einen maximalen Erhalt an Qualität. Betrachtet man beispielsweise einen Operatorgraphen mit der sequentiellen Verkettung der Operatoren Op1,Op2 und Op3. Verwirft man nun 95% aller Tupel, die an Op1 ankommen, so ist klar, dass weniger Tupel im Operatorgraphen weitergereicht werden wie zuvor und die Last enorm reduziert wurde. Dies bedeutet, dass der Speicherbedarf, sowie die Menge an benötigter CPU-Zeit verringert wurde. Aufgrund der hohen Verlustrate an Tupeln kann es jedoch vorkommen, dass die geforderte Dienstgüte nicht erreicht werden kann.
Die naheliegensten Ansätze, um mit Datenströmen in Bezug auf Anfragen umzugehen, sind operatorbasiertes Scheduling(OTS) und graphbasiertes Scheduling(GTS). Jede der Techniken bietet gewisse Vor- und Nachteile, welche abhängig vom Aufbau und Auslastung des Systems sind. Beim OTS (operator-based-scheduling) bekommt jeder Operator seinen eigenen Thread. Für nicht zu große Queues bewirkt diese Technik sehr schnelle Antwortzeiten. Durch die parallele Verarbeitung der Tupel werden diese schnell durch das System geführt. Man beachte jedoch, dass mit jedem zusätzlichen Thread auch die Kosten an Speicher und Prozessorzeit ansteigen. Durch diese erhöhten Verwaltungskosten vergrößert sich bei einem hohen Datenaufkommen (Bursts) auch die Antwortzeit auf Anfragen, da CPU-Zeit und Speicherverfügbarkeit knapp werden.
Die komplementäre Möglichkeit lautet GTS (graph-threaded-scheduling). Dabei wird der komplette Anfragegraph von einem Thread bearbeitet. Innerhalb dieses Threads wird von einem sogenannten Selector der nächste Operator, welcher zur Ausführung kommt, bestimmt. Sobald dieser Operator mit der Bearbeitung fertig ist, wählt der Selector den nächsten Operator aus, der mit der Bearbeitung von Tupeln fortfährt. Problematisch wird es, wenn ein Operator aus irgendwelchen Gründen sehr viel Zeit zur Bearbeitung von Tupeln benötigt. Die weiter ankommenden Tupel müssen dann gespeichert werden und eventuell bei konstant hohem Datenaufkommen aufgrund Speichermangels verworfen werden. Außerdem sinkt die Ausgaberate gegen Null.
Es werden im folgenden verschiedene Systeme und deren Verfahren zur Speicher- und CPU-Verwaltung vorgestellt, bewertet und diskutiert. Besondere Bedeutung wird dem Operator Scheduling beigemessen und dessen Auswirkung auf unsere Ressourcen wie Speicher und CPU-Zeit. Diese Technik wird besonders effizient bei mehreren verschiedenen Eingangsströmen mit variablen Datenaufkommen verwendet [9].

# 2. Scheduling und Speicherverwaltung in Streams

Im ersten Kapitel wurde die Problematik zwischen Scheduling und der Speicherverwaltung bereits angesprochen. Ist die Rate eintreffender Daten sehr hoch, so muss man einen bestmöglichen Kompromiss zwischen CPU-, Speicherverwaltung und der erreichten Genauigkeit der Ausgabe schließen. Eine 100% korrekte Ausgabe ist in Zeiten von hohen und schwankenden Datenaufkommen unter Umständen nicht mehr möglich. Unser Hauptziel wird es nun sein, mit bestimmten Scheduling-Strategien die gegebenen Ressourcen möglichst optimal einzusetzen, um ein Maximum an Tupeln zu bearbeiten und trotzdem in den Grenzen der definierten Dienstgüte zu bleiben. So lange die mittlere Bearbeitungszeit eines Tupels kleiner der Ankunftsrate neuer Tupel ist, benötigt man weder Techniken zur Lastreduktion, noch andere Approximationstechniken. Ist dies jedoch nicht der Fall, so entsteht eine Überlast-Situation in welcher sichergestellt werden muss, dass die Queue-Größe nicht unseren verfügbaren Speicher überschreitet.
Jede Anfrage wird dargestellt durch einen gerichteten azyklischen Graphen. Aufgrund der potentiell großen Zahl paralleler Anfragen innerhalb des Systems weisen oft mehrere Anfragen gemeinsame Teilausdrücke auf. Durch Erkennen solcher Überlappungen kann eine Mehrfachausführung vermieden werden und somit eine ressourcenschonendere Beantwortung der

Anfragen stattfinden. Diese Technik machen sich auch einige Systeme zu Nutzen, was im späteren Verlauf dieses Kapitels näher beleuchtet wird.
Betrachten wir uns zur Einführung dieser Problematik ein leicht verständliches Beispiel, an welchem wir sehen, dass die richtige Auswahl einer geeigneten Strategie große Auswirkungen auf den Speicherbedarf haben kann.
Gehen wir von einem Operatorpfad mit zwei Operatoren aus, Op1 gefolgt von Op2. Op1 hat eine durchschnittliche Selektivität von 20% (Selektivität = durchschnittliche Prozent-Anzahl an Tupeln, welche vom Operator zur Verarbeitung weitergeleitet wird). Bei einem Input von n Tupeln hat Op1 somit einen Output von 0,2n. Dies bedeutet eine Reduktion der Tupelanzahl um 80%, was absolut 0,8n ausmacht. Eine solche Verarbeitung schafft Op1 in einer Zeiteinheit. Op2 kann in einer Zeiteinheit 0,2n Tupel verarbeiten und leitet das Ergebnis der Anfrage an das System weiter. Somit findet an diesem Operator eine absolute Tupelreduktion von 0,2n statt.
Anhand dieses Szenarios werden nun zwei Scheduling-Strategien auf Speicherverbrauch miteinander verglichen.

Strategie 1: FIFO
Es werden die Tupel priorisiert, welche die längste Zeit im System vorhanden sind.

Strategie 2: Greedy
Greedy wählt immer den Operator aus, der eine maximale Verminderung der Tupelanzahl zur Folge hat.

Zur Vereinfachung dieses Beispiels nehmen wir an, dass immer Bündel von n Tupeln pro Zeiteinheit verwendet werden.
In den Zeiteinheiten eins bis sieben kommen jeweils Bündel von n Tupel vor Op1 an. Die Vorgehensweise der beiden Strategien ist in Abbildung 3 graphisch dargestellt.

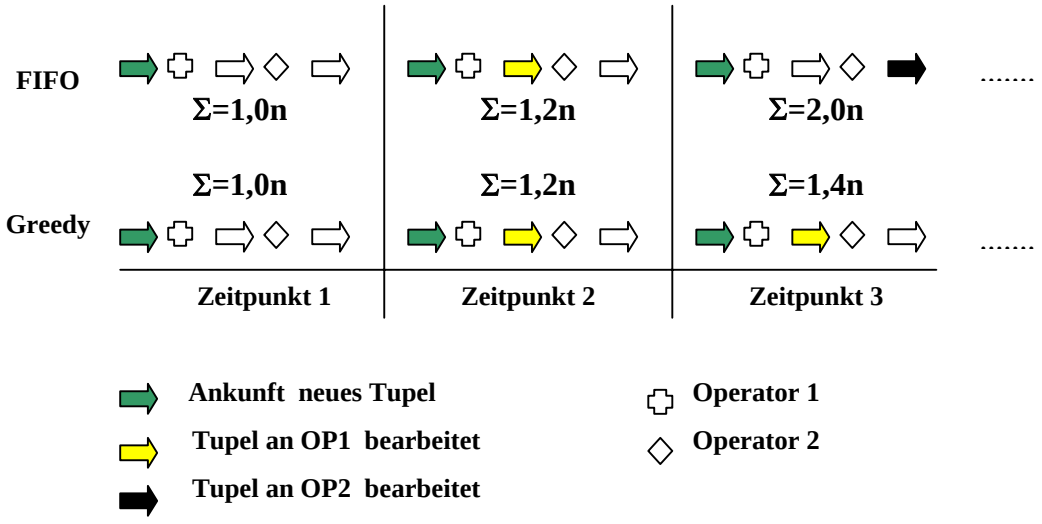


**Abbildung 3: Ablauf der beiden Strategien FIFO und Greedy**

Dies bedeutet für Greedy, dass immer Op1 ausgewählt wird, da dieser eine absolute Reduktion von 0,8n schafft. Die Summe an Tupeln wächst somit pro Zeiteinheit um 0,2n. Erst nach der siebten Zeiteinheit wird damit angefangen die Queue vor Op2 zu leeren.
Die Tabelle zeigt die Summe an Tupeln, welche vor den Eingangswarteschlangen der beiden Operatoren warten.

| ZEITEINHEIT | Greedy-Strategie | | | FIFO-Strategie | | |
|---|---|---|---|---|---|---|
| | Tupel vor Op1 | Tupel vor Op2 | ∑ | Tupel vor Op1 | Tupel vor Op2 | ∑ |
| 1 | 1,0n | 0,0n | 1,0n | 1,0n | 0,0n | 1,0n |
| 2 | 1,0n | 0,2n | 1,2n | 1,0n | 0,2n | 1,2n |
| 3 | 1,0n | 0,4n | 1,4n | 2,0n | 0,0n | 2,0n |
| 4 | 1,0n | 0,6n | 1,6n | 2,0n | 0,2n | 2,2n |
| 5 | 1,0n | 0,8n | 1,8n | 3,0n | 0,0n | 3,0n |
| 6 | 1,0n | 1,0n | 2,0n | 3,0n | 0,2n | 3,2n |
| 7 | 1,0n | 1,2n | 2,2n | 4,0n | 0,0n | 4,0n |

**Tabelle 1: Summe der Eingangswarteschlangen nach 7 Zeiteinheiten**

Was man aus diesem Experiment erkennen soll ist, dass man den Speicherverbrauch dadurch minimiert, indem man die Operatoren zur Ausführung bringt, welche die maximale absolute Anzahl an Tupelverkleinerung bewirkt. Was man jedoch bei der Greedy-Strategie noch erwähnen sollte ist die sehr hohe Latenzzeit der Tupel. Würde in diesem Szenario ständig Daten ankommen, so würde der Algorithmus aufgrund der Pufferung niemals eine Antwort liefern.
Das Verfahren zur Verkleinerung der absoluten Tupelanzahl wird der nun folgende Chain-Algorithmus versuchen auf komplexe Anfragegraphen zu übertragen.

## 2.1 Vorstellung der STREAM-Architektur

STREAM (Stanford stREam datA Manager) [6] ist ein DSMS, welches für den Umgang mit fluktuierenden Datenströmen entworfen worden ist. Ein besonderes Merkmal des STREAM-Konzepts ist die interaktive Benutzerschnittstelle, welche die Anfrageverarbeitung in Echtzeit visualisiert. Dies ist besonders hilfreich für das Experimentieren mit verschiedenen Datenverarbeitungstechniken, da Ergebnisse direkt dargestellt werden und man adäquat darauf reagieren kann. Datenströme werden in diesem Modell als eine unbegrenzte Menge von Paaren aus je einem Tupel und einem Zeitstempel dargestellt. Relationen sind variierende Mengen an Tupeln, welche Einfüge-, Update- und Lösch-Operationen unterstützen. Anfragen können über die deklarative Anfragesprache CQL im System registriert werden.
Datenströme werden mit Hilfe von Fensteroperatoren in Relationen umgewandelt. Die Verarbeitung von Relationen untereinander erfolgt mit den traditionellen Operatoren eines DBMS. Nach der Verarbeitung von Relationen können Ergebnisse wieder als Strom ausgegeben werden.
Nach der Registrierung von CQL-Anfragen in STREAM werden diese in Anfragepläne umgewandelt. Beim Einfügen neuer Anfragen wird in den schon existierenden Anfragepläne nach Überlappungen gesucht, um Bearbeitungszeit einzusparen. Anfragepläne bestehen in STREAM aus drei verschiedenen Komponenten: Anfrageoperatoren, Zwischen-Warteschlangen und Synopsen.
Das Zusammenspiel der verschiedenen Komponenten wird durch den Anfrageplan in Abbildung 4 dargestellt.

Anfrageoperatoren:
Diese verarbeiten Tupel ihrer Eingangswartschlange und leiten die Ausgabe an Zwischen-Warteschlangen anderer Operatoren weiter (In Abb.4 wird ein Join-Operator $O_1$ auf zwei Datenströme R und S ausgeführt).

