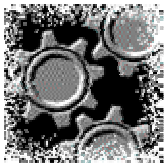
Quality of Service

Seminar: Data Streams



16.06.2005

Bearbeiter: Christian Stamber

Betreuer: Prof. Dr. Ing. Stefan Deßloch

Inhaltsverzeichnis

1 Einleitung & Motivation

Unter einem *Data Stream* (zu Deutsch Datenstrom) kann man einen Strom von Datenelementen verstehen, welcher (mglw. zeitlich ungleichmäßig) permanent neue Daten liefert, und in seiner Länge damit potentiell unbegrenzt ist [BBD+02]. Ursprung dieser Daten sind z.B. Sensoren, Systeme zur Analyse von Aktienverläufen an der Börse oder Anwendungen die die Position von Objekten überwachen (vgl. LKW-Maut). Ein Datennstrom kann damit zum Zweck der Modellbildung auch als *append-only relation* betrachtet werden, also eine Relation an die man nur anfügen kann.

Um die Daten solcher Data Streams zu verwalten und Anfragen darauf auszuwerten, kommen so genannte *Data Stream Management Systems* (DSMS) zum Einsatz. Im Vergleich zu gewöhnlichen Datenbankmanagementsystemen (DBMS), welche üblicherweise für die Speicherung und Informationsauswertung von Geschäftsdaten zuständig sind, ergeben sich einige wesentliche Unterschiede:

- Eine wichtige Besonderheit der DSMS ist beispielsweise der zeitliche Faktor, der viel mehr im Vordergrund steht. Oftmals werden diese Systeme sogar im Umfeld von Echtzeitanwendungen eingesetzt. Der Grund dafür ist, dass man in DSMS Deadlines für die Anfragebearbeitung setzen kann.
- In DBMS werden die neu eingegebenen Daten dauerhaft gespeichert (*persistente* Tupel) und erst im Anschluss bearbeitet bzw. ausgewertet. In DSMS ist eine permanente Speicherung aller Daten, aufgrund der potentiell unendlichen Datenmenge und der zeitlichen Beschränkungen, nicht möglich. Anfragen werden deshalb auf *temporären* Tupeln ausgeführt, welche nach ihrer Verarbeitung gelöscht bzw. archiviert werden, sofern man sie nicht weiter benötigt.
- DSMS unterstützen das Konzept der *kontinuierlichen Anfrage*. Dabei handelt es sich um Anfragen, die permanent die neu ankommenden Tupel aus den Eingabeströmen auswerten und ständig neue Ergebnisse, wieder in Form von Streams, liefern. Ein ähnliches Konzept existiert in klassischen DBMS nicht; hier werden Anfragen üblicherweise einmalig (bzw. mehrmals getrennt hintereinander) ausgewertet und liefern jeweils nur einem Ergebnis zurück.
- Der Fokus von DBMS liegt auf der Berechnung von korrekten und vollständigen Anfrageergebnissen. In Data-Streams-Anwendungen können zumeist keine exakten Antworten auf die Anfragen berechnet werden. Das ist dadurch begründet, dass das System – im Gegensatz zu DBMS – keinen Zugriff auf den gesamten Datenbestand hat, da die Daten erst nach und nach ankommen und ältere Daten nicht unbegrenzt gespeichert werden können.
- Die angeknüpften Datenströme eines DSMS sind ständig quantitativen Fluktuationen unterworfen, d.h. es kommt immer wieder vor, dass in einem kurzen Zeitabschnitt sehr viele Daten auf einmal im System ankommen. Somit kann es sein, dass die vorhandenen Ressourcen nicht ausreichen, um die zeitlichen Erwartungen (zum Teil Echtzeitanforderungen), die an das System gestellt werden, zu erfüllen. Deshalb sind Mechanismen erforderlich, die die Last reduzieren, indem sie etwa einen Teil der ankommenden Tupel verwerfen.
- Auch die Dateneingabe erfolgt auf anderem Weg: in DBMS werden die Daten meist von Menschen eingegeben, während in DSMS die Daten von Sensoren und anderen externen Quellen generiert werden. Eine zentrale Aufgabe dieser Systeme ist, den Benutzer über Auffälligkeiten zu informieren, d.h. in bestimmten Situationen Alarm auszulösen. Man nennt sie deshalb auch DBMS-Active, Human-Passiv (DAHP), wohingegen man traditionelle DBMS als Human-Active, DBMS-Passiv bezeichnet (HADP) [ACC+03b].