Zwischen-Warteschlangen:
Ihre Aufgabe besteht darin die Ausgabe eines Operators zu puffern und an einen oder mehrere Operatoren weiterzugeben (In Abb. 4 puffert $q_3$ Ausgabe von $O_1$ und leitet diese an $O_2$ und $O_3$ weiter).

Synopsen:
Diese Strukturen stellen die Tupel zur Verfügung, die Operatoren im Anfrageplan zur Beantwortung der Anfrage benötigen (z.B. Schiebefenster-Synopse). Auch das Zusammenfassen von Tupeln für eine approximierte Beantwortung zählt zu den Synopsen (In Abb.4 werden zwei Fenster $S_1$ und $S_2$ zur Bearbeitung des Join-Operators $O_1$ benötigt).

Das Finden eines speicheroptimalen Anfrageplans ist ein NP-vollständiges Problem [14]. Ein Algorithmus, der dieses Problem in polynomialer Zeit löst müsste jedoch Wissen über zukünftig im System ankommende Daten besitzen. Die Forschungsgruppe der Stanford-Universität entwickelte für das STREAM-System ein Scheduling-Verfahren, welches im Bezug auf minimalen Speicherverbrauch sehr nahe an der optimalen Lösung liegt – den Chain-Algorithmus.

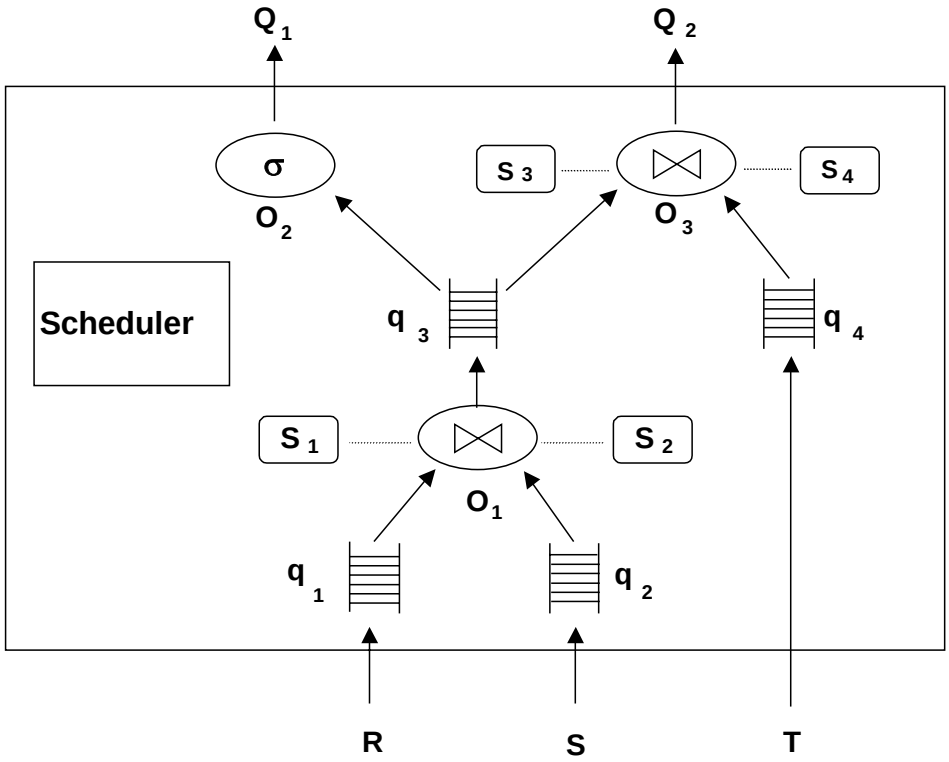


**Abbildung 4: Anfrageplan in STREAM**

## 2.1.1 Chain Scheduling

Unsere Forderungen an gute Scheduling-Strategien sind zum einen die Garantie auf schnelle Antwortzeiten und somit einer niedrigen Latenzzeit und zum anderen eine effiziente Ausführung des Algorithmus, da dieser sehr häufig ausgeführt wird. Dies stellt eine besondere Herausforderung dar, wenn die Datenrate variabel ist und plötzlich sehr hohe Datenraten auftreten. Treten solche Daten-Bursts nur kurz auf, so werden die ankommenden Tupel gepuffert und in Phasen mit geringer Auslastung verarbeitet. Für das System ist es von enormer Wichtigkeit den Speicherbedarf dieser Datenpufferung so gering wie möglich zu halten, um den verfügbaren Speicher nicht zu erschöpfen.
Der Chain-Algorithmus verhält sich annährend optimal bei der Speicherminimierung während der Laufzeit und verbraucht selbst wenige Ressourcen [11]. Grundlage des Scheduling-Algorithmus sind die sogenannten Progress.

## 2.1.2 Progress Charts

In einem Progress Chart wird die Verarbeitung von Tupeln in einem Anfragegraphen festgehalten. Operatoren werden in diesem Zusammenhang wie Filter mit einer gewissen Selektivität betrachtet. In dieser Darstellung wird jedoch die bedingte Selektivität betrachtet, d.h. die Selektivität eines Operators, nachdem das Tupel schon mehrere Operatoren vorher durchlaufen hat. Tupel sind in der von uns gewählten Sichtweise nichts anderes als Speichereinheiten, d.h. ein Operator mit einem Tupel Input, welcher die Selektivität s besitzt, benötigt im Mittel s Speichereinheiten für den Output. Anhand der folgenden Progress Charts wird deren Erstellung im Einzelnen erläutert:

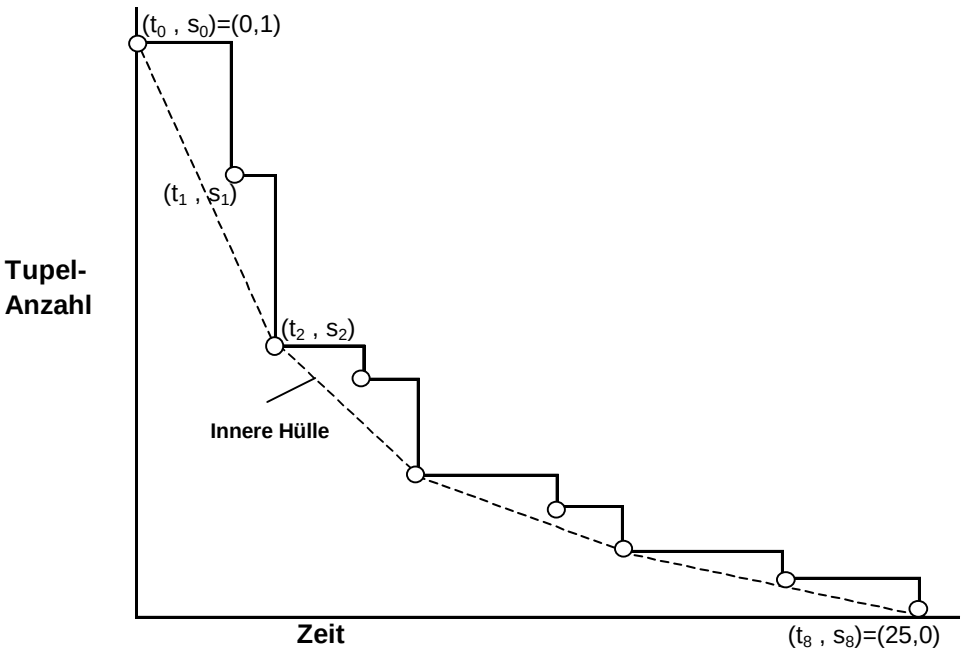


**Abbildung 5: Progress Chart : Alle Operatoren besitzen eine Selektivität < 1**

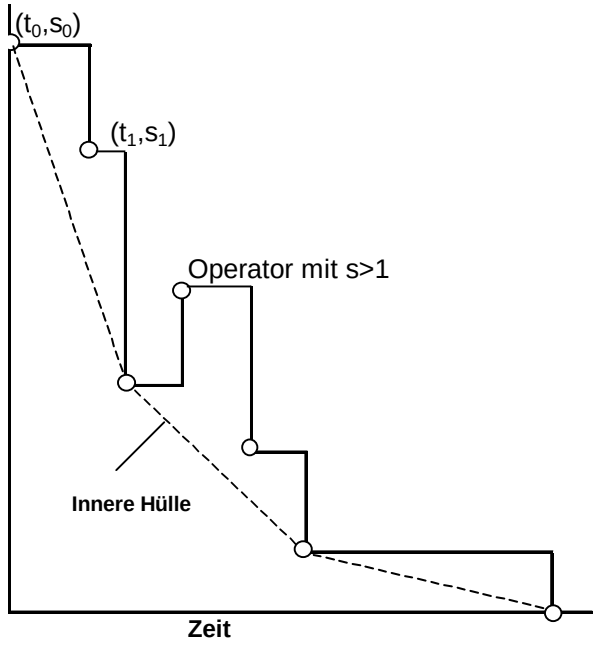


**Abbildung 6: Progress Chart : Ein Operator (z.B. Join) besitzt eine Selektivität > 1**

Wie an der Achsenbeschriftung deutlich wird, ist horizontal die Zeit und senkrecht den Anteil der noch verbleibenden Tupel relativ zur Eingabemenge eingetragen. Dieser Anteil wird auch als konditionale Selektivität bezeichnet. Bei n Operatoren werden n+1 Operator-Punkte eingetragen($t_0$,$s_0$),.....($t_n$,$s_n$), wobei ($t_0$,$s_0$) immer für (0,1) steht, da zu Beginn noch keine Verarbeitung von Tupeln stattgefunden hat. Operator i benötigt also folglich $t_i$-$t_{i-1}$ Zeiteinheiten, um eine Tupelgröße von $s_{i-1}$ zu verarbeiten. Die Selektivität eines Operators i berechnet sich mittels $s_i/s_{i-1}$ [11].
Um den Progress Chart aufzustellen geht man vom aktuellen Operatorpunkt waagrecht nach rechts, bis man den t-Wert des nächsten Punktes erreicht hat. Bei Operatoren mit Selektivitäten < 1 findet man den nächsten Operatorpunkt unterhalb des aktuellen. Join-Operatoren können Selektivitäten > 1 aufweisen, was dazu führt, dass man aufgrund der erhöhten Tupelanzahl den nächsten Operatorpunkt nach oben verbinden muss.
Tupel, welche den Anfragegraph durchlaufen, wandern im Mittel an dieser Progress-Linie entlang. Die Interpretation der Grafik in Abbildung 5 und 6 ist also, dass die waagrechten Linien die Ausführungszeit des jeweiligen Operators und die senkrechten Linien das Verwerfen von Tupelteilen entsprechend seiner Selektivität darstellen. Dadurch dass das System am Ende des Progress Chart die Tupel nicht mehr Puffern muss, da die Tupel das System verlassen, hat der letzte Operator die Selektivität 0. Bei diesem Konzept gehen wir jedoch davon aus, dass die Selektivität und Bearbeitungszeit pro Tupel schon bekannt sind. Ist dies nicht der Fall, so kann man den Zeitstreifen in Segmente unterteilen und für jedes Segment unabhängig voneinander statistische Werte erfassen. Mit diesen Statistiken kann man nun aus dem aktuell betrachteten Segment das nachfolgende berechnen. Um nun Chain Scheduling anwenden zu können ermitteln wir aus den Progress Chart die sogenannte innere Hülle. Um diese zu berechnen starten wir am ersten Operatorpunkt und ermitteln von dort an die betragsmäßig größte Steigung zu den anderen Punkten. Diese beiden Operatorpunkte werden miteinander verbunden und das Verfahren bei dem neuen Operatorpunkt wiederholt, bis man den letzten Operatorpunkt erreicht hat. Dadurch wird der Graph in Segmente unterschiedlicher Steigung unterteilt. Die inneren Hüllen sind in den beiden Skizzen in Abbildung 6 als gestrichelte Linie eingezeichnet.

## 2.1.3 Chain Scheduling mit einem Datenstrom

In einem DSMS sind zu einem Zeitpunkt in der Regel mehrere Anfragen aktiv, repräsentiert durch ihre Anfragegraphen. Diese haben wiederum unterschiedliche Progress Charts mit den dazugehörigen inneren Hüllen. Das Konzept des Chain Scheduling lautet nun wie folgt:
*Betrachte zu jeder Zeit alle Tupel, welche sich im System befinden. Betrachte alle Operatoren mit Tupeln in ihrer Queue und führe die Operatoren mit der betragsmäßig größten Steigung auf der inneren Hülle aus. Wenn sich dafür mehrere Tupel qualifizieren, dann wähle das Tupel, welches die früheste Ankunftszeit aufweist.*
Diese Vorgehensweise führt dazu, dass Anfrageoperatoren, die in gleichen Segmenten in der inneren Hülle liegen, zu Ketten zusammengefasst werden. Scheduling von verschiedener Ketten wird mit Hilfe des Greedy-Algorithmus vollzogen. Innerhalb einer jeden Kette werden die Operatoren jedoch bezüglich der FIFO-Strategie zur Ausführung gebracht.

## 2.1.4 Chain Scheduling auf mehreren Datenströmen

Im kommenden Abschnitt wird die Problematik im Umgang mit mehreren Datenströmen bei tupelbasierten Joins mit Chain Scheduling präzisiert. Tupelbasiert bedeutet in diesem Kontext, dass bei einem Join mit zwei Datenströmen R und S kein Tupel aus R mit Zeitstempel t verarbeitet wird, bevor nicht alle Tupel aus S mit Zeitstempel < t verarbeitet worden sind.
Eine Anfrage mit mehreren Datenströmen kann als ein gerichteter azyklischer Graph (DAG) angesehen werden. Datenströme entsprechen den Blättern des Graphen. Für jeden eingehenden Datenstrom wird nun der Anfragegraph aufgespaltet in sogenannte Operatorketten, welche den Ketten aus Abschnitt 2.1.3 entsprechen.

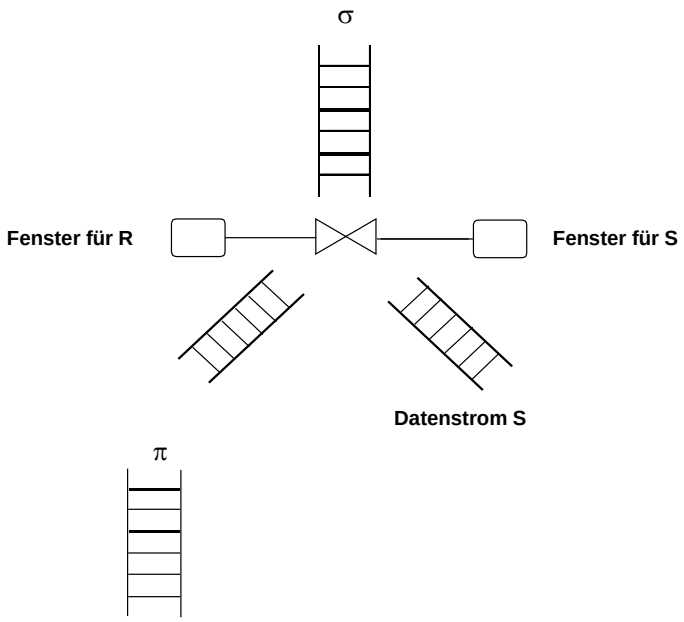


**Abbildung 7: JOIN über zwei Datenströme**

Die Anfrage in Abbildung 7 enthält einen Sliding-Window-Join zwischen zwei ankommenden Datenströmen R und S, wobei für jeden Datenstrom eine eigene Fenstergröße gewählt werden kann. Eine Aufspaltung würde in diesem Beispiel zu den beiden Ketten R>π>⋈>σ und S>⋈>σ führen, wobei ⋈>σ in beiden Operatorketten enthalten ist.
Diese Ketten können nicht ohne weiteres auf das vorgestellte Progress-Chart-Modell angewendet werden. Dies liegt an dem hier enthaltenen Join. Da R und S sich gegenseitig beeinflussen, müssen die Selektivitäten und die Tupelbearbeitungszeitin des Joins einer Operatorkette in Abhängigkeit des jeweils anderen Stroms und dessen Schiebefensters berechnet werden.
Die Formeln 1 und 2 beschreiben das Vorgehen zur Bestimmung der Selektivität und der Tupel-Bearbeitungszeit für Join-Operatoren, wobei $\lambda_i$ die durchschnittliche Anzahl an Tupel in einem Strom i pro Zeiteinheit und $t_i$ die durchschnittliche Bearbeitungszeit für ein Tupel des Stroms i ist. $\alpha_w(S)$ stellt die durchschnittliche Anzahl an Tupel des Fensters von Strom S dar, die mit Tupeln aus R einen Join eingehen.

$$X = \frac{\lambda_R * \alpha_w(S) + \lambda_S * \alpha_w(S)}{\lambda_R + \lambda_S}$$

Selektivität

**Formel 1**

$$Y = \frac{\lambda_R * t_R + \lambda_S * t_S}{\lambda_R + \lambda_S}$$

Tupel-Bearbeitungszeit

**Formel 2**

Nachdem man nun die benötigten Werte bestimmt hat, werden die Progress Charts wie beim Verfahren für einzelne Ströme aufgestellt und die beschriebene Chain-Scheduling-Strategie durchgeführt.
Es muss jedoch darauf geachtet werden, dass die Blockade bei Ketten mit Joins durchgeführt wird, bei denen eine Warteschlange keine Tupel mehr enthält. Selbst wenn eine Kette noch Tupel in der Eingangswarteschlange besitzt muss man immer die korrespondierende Warteschlange der anderen am Join beteiligten Kette betrachten, ob diese eventuell leer ist. Der Operator wird dann beim Scheduling so lange ignoriert, bis wieder neue Daten an der bisher leeren Queue ankommen.

## 2.1.5 Modifikationen des Algorithmus

Die Grundvariante des Chain-Scheduling-Algorithmus befasste sich ausschließlich mit der Minimierung des Speicherbedarfs. Es gibt jedoch auch Modifikationen, welche sich mit der Minimierung der Antwortzeit beschäftigen. In der Grundvariante kommt es nämlich sehr schnell zum Aushungern von Teilgraphen, da man ja primär die Steilheit der Segmente betrachtet und danach erst die Ankunftszeit. Dadurch werden neuere Tupel vor Operatoren mit hoher Selektivität(also in einem Segment mit stark negativer Steigung) älteren Tupel vor Operatoren mit niedriger Selektivität vorgezogen. Dies führt zu hohen Latenzzeiten, was insbesondere für zeitkritische Anwendungen zu unakzeptablen Ergebnissen führt. Es wird nun eine solche Modifikation vorgestellt, die ein Überschreiten von vom Benutzer festgelegten Obergrenzen für die Antwortszeit vermeidet und den Speicherverbrauch trotzdem so gering wie möglich hält. Es ist klar, dass bei extrem hohen Datenaufkommen eine zu 100% korrekte Antwort selbst bei einer annähernd optimalen Strategie nicht realistisch ist, wenn die Antwor-

ten in Echtzeit wiederzugeben sind. Mögliche Lösungen hierzu wären eine Lockerung der Bedingungen für Latenzzeit, oder das Verwerfen ausgewählter Daten, um näher an die vom Benutzer definierten Werte für Dienstgüte zu kommen. Solche Strategien werden später noch weiter ausgeführt.
Der Chain-Flush-Algorithmus verfährt zunächst genau wie der eingeführte Chain-Algorithmus, bis Forderungen an Deadlines eine Änderung der Strategie erzwingen. Wir gehen von n Tupeln aus, die aufsteigend nach ihrer Ankunft im System geordnet sind, wobei $r_i$ für die Deadline und $t_i$ für die Bearbeitungszeit des i-ten Tupels steht. Somit ist $t_1$ das älteste unbearbeitete Tupel.
Betrachten wir die nun folgende Bedingung:

$$\sum_{J=0}^{i} t_j = r_i \qquad \sum_{J=0}^{i'} t_j < r_{i'}$$

Für ein 0<i<=n  Für alle 0<i´<i

**Formel 3**  **Formel 4**

Formel 3 beschreibt die Situation, dass die Summe der Bearbeitungszeiten der ersten i Tupel genau ausreicht, um die Deadline des i-ten Tupels nicht zu verletzen.
Formel 4 hingegen fordert nun, dass alle Tupel mit Index kleiner i, ihre eigene Deadline nicht verletzen. Diese Bedingung ist aufgrund der Formel 3 nicht trivialerweise erfüllt. Die Ordnung wurde hergestellt aufgrund der Ankunftszeit im System und nicht mit Hilfe der Größe der Deadlines. Dies bedeutet, dass r2 > r4, aber r4 < r3 durchaus im Bereich des möglichen liegt.
Sind diese beiden Bedingungen von Formel 3 und 4 erfüllt, so steht das i-te Tupel davor seine Deadline zu verpassen. Ab diesem Zeitpunkt ist es dem System untersagt neu ankommende Tupel zu beachten, da diese ja eventuell durch höhere Priorisierung aufgrund der Selektivität sich vor das i-te Tupel schieben könnten. Erst wenn das i-te Tupel und natürlich auch alle vorherigen Tupel verarbeitet worden sind läuft auch die Verarbeitung neuer Tupel wieder an. Damit soll sichergestellt werden, dass die Tupel, die am längsten im System vorhanden sind und knappe Forderungen in Form von Deadlines besitzen, das System auch so schnell wie möglich innerhalb ihrer Deadline verlassen. Erst danach geht es mit dem Scheduling der übrigen Tupel weiter.
Problematisch an diesem Algorithmus ist, dass dieser jetzt tupelbasiert ist, da so jedes Tupel einzeln untersucht werden muss. Damit stellt dieser Lösungsweg ein großes CPU-Verwaltungsproblem bei Systemen mit vielen kontinuierlichen Datenströmen dar, was eine direkte Implementierung ausschließt.
Bei einer Vielzahl an Systemen sind die Anforderungen an die Antwortzeit jedoch nicht so stark. Kurze Überschreitungen dieser Schwellen führen somit nicht zu fatalen Folgen und werden toleriert. Eine solche Technik, welche sich der Approximation bedient ist der erweiterte Chain-Flush-Algorithmus. Es wird bei jeder Queue des Systems ein 4-Tupel betrachtet mit folgender Semantik:

$q$ = Eingangswarteschlange
$t_h$ = Zeitstempel des ersten Tupels
$r_q$ = Latenzbedingung des Tupels
$p_q$ = durchschnittliche Bearbeitungszeit eines Tupels vom ersten Operator bis zum Letzten im Anfragegraphen

Der Scheduler arbeitet nun mit folgender Bedingung: Aktuelle Zeit + $p_q$ >= $t_h$ + $r_q$

Ist diese Gleichung nicht erfüllt, so wird der „normale“ Chain-Algorithmus durchgeführt und die höchste Priorität beachtet.
Ist diese Gleichung allerdings erfüllt, so wird in den Flush-Modus gewechselt, um das erste Tupel durch den kompletten Anfrageplan zu schleusen. Es wird also in gewisser Weise ein imaginäres Segment geschaffen vom lesenden, aktuellen bis zum letzten Operator im Anfrageplan.
Offensichtlich können bei dieser Implementierung des Algorithmus präzise Antwortzeiten nicht garantiert werden, jedoch ist die Annährung an den geforderten Schwellenbereich in der Praxis recht gut.
Eine weitere einfache Modifikation des Chain-Algorithmus ist der Mixed-Algorithmus. Bei diesem werden flache Stücke zum Ende der inneren Hülle miteinander zu einem gemeinsamen Segment verbunden. Ausgangspunkt der Entscheidung welche Segmente zusammengefügt werden ist der vom Benutzer festgelegte Schwellenwert (slope threshold) für die Steigung. Dieses neue Segment weist natürlich eine größere Steigung als das flachste Segment innerhalb dieser Zusammensetzung. Innerhalb des Progress Chart wird dieses neue Segment mit Hilfe der Greedy-Strategie ausgeführt. Danach wird auf die zusammengesetzten Teilstücke innerhalb des neuen Segments die FIFO-Strategie angewendet. Durch diese Technik wird auf einfache Weise das Aushungern von flachen Segmenten in den Progress Charts verhindert, da flache Teilstücke zusammengefasst werden zu einem ausgeglichenerem Segment und dadurch das neue Segment stärker priorisiert wird wie die flachsten Segmente dieser Verbindung.

## 2.1.6 Bewertung und Vergleich mit anderen Strategien

In den folgenden Tabellen ist ein Vergleich aufgestellt worden zwischen den Strategien Chain, FIFO, Mixed. In Tabelle 2 und 3 wird von einem Progress Chart mit den folgenden Punkten für drei Operatoren mit Selektivität < 1 (z.B. Projektion, oder Selektion) ausgegangen: (0;1),(1;0,1),(99;0,001),(100;0). Dies bedeutet, dass es drei Segmente im Anfragegraph gibt, wobei das letzte ein sehr flaches Teilstück ist. Für die komplette Bearbeitung eines Tupels durch alle Operatoren dauert es 100 Zeiteinheiten.
Als Strategie verwenden wir den Mixed-Algorithmus mit dem Wert 0,01. Mixed(0,01) verbindet also die beiden untersten Segmente und wendet darauf den Chain-Algorithmus an.
Bei dem Ergebnis des Experiments in Tabelle 2 sind 100 Tupel im System angekommen, wobei das i-te Tupel zum Zeitpunkt 99*i angekommen ist. Dies bedeutet, dass bevor das Tupel vom letzten Operator (sehr flaches Segment im Anfragegraphen) verarbeitet wird, kommt ein neues Tupel in das System, welches priorisiert behandelt wird (wegen steilerem Segment).
Bei dem Ergebnis des Experiments in Tabelle 3 kommen nur 10 Tupel an, jedoch in einer höheren Ankunftsrate. Dort trifft das i-te Tupel zum Zeitpunkt 10*i ein.

| Metrik | Chain | FIFO | Mixed |
|---|---|---|---|
| Max Speicherverbrauch | 1.099 | 2 | 2 |
| Durchschn. Latenzzeit | 5000 | 150 | 150 |
| Max Latenzzeit | 9901 | 199 | 199 |

**Tabelle 2: Vergleich über Speicherverbrauch und Latenzzeit zwischen FIFO – GREEDY- MIXED bei vielen Tupeln, jedoch geringer Ankunftsrate**

| Metrik | Chain | FIFO | Mixed |
|---|---|---|---|
| Max Speicherverbrauch | 1.9 | 9 | 1.9 |
| Durchschn. Latenzzeit | 595 | 550 | 595 |
| Max Latenzzeit | 910 | 910 | 910 |

**Tabelle 3: Vergleich über Speicherverbrauch zwischen FIFO – GREEDY- MIXED bei wenigen Tupeln, jedoch hoher Ankunftsrate**

In Tabelle 2 kann man deutlich sehen, dass sehr flache Stücke in den Progress Charts zu erheblichen Verschlechterungen der Antwortzeiten beim Chain-Algorithmus führen aufgrund des Aushungerns dieser Teilstücke. Tabelle 3 zeigt, dass durch die mangelnde Adaptivität der FIFO-Strategie der Speicherverbrauch sehr hoch ist im Vergleich zu den beiden anderen Strategien. Der Mixed-Algorithmus stellt eine Möglichkeit dar die beiden negativen Eigenschaften von Chain und FIFO zu entschärfen und einen ausgeglichenen Mittelweg zu gehen. Durch Heraufsetzen des Schwellenwertes nähert sich Mixed im Verhalten FIFO an und eine Verminderung des Parameters in Richtung 0 führt zum herkömmlichen Chain-Verfahren. Dies wurde experimentell in mehreren Versuchen geprüft und bestätigt. Die folgende Tabelle 4 bestätigt diese These und weist die Unterschiede auf, welche die vom Benutzer definierten Werte verursachen:

| Algorithmus | Max Queue-Größe (KB) | Max Latenzzeit (ms) |
|---|---|---|
| Chain | 2.5 | 76.2 |
| FIFO | 17.1 | 24.1 |
| Chain-Flush (Latenzschwelle = 50 ms) | 9.2 | 50.9 |
| Chain-Flush (Latenzschwelle = 10 ms) | 17.6 | 10.6 |
| Mixed (Steigungsschwelle = 0.0001) | 2.9 | 17.2 |
| Mixed (Steigungsschwelle = 0.0002) | 6.3 | 34.4 |
| Mixed (Steigungsschwelle = 0.0008) | 17.1 | 24.2 |

**Tabelle 4: Vergleich über Peak-Werte zwischen verschiedenen Strategien**

Eine Herausforderung und sehr aktives Forschungsgebiet sind adaptive Algorithmen, welche die Schwellenwerte für den Mixed-Algorithmus während der Laufzeit so wählen, dass die für die jeweilige Anwendung erforderliche Dienstgüte eingehalten wird.

## 2.2 Vorstellung der PIPES-Infrastruktur

In vielen Systemen, welche mit Datenströmen arbeiten, wird die Anfrageverarbeitung mehrerer paralleler Anfragen durch isolierte Operatorbäume beschrieben. In der Praxis existieren jedoch sehr oft Überlappungen in Teilgraphen zwischen verschiedenen Operatorbäumen. Um nun eine Lastreduktion durchzuführen werden genau solche Überlappungen gesucht und gemeinsame Teilstücke nur einmal ausgeführt. Man spricht nun nicht mehr von isolierten Operatorbäumen, sondern von einem Graph von Operatoren.
In diesem Zusammenhang wird jetzt PIPES (Public Infrastructure for Processing and Exploring Streams) vorgestellt, eine bibliotheksorientierte Infrastruktur zur Entwicklung von Datenstrommanagementsystemen, die auf die jeweilige Anwendung zugeschnitten werden kann. PIPES ist eine Erweiterung von XXL, einer Java-Bibliothek für komplexe Anfragebearbeitung über heterogene, relationale Datenbanken. In dem PIPES-Modell werden drei Typen von Knoten unterschieden: Quellen, Operatoren und Senken.

Quellen:
Eine Quelle liefert Tupel an eine Menge von Senken, welche bei der Quelle angemeldet sind

Senken:
Eine Senke konsumiert Tupel von den bei der Senke angemeldeten Quellen.

Operatoren:
Ein Operator, auch Pipe genannt, vereint die beiden Funktionalitäten von Quelle und Senke. Er nimmt Tupel auf, verarbeitet diese und leitet sie an alle zu ihm gehörenden Senken weiter.

Dieses Prinzip der Quellen und Senken nennt man auch den Publish-Subscribe-Mechanismus.

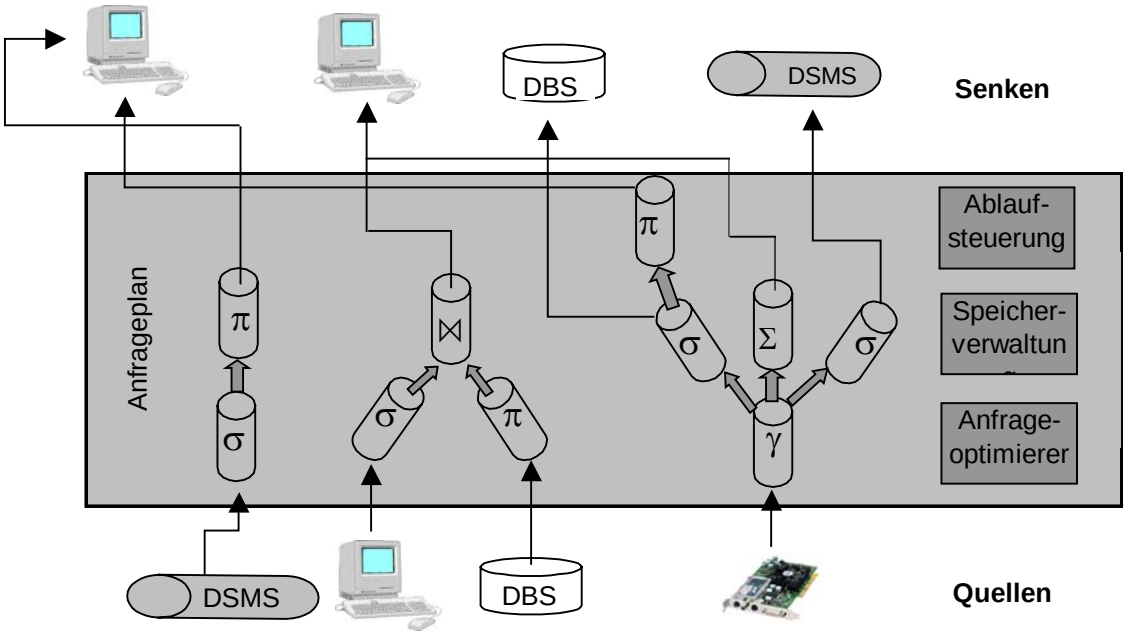


**Abbildung 8: Architektur von PIPES**

Abbildung 8 zeigt die drei zentralen Komponenten zur Laufzeitumgebung von PIPES – Ablaufsteuerung, Speicherverwaltung und der Anfrageoptimierer.

Ablaufsteuerung:
Darunter wird der Scheduler der PIPES-Infrastruktur verstanden. Dieser ordnet den verschiedenen Knoten im System CPU-Zeit zur Anfrageverarbeitung zu. Entscheidungsgrundlage für die Zuteilung stellen benutzerdefinierte QoS-Werte dar wie Antwortzeit und Datenverlustrate.

Speichermanager:
Diese Komponente ist für die Verwaltung von Haupt- und Externspeicher zuständig. Eine adaptive Umverteilung ist zur Laufzeit möglich.

Optimierer:
Dieser entwirft zu jeder Anfrage eine Menge von äquivalenten Anfragepläne. Danach vergleicht er diesen mit dem momentanen Anfragegraphen. Mit Hilfe einer Kostenfunktion wird der Anfragegraph hinzugefügt, der für das System den minimalsten Neuaufwand bedeutet. Die neuen Knoten werden dann über das Publish-Subscribe-Prinzip in den Graphen eingefügt.

Die verschiedenen Komponenten sind in XXL als Pakete vorhanden und erlauben uns durch deren Kombination ein adaptives System zu erstellen. Im Folgenden werden die in PIPES integrierten Techniken im Umgang mit der Thread-Verwaltung, die einen Forschungsschwerpunkt bilden, vorgestellt und bewertet [10].

### 2.2.1 Problemfeld der CPU- und Speicherverwaltung in PIPES

Die in der Kapitel 1 vorgestellete Problematik bezüglich OTS und GTS stellte für viele Forschungsbereiche eine Motivation dar ein Verfahren zu entwickeln, welches die Vorteile von OTS und GTS vereint und versucht deren Schwächen zu vermeiden. Ansätze zur Lösung dieser Problematik sind ein Hauptgebiet im PIPES-Projekt, wodurch es dem Fachbereich Informatik der Universität Marburg gelungen ist ein solches Verfahren mit dem Namen Hybrid-Multi-threaded-scheduling-Strategie (HMTS) in ihr PIPES-System zu integrieren.

### 2.2.2 Hybrid Multi-Threaded Scheduling

Wie bei der Vorstellung der einfachen Scheduling-Verfahren geschildert, werden in DSMS im Anfragegraph zwischen aufeinanderfolgende Operatoren Warteschlangen zur Entkopplung benutzt. Diese Technik bedeutet für das DSMS einen zusätzlichen Scheduling- und Speicheraufwand. PIPES hingegen bedient sich einer Technik namens direct interoperability (DI). Darin werden Operatoren direkt, also ohne Zwischenpuffer, miteinander verbunden. Ist ein Operator mit der Verarbeitung eines Tupels fertig, so wird dieses direkt an den folgenden Operator weitergegeben, welcher sofort ohne zusätzliches Scheduling des Systems mit der Bearbeitung fortfährt. Dieses Prinzip bezeichnet man auch *implizites* Scheduling. Solche Operatoren, welche durch DI aus mehreren physischen Operatoren entstanden sind, werden als *virtuelle* Operatoren bezeichnet. In der Abbildung 9 wird dieses Prinzip an zwei Stellen eingesetzt.
Das komplementäre Konzept zum impliziten Scheduling ist exklusives Scheduling. Dabei arbeiten Operatoren ausdrücklich nur dann, wenn diese durch den Scheduler CPU-Zeit zur Verfügung gestellt bekommen.
HMTS ist ein dreischichtiges Scheduling-Verfahren. Dieses umfasst die den beiden eingangs erwähnten Strategien OTS und GTS, sowie das Verfahren direct interoperability (DI). Auf zwei Ebenen kann in HMTS ein explizites Scheduling angewendet werden, wobei das oben

erwähnte DI implizites Scheduling durchführt. Dies wird durch den in Ebene zwei eingeführten Buffer Pipe Scheduler realisiert. Dieser verwaltet die physischen und virtuellen Operatoren, die er über die zugehörigen Buffer Pipes ansteuern kann. Mit dieser Technik werden nun Teile des Anfragegraphen in einem eigenen Thread laufen gelassen. Dadurch lässt sich vermeiden, dass das System aufgrund einer Wartesituation eines Operators komplett stillgelegt wird. HMTS erlaubt eine dynamische Adaptierung der Thread-Anzahl und findet ausgewogene Lösungsmöglichkeiten zwischen dem Grad an Nebenläufigkeit und den dafür erforderlichen Mehraufwand.
Ein Beispiel für den Aufbau des dreischichtigen Schedulings ist in Abbildung 9 aufgezeigt. In der ersten wird die Entscheidung getroffen, welche physischen Operatoren zu virtuellen Operatoren zusammengesetzt werden. Zur Entkopplung werden in PIPES Buffer-Pipe-Operatoren benutzt, welche die Funktion einer Queue übernehmen.
In der zweiten Ebene werden alle Operatoren der ersten Ebene in Teilgraphen aufgeteilt. Jede dieser Teilmengen wird an den Eingängen mit Buffer-Pipe-Operatoren versehen. Ab dieser zweiten Ebene ist ein explizites Scheduling möglich. Dieses Scheduling auf zweiter Ebene wird mit einem sogenannten Buffer Pipe Scheduler (BPS) realisiert, dessen Verhalten durch ein Strategie-Pattern gesteuert werden kann (z.B. triviale Strategien wie FIFO, Greedy). Die Buffer-Pipe-Scheduler mit ihren Buffer-Pipes und Operatoren laufen auf dieser zweiten Ebene in einem eigenen Thread.
In Ebene 3 werden mehrere BPS mit den dazugehörigen Pipes und Buffer Pipes aus Ebene 2 nebenläufig in jeweils eigenen Threads ausgeführt. Diese Nebenläufigkeiten werden mit Hilfe eines Thread Schedulers (TS) höchster Priorität verwaltet.

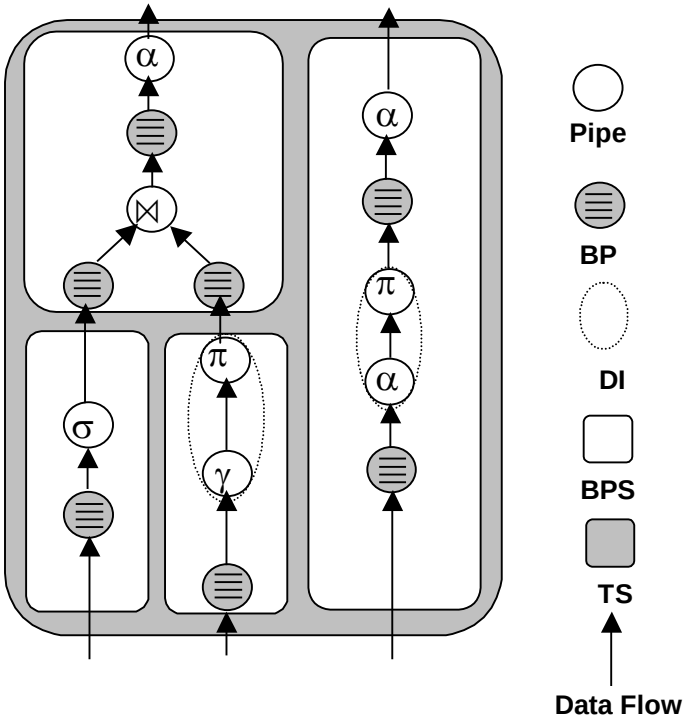


**Abbildung 9: Beispiel für die drei Scheduling Ebenen in PIPES**

Das PIPES-Scheduling-Konzept erfordert besondere Techniken, da aufbauend auf der Thread-Verwaltung von Java die anwendungsspezifische Strategie von PIPES simuliert werden muss. Dies ist die Aufgabe des Thread Schedulers (TS), der mittels eines Prioritäten Konzepts die oberste Ebene in PIPES realisiert. Die Steuerung der Ausführungsreihenfolge erfolgt dynamisch, indem man dem nächsten auszuführendem Arbeits-Thread für kurze Zeit maximale Priorität zuweißt. Dadurch wird im Allgemeinem gewährleistet, dass Java diesen

male Priorität zuweißt. Dadurch wird im Allgemeinem gewährleistet, dass Java diesen Thread als nächstes auswählt.

## 2.2.3 Bewertung und experimentelle Auswertungen

Da es viele gute Scheduling-Strategien gibt, welche jedoch aufgrund des Mehraufwands für die von uns betrachteten DSMS in Bezug auf Speicher und Latenzzeit unbrauchbar sind (z.B. bei Echtzeitanforderungen), soll der fast zu vernachlässigenden Mehraufwand in PIPES durch ein Experiment aufgezeigt werden.
Das Experiment soll nochmals die Vorteile von HMTS gegenüber GTS aufzeigen. Der Anfragegraph im Experiment in Diagramm 1 besteht aus zwei aufeinanderfolgende Operatoren, welche eine Selektivität von 0,3 und $9*10^{-4}$ besitzen. HMTS besitzt den geringeren Speicherverbrauch, sowie eine minimal kürzere Laufzeit im Vergleich zu GTS. Das atomare Verarbeiten von Tupeln führt zu sogenannten Stalls bei den Operatoren, d.h. die Warteschlangen vor den Operatoren füllen sich sehr schnell bei hohem Datenaufkommen, was den Verbrauch an Speicher drastisch erhöht.
Das Experiment zeigt, dass die einfachen Verfahren OTS und GTS suboptimales Verhalten zeigen und dass HMTS bei guten Antwortzeiten den Puffer minimal hält.

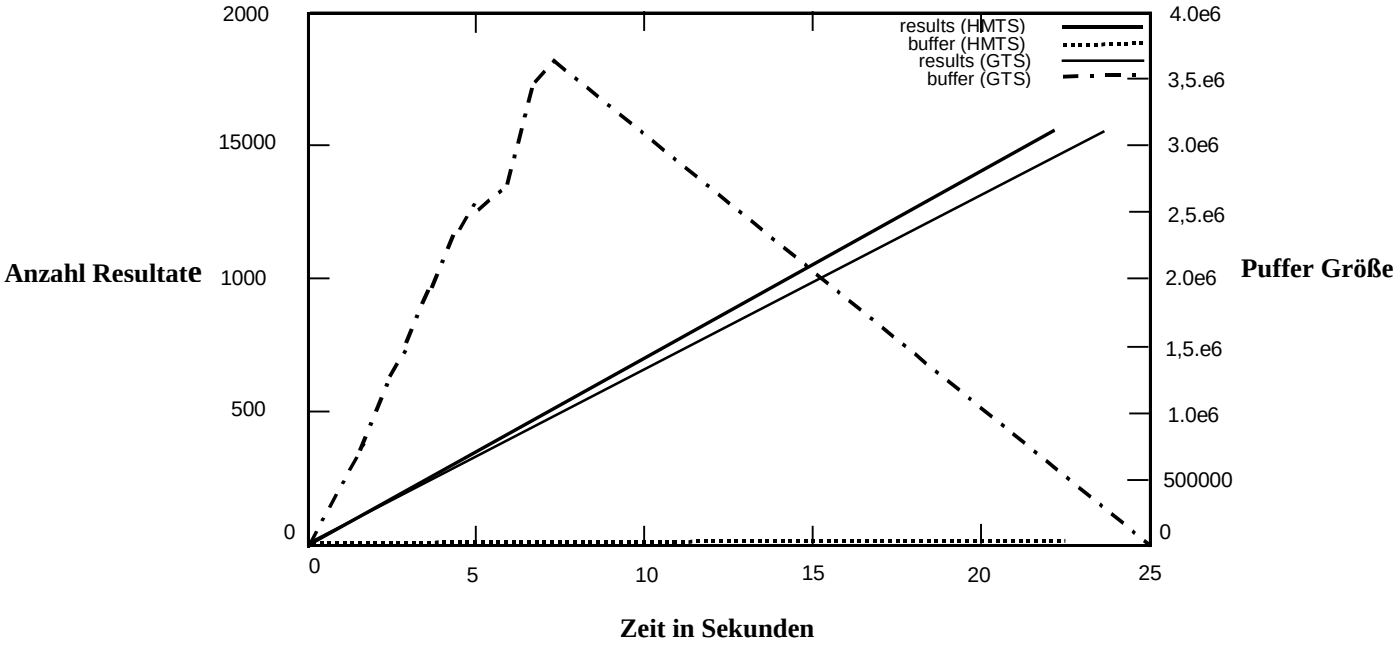


**Diagramm 1: Direkter Vergleich zwischen HMTS und GTS**

Momentane Forschungsarbeiten innerhalb des Projekts konzentrieren sich zur Zeit auf drei große Leitfragen:

- Wie teilt man die Operatoren optimal auf die Gruppen der verschiedenen BPS auf und welche Operatoren werden zu physischen Operatoren zusammengefasst?
- Wie verwaltet man die beiden obersten Schichten in Bezug auf QoS-Fragen?
- Wie führt man eine Neuordnung zur Laufzeit aus?

Momentane Forschungsarbeiten befassen sich mit diesen drei Problemstellungen. Es existieren generische Ansätze, welche eine dynamische und speicheradaptive Schätzung von wichtigen Statistiken auf Datenströmen erlauben. Damit werden dann Selektivitätsabschätzungen für Anfragen unterstützt. PIPES würde somit ähnliche prioritätsorientierte Strategien wie das vorher angesprochene Chain Scheduling unterstützen. Diese sind wie man gesehen hat sehr gut im Speicherverbrauch, jedoch zu Lasten der Antwortzeiten.
Auf der Website der Uni Marburg
„http://dbs.mathematik.uni-Harburg.de/Home/Research/Projects/XXL/“ ist mittlerweile ein Online-Demo von PIPES kostenlos zur Verfügung gestellt worden, mit der man sich von der Funktionsweise des Systems selbst ein Bild machen kann.

## 2.3 Vorstellung des AURORA-Modells

Der Unterschied zu den bisher betrachteten Systemen ist die Beschränkung auf die Verarbeitung von Datenströmen in AURORA. In AURORA wird jedes ankommende Tupel mit einer exakten Angabe der Datenquelle und einem Anfangszeitstempel versehen, so dass die jeweils aktuellen QoS überwacht werden können, wie z.B. die Latenzzeit. Diese ankommenden Datenströme werden versucht vom DSMS so zu verarbeiten, wie der Benutzer dies mit Hilfe von Werten für die Dienstgüte angegeben hat. In AURORA wird die Verarbeitung von kontinuierlichen Anfragen mit Hilfe des boxes-and-arrows-Paradigmas beschrieben. Die in das System einströmenden Tupel durchlaufen einen gerichteten, azyklischen Graphen. Die ankommenden Tupel werden durch die im Anfrageplan existierenden Operatoren (boxes) verarbeitet und im Anfragegraphen als Datenstrom an einen oder mehrere Operatoren weitergeleitet. Abbildung 10 zeigt das beschriebene Schema in einem Anfrageplan von AURORA.
Dem Benutzer wird eine GUI zur Verfügung gestellt, mit der man den Anfragegraphen direkt modifizieren kann. Auch das Stellen von Ad-hoc-Anfragen wird in diesem System unterstützt. Dieses Feature wird durch die sogenannten Connection Points ermöglicht, welche Zugriff auf historische Daten haben. Mit Hilfe der GUI können Connection Points dynamisch zur Laufzeit an beliebigen Kanten im System definiert werden. Strömen nun Tupel über diese markierten Kanten, so werden diese für eine bestimmte Zeit zwischengespeichert. Inwieweit eine solche Datensicherung durchgeführt wird kann ebenfalls durch den Benutzer bestimmt werden.
Problematisch an diesem Konzept ist das Stellen von Ad-hoc-Anfragen, wenn an den benötigten Stellen im Anfrageplan keine Connection Points verfügbar sind. In diesem Punkt ist das AURORA-System noch recht unflexibel [3].

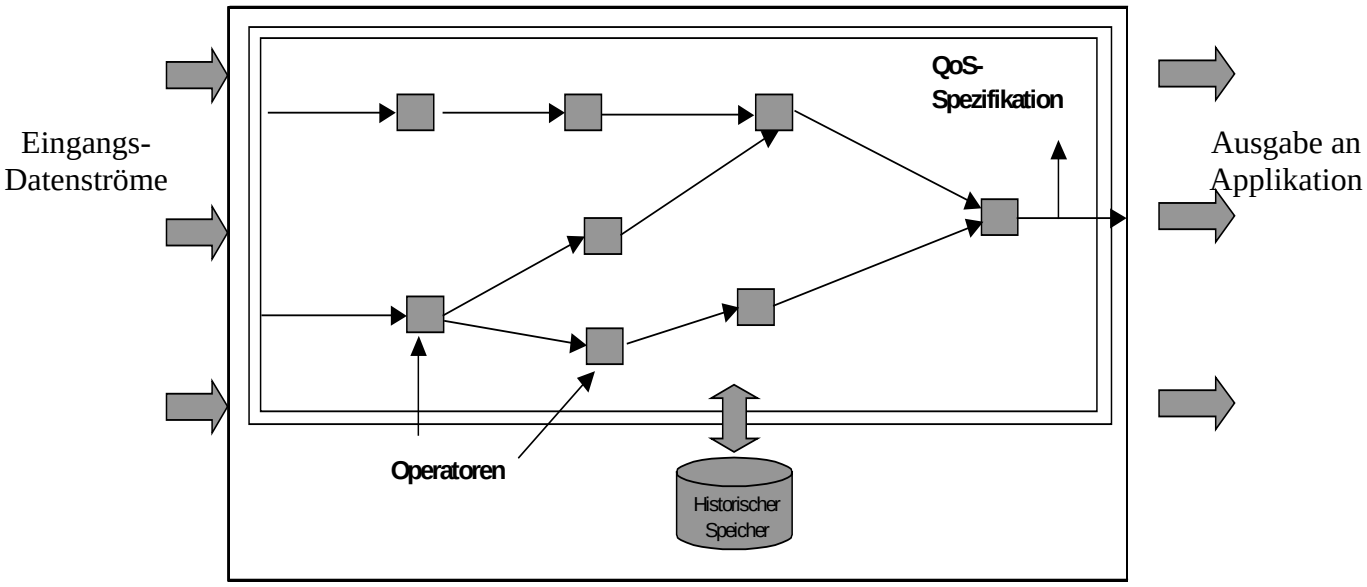


**Abbildung 10: Anfrageverarbeitung in AURORA**

## 2.3.1 Zwei-Phasen-Scheduling

Im AURORA-Modell wird das Scheduling mehrerer kontinuierlichen Datenströme in zwei Phasen vollzogen. In der ersten Phase steht die Entscheidung, welche der kontinuierlichen Anfragen bearbeitet wird, an. Grundlage der Entscheidung sind die vom Benutzer definierten QoS-Werte. In der zweiten Phase wird dann entschieden, wie diese Anfrage im einzelnen abgearbeitet wird, d.h. in welcher Reihenfolge die Operatoren abgearbeitet werden. Das Ergebnis dieser beiden Phasen stellt dann den Scheduling-Plan dar.
Um den Aufwand beim Scheduling und den Operatoren zu minimieren setzt AURORA sehr viel auf Batch-Techniken. Im folgenden werden jetzt einige Batch-Techniken des Systems vorgestellt und anschließend bewertet.

**Operator Batching (Superbox Processing)**
Die Technik des Operator Batching entspricht der bei PIPES vorgestellten Strategie DI. Eine Superbox ist eine Zusammenfassung mehrerer Operatoren, welche als atomare Gruppe abgearbeitet werden. Diese Technik wird genutzt, um zum einen den Scheduling-Overhead zu minimieren, da jetzt weniger Scheduling-Schritte absolviert werden müssen. Zum anderen muss der Speichermanager jetzt nicht mehr so viel Speicherplatz für Warteschlangen reservieren, da die Operatoren nicht mehr entkoppelt sind. Die Abarbeitung erfolgt somit implizit. Eine solche Zusammenfassung wird von AURORA nur ausgehend vom letzten Operator einer Anfrage durchgeführt. Der Grund hierfür ist die Kontrolle über die Ausgabeergebnisse in Bezug auf die definierten Werte für die Dienstgüte und außerdem würde sonst die Suche nach geeigneten Superbox-Kombinationen in AURORA-Netzwerken zu komplex werden aufgrund der Kombinationsvielfalt.
Die erste Phase dieser Technik wird im AURORA-System statisch durchgeführt. Dies bedeutet, dass schon vor dem Beginn der Anfragebearbeitung potentielle Superboxes identifiziert werden.

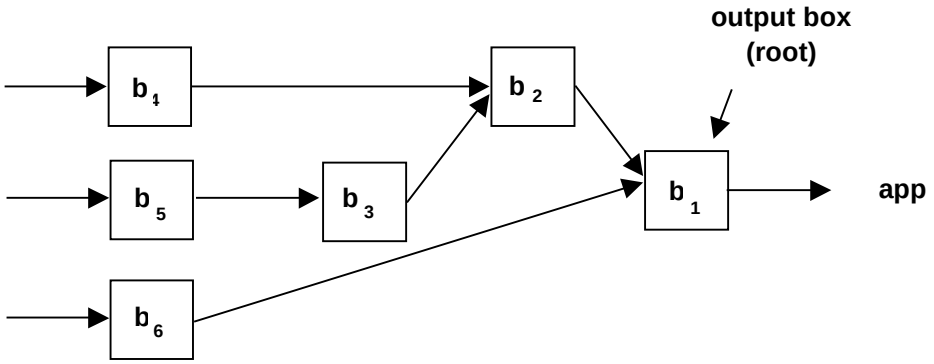


**Abbildung 11: Beispielhafter Anfragebaum mit sechs Operatoren**

Superboxes können dann nacheinander über verschiedene Scheduling-Vorschriften, wie z.B. Round Robin angesprochen werden.
Hat man dann die Superbox bestimmt, welche bearbeitet werden soll, so wird dann im zweiten Schritt die Reihenfolge innerhalb der Superbox über spezielle Verfahren festgelegt, welche im kommenden Abschnitt noch vorgestellt werden. Diese Technik nennt man Traversierung und hat das Ziel alle Tupel in den Queues abzuarbeiten. Zur näheren Untersuchung werden drei verschiedene Algorithmen unter die Lupe genommen, welche verschiedene Stärken in den Bereichen des Durchsatzes, der Latenzzeit und des Speicherverbrauchs besitzen [5].
Betrachten wir zur Verdeutlichung unserer Strategien den Anfragebaum in Abbildung 11. Dort wird der Einfachheit halber davon ausgegangen, dass die Superbox den kompletten Anfragegraphen umfasst:

**Min-Cost:**
Der Min-Cost-Algorithmus hat die Minimierung der Bearbeitungskosten pro Tupel (daraus resultierend ist eine Maximierung des Durchsatzes) durch eine Minimierung der Anzahl der Operatoraufrufe zum Ziel [3]. Dies wird durch eine Umordnung in Post-Ordnung innerhalb der Superbox erreicht, d.h. ein Operator darf erst beachtet werden, wenn alle Operatoren im Baum unterhalb diesem schon bearbeitet worden sind. In diesem Min-Cost Algorithmus werden alle Tupel, die zu Beginn der Ausführung vorliegen, verarbeitet, wobei jeder Operator nur einmal zur Ausführung kommt. Sei p die Bearbeitungskosten pro Tupel, o der Overhead eines Operatoraufrufs und die Selektivität aller Operatoren sei 1. Bei jedem Operator liegt zu Beginn dieses Verfahrens bereits ein Tupel vor. Eine mögliche Traversierung der Superbox sieht somit folgendermaßen aus:

b4⇒b5⇒b3⇒b2⇒b6⇒b1

Die Berechnung der Kosten, um alle Tupel zu bearbeiten, liefert uns somit 6o (6 Operatoraufrufe).
Die Bearbeitungskosten für die Ausgabe aller Tupel liegt bei 15p, was aus Tabelle 5 ersichtlich wird.

| | Anzahl der Tupel vor Operator: | | | | | |
|---|---|---|---|---|---|---|
| **Zeit-einheit** | **B1** | **B2** | **B3** | **B4** | **B5** | **B6** |
| **0** | 1 | 1 | 1 | 1 | 1 | 1 |
| **1** | 1 | 2 | 1 | 0 | 1 | 1 |
| **2** | 1 | 2 | 2 | 0 | 0 | 1 |
| **3** | 1 | 3 | 1 | 0 | 0 | 1 |
| **4** | 1 | 4 | 0 | 0 | 0 | 1 |
| **5** | 2 | 3 | 0 | 0 | 0 | 1 |
| **6** | 3 | 2 | 0 | 0 | 0 | 1 |
| **7** | 4 | 1 | 0 | 0 | 0 | 1 |
| **8** | 5 | 0 | 0 | 0 | 0 | 1 |
| **9** | 6 | 0 | 0 | 0 | 0 | 0 |
| **10** | 5 | 0 | 0 | 0 | 0 | 0 |
| **11** | 4 | 0 | 0 | 0 | 0 | 0 |
| **12** | 3 | 0 | 0 | 0 | 0 | 0 |
| **13** | 2 | 0 | 0 | 0 | 0 | 0 |
| **14** | 1 | 0 | 0 | 0 | 0 | 0 |
| **15** | 0 | 0 | 0 | 0 | 0 | 0 |

**Tabelle 5: Berechnung der Ausgabezeit aller Tupel in Min-Cost**

Dies bedeutet, dass man 15 Zeiteinheiten benötigt, um jedes Tupel (in diesem Beispiel befindet sich vor jedem Operator *ein* Tupel) zu bearbeiten und an die Applikation auszugeben. Im Mittel ergibt sich für Tupel dann der Wert: 12,5p + 6o.

**Min-Latency:**
Die durchschnittlichen Latenzzeit für die Beantwortung eines Tupels kann man dadurch verringern, dass man Tupel, welche im Anfragebaum schon weit fortgeschritten sind, vorrangig bearbeitet. Um die Traversierung in diesem Fall durchzuführen, werden die Operatoren aufsteigend in Bezug auf die Ausgabekosten sortiert. Für die Berechnung der Ausgabekosten eines Operators wird nun folgende Metrik eingeführt:

$$\textbf{o\_Sel(b)} = \prod_{k \in D(b)} sel(k)$$

$$\textbf{Cost(b)} = \sum_{k \in D(b)} cost(k) / o\_sel(k)$$

In den angegebenen Formeln steht D(b) für alle nachfolgenden Operatoren, die inklusive b noch passiert werden müssen, um das Tupel komplett bis zur Ausgabe zu bearbeiten.
Beginnend mit einer leeren Sequenz wird somit mit dem kleinsten dieser Kostenwerte begonnen. Danach folgt die Operatorsequenz mit dem zweit kleinsten Kostenwert. Anhand er Formeln lässt sich schnell erkennen, dass der letzte Operator vor der Ausgabe immer derjenige mit dem geringsten Kostenwert ist. Somit lautet die Operatorsequenz mit dem kleinsten Kostenwert b1. Da alle Operatoren Selektivität 1 besitzen hat man nun die Option zwischen zwei Operatorsequenzen als zweitbilligste Operatorsequenz, nämlich b2⇒b1 oder b6⇒b1. In dem

Beispiel für die Traversierung haben wir uns für b2⇒b1 entschieden. Führt man dieses Verfahren weiter, so kommt man im Beispiel auf folgende Traversierung (man bedenke, dass alle Operatoren die Selektivität 1 besitzen) :

b1⇒b2⇒b1⇒b6⇒b1⇒b4⇒b2⇒b1⇒b3⇒b2⇒b1⇒b5⇒b3⇒b2⇒b1

Diese Traversierung benötigt neun zusätzliche Operatoraufrufe im Vergleich zu Min-Cost und die gesamten Ausführungskosten belaufen sich auf 15p+15o. Der Mittelwert für Min-Latency beträgt 7,17p+7,17o. Was man erkennt ist, dass Min-Cost immer die geringere Gesamt-Ausführungszeit besitzt als Min-Latency. Bei sehr hohem Aufkommen an Tupeln liefert Min-Cost die bessere Performance, da der Durchsatz höher ist.

**Min-Memory:**
Diese Strategie wird verwendet, um das Maximum von Daten pro Zeiteinheit zu bearbeiten. Mit anderen Worten: Es werden die Operatoren ausgeführt, welche den meisten Speicher im System wieder freigeben. Der Algorithmus verwendet folgende Formel zur Ermittlung der Memory-Reduktions-Rate Mem_rr eines Operators b :

$$\mathbf{Mem\_rr(b) = (tsize(b) * (1\text{-}Sel(b))) \div cost(b)}$$

wobei tsize(b) die Menge an n-Tupel in der Eingangswarteschlange ist. Der Ausdruck 1-Sel(b) drückt aus, wie viele Tupel an diesem Operator verworfen werden. Je höher dieser Wert ist, desto besser wirkt dies sich auf die Speicherfreigabe aus. Natürlich müssen auch die Selektivität und die Kosten der Operatoren für die Bearbeitung eines Tupels bekannt sein. Gehen wir also nun mal von folgenden Selektivitäten und Kosten für die Operatoren b1 bis b6 aus:

| Operator | Selektivität | cost |
|---|---|---|
| 1 | 0,9 | 2 |
| 2 | 0,4 | 2 |
| 3 | 0,4 | 3 |
| 4 | 1,0 | 2 |
| 5 | 0,4 | 3 |
| 6 | 0,6 | 1 |

**Tabelle 6: Kosten und Selektivitäten der Operatoren b1 bis b6**

Mit Hilfe der Tabelle 6 kann man nun die Speicherfreigaben der Operatoren berechnen. Diese Berechnung muss in jedem Schritt, also dynamisch, erneut durchgeführt werden. Zu Beginn des Szenarios ergeben sich die folgenden Mem_rr-Werte der Operatoren b1 bis b6:

- Mem_rr(b1) = 0,05
- Mem_rr(b2) = 0,3
- Mem_rr(b3) = 0,5
- Mem_rr(b4) = 0
- Mem_rr(b5) = 0,2
- Mem_rr(b6) = 0,4

Durch das iterative Anwenden der Strategie ergibt sich nun die folgende Traversierung:

b3⇒b6⇒b2⇒b5⇒b3⇒b2⇒b1⇒b4⇒b2⇒b1

Man beginnt mit den Operatoren mit maximalen Mem_rr-Wert. Während der Berechnung kann sich diese Traversierung natürlich ändern, wenn z.B. ein Operator keine Tupel ausgibt aufgrund seiner Selektivität.
In diesem Anfragebaum sind die folgenden Speicheranforderungen durch Berechnungen eines jeden Operators aufgrund der durchschnittlichen Operatoraufrufe nach Ausführung zu Stande gekommen:

Mem(Min-Memory) = 36
Mem(Min-Cost) = 39
Mem(Min-Latency) = 40

Problematisch an dieser Technik ist, wie schon in anderen Systemen gesehen, die unvorhersehbare Antwortzeit. Haben Operatoren

**Tupel-Batching (Train Processing)**
Der Begriff *train* steht für eine Sequenz an Tupeln, welche in einem einzigen Operatoraufruf verarbeitet werden. Damit sollen die Gesamtkosten für die Tupelverarbeitung reduziert werden. Dies geschieht auf mehreren Wegen. Zum einen wird die Anzahl der Operatoraufrufe und die damit verbundenen Kosten für CPU und Speicher reduziert. Genau wie beim Superbox-Scheduling wird auch hier die Speicherverwaltung entlastet. Dies beruht auf den reduzierten Queue-Einträgen und der Minimierung Kontextwechseln. Die Entscheidung wie viele Tupel gleichzeitig vom Operator konsumiert werden trifft in AURORA der Scheduler [3]. Intelligente Strategien für die Wahl der Anzahl von Tupeln in einer *train* sind wichtig für den effizienten Einsatz von Tupel-Batching.

## 2.3.2 QoS-gesteuertes Scheduling

Im kommenden Abschnitt wird die Ermittlung von Prioritäten für Operatoren beschrieben. Diese Scheduling-Methode in AURORA wird als Priority Assignment bezeichnet. Dazu werden QoS-Spezifikationen und Latenzzeit-Anforderungen der aktuell ablaufenden Applikationen verwendet. Prioritäten von Operatoren werden anhand zweier Faktoren bestimmt – die Dringlichkeit und der Nutzen eines Operators.
Um den aktuellen Zustand der Dienstgüte zu kontrollieren und auf eventuell nicht mehr zu akzeptierende Ausgaben reagieren zu können, wird der Verlauf der Dienstgüte in einem Diagramm festgehalten. Diagramm 2 zeigt eine solche QoS-Kurve mit kritischen Punkten, an denen sich die Dienstgüte stark verschlechtert.
Der Nutzen ist definiert als die Steigung im Graphen an der Stelle eol(b).

**utility(b) = gradient (eol(b))**

Die Abkürzung eol steht für expected output latency. Dieser stellt die aktuelle Position auf der QoS-Latenz-Kurve dar. Dieser Wert berechnet sich durch die Latenzzeit des Operators selbst und die Bearbeitungsdauer aller nachfolgenden Operatoren.

$$\mathbf{Eol(b) = latency(b) + cost(D(b))}$$

Die Zusammenhänge werden in Diagramm 2 anschaulich dargestellt.

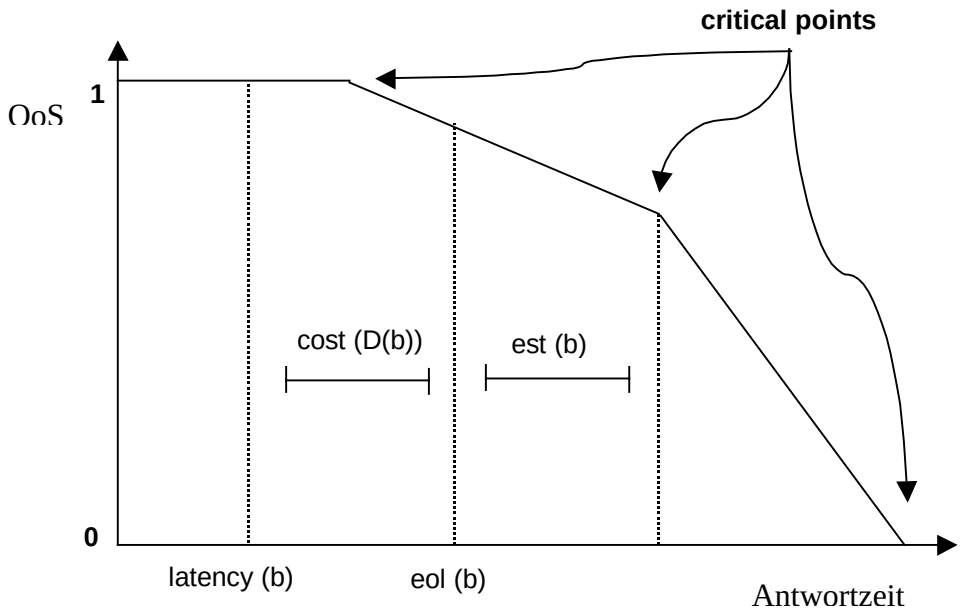


**Diagramm 2: QoS-Kurve mit kritischen Punkten**

Die in Diagramm 2 angegebenen kritischen Punkte sind Stellen, an den sich die QoS-Funktion stark verändert. Der Wert est(b) gibt an, wie weit der Operator b von einem kritischen Punkt entfernt ist (Slack-time). Je kleiner est(b), desto mehr sollte darauf geachtet werden, dass die Ausgabe noch in den vom Benutzer definierten Werten für die Dienstgüte liegt. Ist die Einhaltung der Dienstgüte gefährdet, so muss durch bevorzugte Behandlung von Operatoren versucht werden das Nichterfüllen zu verhindern. In AURORA wird mit dem folgenden 2-Tupel gearbeitet, um Prioritäten von Operatoren auszudrücken:

$$\mathbf{priority(b) = ( utility(b) , – est(b) )}$$

In jedem Scheduling-Schritt kann man nun die verschiedenen Operatoren nach den Prioritäten sortieren. Die Operatoren werden zuerst nach ihrem Nutzen und dann nach Dringlichkeit ausgewählt.
Was jedoch ein Problem genau wie bei allen anderen Scheduling-Strategien darstellt, ist das Szenario eines hohen Datenaufkommens. Das Problem liegt darin, dass die Dienstgüte ab einem bestimmten Punkt bei zu hohen Belastungen einbricht. In Versuchen wurde bei Vergleichen der konventionellen Round-Robin-Strategie und dem AURORA-QoS-Scheduling interessante Ergebnisse festgestellt. Es hat sich herausgestellt, dass ab einem Schwellenwert beide Scheduling-Strategien bezüglich QoS in Richtung Null einbrechen. Eine mögliche Lö-

sung, welche für solche Probleme prädestiniert ist, ist das Zurückgreifen auf Approximationstechniken. Die hier eingesetzte Technik nennt sich Bucketing. Ohne Approximationstechnik würde für das Ausrechnen der p-Tupel und das anschließende Sortieren ein Aufwand von O(n * log(n)) entstehen. Es sind daher Lösungen erforderlich, die unter diesem Aufwand bleiben. Dafür wird der Bereich des Nutzwertes und der Dringlichkeit schon vor Anlauf des Programms in sogenannte Buckets unterteilt (siehe Abbildung 12). Ziel ist es nun die verschiedenen Operatoren zur Laufzeit mit einem Aufwand von O(1) anhand der Prioritätswerte in den zum Operator passenden Bucket unterzubringen. Die Buckets werden dann vom Scheduler in absteigender Reihenfolge durchlaufen und die darin einsortierten Operatoren in die Ausführungsqueue eingereiht [3].

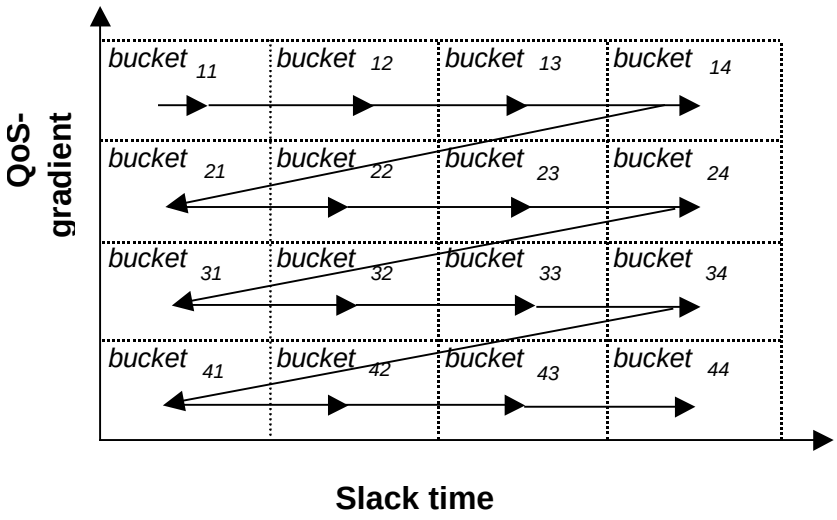


**Abbildung 12: Durchlauf der Buckets**

Für die Einteilung der Operatoren in Buckets gemäß ihrer Priorität und ihrer Nützlichkeit benötigt man zwei Hilfstabellen. Zum einen braucht man einen Gradient-Latency-Graph (Abb. 14), in dem Steigung, Latenzzeit und Bucket-Zugehörigkeit betrachtet werden und zum anderen einen Slack-Latency-Graph (Abb. 14), der Latenzzeit, Slack time und Bucket-Zugehörigkeit enthält. Diese beiden Hilfstabellen werden durch die vom Benutzer definierten Grenzen erstellt und die Bucket-Grenzen ermittelt. Abhängig von den Ansprüchen an die Dienstgüte stellen somit die einzelnen Buckets ein Maß für die Abstraktion dar, d.h. je weniger Buckets verwendet werden, desto ungenauer die Annährung an das gewollte Verhalten. Dies liegt daran, dass man aufgrund weniger Zuordnungen nicht mehr ein hohes Maß an Kontrolle besitzt und somit manche Operatoren erst sehr spät CPU-Zeit zugesprochen bekommen. Allgemein kann man also sagen, dass die Minimierung an Buckets eine Maximierung des Approximationsgrades mit sich bringt.

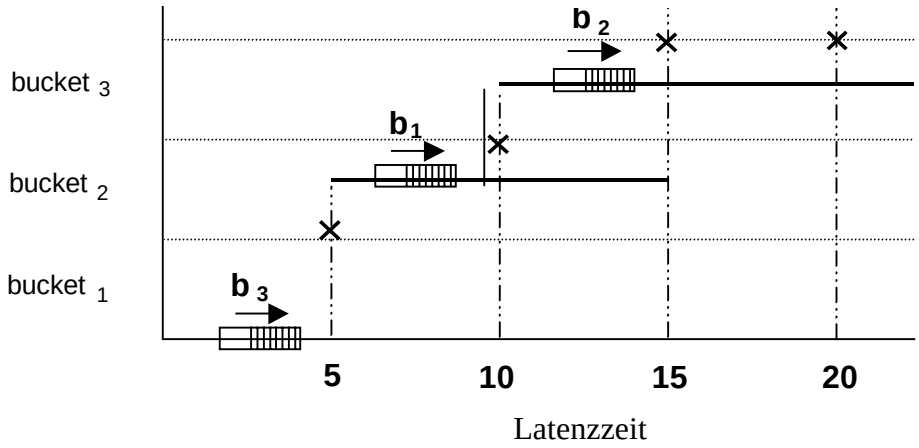


**Abbildung 13: Gradient-Latency-Graph**

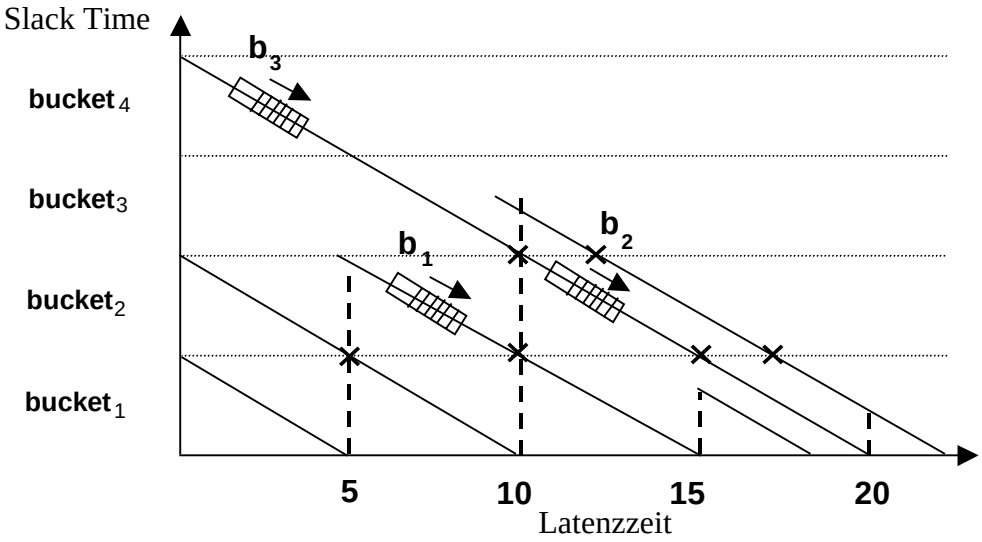


**Abbildung 14: Slack-Latency-Graph**

Es werden vom Benutzer nun sogenannte Schwellenwerte für die Dienstgüte definiert, anhand deren die Bucket-Zugehörigkeit ermittelt werden soll. Die Einteilung der Buckets in unseren Tabellen steht für eine Steuerung der Approximation. Anhand der Graphen aus Abbildung 13 und 14 kann man die Spannweite der einzelnen Buckets ablesen.
Wenn die Ausführungsqueue nun kurz davor steht leer zu werden, wird der Scheduler aktiviert. Dieser führt dann eine Bucket-Zuweisung aus, indem er alle Operatoren durchläuft und diese passende Buckets (bezüglich Dienstgüte) einordnet. Das Durchgehen geschieht in der Zeit O(n), die Zuweisung geschieht in O(1), da die Buckets ja vorher, d.h. nicht zur Laufzeit, schon generiert wurden und mit den benutzerabhängigen Einteilungen ausgestattet sind.

## 2.3.3 p-Tupel-Algorithmus und Bucketing im Vergleich

Diagramm 3 zeigt die Ergebnisse eines Vergleichs des p-Tupel- und des Bucketing-Verfahrens. Für die Messungen wurden 200 Anfragen mit jeweils fünf Operatoren gestellt.. Die Anzahl der Buckets ist an die Anzahl der Partitionen gebunden, d.h. 10 Partitionen sind 100 Buckets. In Abbildung 12 handelt es sich um eine Einteilung in vier Partitionen. Partitionen sind damit ein Maß für die Anzahl an Buckets.
Das p-Tupel-Verfahren hat über den ganzen Versuch über einen QoS-Wert von 0,796, da dieser Algorithmus exakt arbeitet und in dem Experiment nur die Parameter des Bucketing verändert werden und somit dessen Approximationsgrad. Bei dem Bucketing-Algorithmus wird zu Beginn mit nur einem Bucket ein sehr schlechter QoS-Wert erreicht, nämlich 0,427. Dies liegt daran, dass alle Operatoren einzeln gruppiert werden und somit das Verfahren identisch zu Round Robin verläuft. Bei einer Vergrößerung der Anzahl von Partitionen auf 20, d.h 200 Buckets, liegt der QoS-Wert bei 0,85 und somit 7% über dem Ergebnis vom p-Tupel-Algorithmus. Die Begründung hierfür ist, dass der Mehraufwand für das Scheduling mit Sortierung der p-Tupel stärkere Auswirkungen auf den QoS-Wert hat wie der Verlust an Präzision durch die Buckets. Erhöht man nun die Anzahl der Partitionen weiter, so wächst damit natürlich auch der Aufwand für das Scheduling mit Buckets, was man in Diagramm 3 deutlich erkennen kann. Bei einer Erhöhung auf etwa 260 Partitionen ist der Mehraufwand so groß, dass der QoS-Wert für das Bucketing unter den Wert des p-Tupel-Algorithmus fällt.

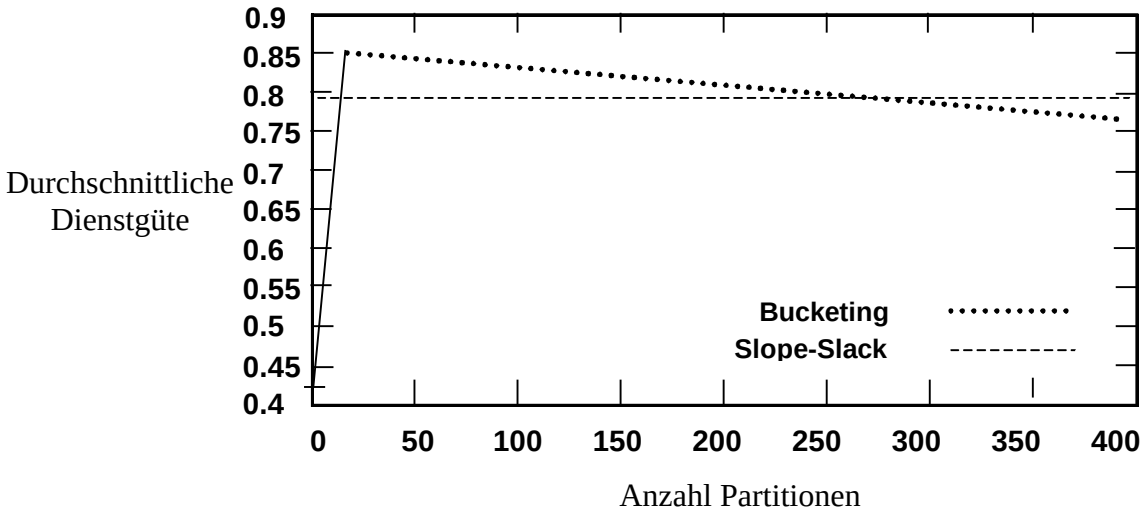


**Diagramm 3: Bucketing Effekte bei QoS**

## 2.3.4 Bewertungen bezüglich Scheduling in AURORA

Wie man an den oben beschriebenen Ergebnissen der Experimente sieht, hat der Mehraufwand, hervorgerufen durch Scheduling-Aufrufe, sehr große Auswirkungen auf die Dienstgüte in Systemen. Man hat gesehen, dass eine Verarbeitung eines jeden Tupels auf jedem Operator im Anfragegraph das System im Extremfall fast zum Stillstand zwingen kann. Techniken wie Train-Scheduling und Superbox-Scheduling zeigen, dass man durch Näherungsverfahren den Scheduling-Overhead reduzieren kann, ohne durch den resultierenden Verlust an Genauigkeit

zu große Einbußen bezüglich der Qualität des Schedulings zu erfahren. Die so für die Datenverarbeitung frei gewordenen Systemressourcen führen sogar zu einer Verbesserung der insgesamt erzielbaren Dienstgüte. Um diese Algorithmen jedoch sinnvoll in einem System einzusetzen sind sogenannte Regler-Attribute erforderlich, um bei Änderung des Traffics darauf adäquat reagieren zu können. Diese Algorithmen sind in AURORA eingesetzt worden, um QoS-Anforderungen von bestimmten Applikationen gerecht zu werden. In diesem Kontext wurde auch noch das Bucketing eingeführt, um auf Kosten der Qualität den Mehraufwand des Schedulers zu minimieren.
Zukünftige Projekte werden sich darauf konzentrieren die Parameter dynamisch, also zur Laufzeit, im Kontext der gerade herrschenden Last und der zur Verfügung stehenden Ressourcen selbst einzustellen.

# 3. Fazit und Ausblick

In diesem Seminar wurden verschiedene DSMS und deren Umgang mit Scheduling und Speicherverwaltung vorgestellt. Es wurde z.B. gezeigt, dass Chain Scheduling sehr nahe an die optimale Strategie zur Speicherminimierung rankommt. Problematisch daran ist jedoch die lange nicht vorhersehbare Antwortzeit. Es wurden darauf hin Modifikationen vorgestellt, die versuchen einen ausgewogenen Mittelweg zu wählen. Die Annahmen, welche bei den Versuchen vorrausgesetzt wurden, sind in den meisten Systemen jedoch nicht gegeben. Deswegen existieren durchaus Unterschiede zwischen den hier vorgestellten Ergebnissen und denen die in der Realität auftreten.
Allgemein wurde in diesem Seminar festgestellt, dass es noch kein System gibt, welches optimale Ergebnisse für alle Arten von Applikationen liefert. Was das Scheduling so schwierig gestaltet sind die unvorhersehbaren Datenraten, die bei statischen Strategien schnell zu sehr schlechten Werten der Dienstgüte führen. Deswegen wurden Approximationstechniken eingeführt, welche in Zeiten von sehr hohem Datenaufkommen versuchen die Last, die auf dem System existiert auf Kosten der Genauigkeit zu reduzieren, ohne jedoch extrem große Einbrüche der Qualität hinzunehmen.
Aktuelle Forschungsarbeiten beschäftigen sich mit der Generierung von sogenannten Managern, deren Aufgabe es ist die Systemressourcen dynamisch so aufzuteilen, dass die Anforderungen an die Dienstgüte bestmöglich erfüllt sind. Diese Aufgabe stellt sehr hohe Anforderungen an das zu entwerfende Scheduling-Verfahren und ist bis zum jetzigen Zeitpunkt noch nicht optimal gelöst.

# 4. Quellenverzeichnis


[1] B. Babcock, S. Babu, M. Datar, R. Motawani, and J. Widom, "*Models and issues in data stream systems"*, ACM PODS 2002 (tutorial).

[2] Golab & Ozsu, *Issues in Data Stream Management*. ACM SIGMOD Record, June2003.

[3] D. Carney, U. Çetintemel, A. Rasin, S. Zdonik, M. Cherniack, M. Stonebraker. *Operator Scheduling in a Data Stream Manager*. In proceedings of the *29th International Conference on Very Large Data Bases* (*VLDB*'03), Berlin, Germany, September 2003.

[4] N. Tatbul, U. Çetintemel, S. Zdonik, M. Cherniack, M. Stonebraker. *Load Shedding in a Data Stream Manager***.** In proceedings of the *29th International Conference on Very Large Data Bases* (*VLDB*'03), Berlin, Germany, September 2003, pp. 309-320.

[5] D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik. *Monitoring Streams: A New Class of Data Management Applications.* In proceedings of the *28th International Conference on Very Large Data Bases* (*VLDB*'02), Hong Kong, China, August 2002.

[6] The STREAM Group. *STREAM: The Stanford Stream Data Manager* (short overview paper), *IEEE Data Engineering Bulletin, Vol. 26 No. 1, March 2003*

[7] Motwani et al. (Stanford group), *Query Processing, Resource Management, and Approximation and in a Data Stream Management System*. CIDR 2003.

[8] Kang, J., Naughton, J.F., Viglas, S.D., *Evaluating Window Joins Over Unbounded Streams***.** Evaluating Window Joins over Unbounded Streams, ICDE 2003.

[9] Arasu et al. *Characterizing Memory Requirements for Queries over Continuous Data Streams*. PODS 2002.

[10] Michael Cammert, Christoph Heinz, Jürgen Krämer, Alexander Markowetz, Bernhard Seeger. *PIPES: A Multi-Threaded Publish-Subscribe Architecture for Continous Queries over Streaming Data Sources* 2004.

[11] Brian Babcock, Shivath Babu, Mayur Datar, Rajeev Motwani. *Chain: Operator Scheduling for Memory Minimization in Data Stream Systems* 2004.

[12] Brian Babcock, Mayur Datar, Rajeev Motwani, *Sampling from a Moving Window Over Streaming Data***.**

[13] Terry, D.B., Goldberg, D., Nichols, D. und Oki, B.M. (1992). *Continuous Queries over Append-Only Databases***.** SIGMOD Conference: 321-330.

[14] Brian Babcock, Ahivnath Babu, Mayur Datar. *Operator Scheduling in Data stream systems*