Im Unterschied zu klassischen DBMS, bei denen einzig und allein die Korrektheit der gestellten Anfrageergebnisse im Vordergrund steht, spielt bei DSMS vor allem die rechtzeitige Antwort auf eine Anfrage eine entscheidende Rolle. Oft gelten für die Anwendungen sogar Realzeitanforderungen und sie verlangen vom System, dass die Ergebnistupel mit einer möglichst minimalen Verzögerung ankommen. Ferner sind oftmals exakte Anfrageergebnisse aus unterschiedlichen Gründen nicht möglich, so dass eine Approximation der Ergebnisse von Nöten, und für viele Anwendungen auch durchaus akzeptabel ist. Die konkreten Anforderungen an die maximal zulässige Verzögerung und die Genauigkeit der Antwort sind aber von Anwendung zu Anwendung unterschiedlich. Deshalb ist es notwendig, Anforderungen bezüglich der Dienstgüte (Quality of Service, QoS) dem DSMS bekannt zu machen, damit diese eingehalten bzw. ausgenutzt werden können.

Wie schon erwähnt, fließt in Datenströmen kontinuierlich eine unbegrenzte, zeitlich schwankende Menge von Daten. Ein DSMS ist oft mit mehreren tausend Strömen verknüpft, die möglicherweise auch noch hohe Datenraten besitzen. Des Weiteren greifen unter Umständen sehr viele Anwendungen zur selben Zeit auf das System zu und so kann es leicht vorkommen, dass mehrere hundert Anfragen gleichzeitig vom DSMS bearbeitet werden und so um die Ressourcen des Systems konkurrieren. Aus all diesen Gründen kommt es vor, dass die zur Verfügung stehenden Ressourcen nicht ausreichen, um die Anforderungen der Benutzer zu erfüllen. Deshalb ist es unentbehrlich, dass das System die Anforderungen des Anwenders bezüglich der Dienstgüte kennt, um intelligent die vorhandenen Ressourcen zuzuordnen und Operator Scheduling zu realisieren. Ein weiteres Problem sind die so genannten *bursts*, also überhöhte Ankunftsraten in den Datenströmen, bei denen innerhalb eines kurzen Zeitraums sehr viele Daten auf einmal ankommen. Diese Situationen machen es notwendig Mechanismen zu benutzen, die die Systemlast verringern und dabei einen möglichst minimalen Kompromiss bei den Anwendungsanforderungen eingehen (*graceful degradation / load shedding*; dazu später mehr).

Es sind also zwingend Maßnahmen erforderlich, mit denen man Dienstgüteanforderungen spezifizieren und sicherstellen kann. Die Anforderungen an das System übergibt der Anwender im Vorfeld. Die Sicherstellung dieser Anforderungen geschieht zur Laufzeit in Form von Leistungsanpassungen. Quality of Service hat nun im Wesentlichen drei verschiedene Aufgaben:

1. Spezifikation der QoS-Anforderungen durch den Benutzer
2. Überwachung der vom System erreichten Dienstgüte durch Vergleich des Ist-Zustands (akt. Systemmerkmale bzgl. QoS ermitteln) mit den Anforderungen
3. Berücksichtigung der QoS-Anforderungen zur Durchführung von Systemaufgaben (z.B. Zuteilung von Ressourcen, Lastverminderung)

In dieser Ausarbeitung soll das Konzept von Quality of Service im Bereich DSMS genauer untersucht werden. Die wesentlichen Aspekte werden dabei anhand des DSMS Aurora erläutert. Bei der Entwicklung von Aurora wurde besondere Aufmerksamkeit auf Quality of Service - Elemente gelegt und somit eine Vorreiterstellung in diesem Bereich übernommen.

Dazu erfolgt zunächst ein kurzer Überblick von Aurora, in dem auf die Sprachmittel und Anfragetypen eingegangen wird. Danach werden in Kapitel 3 die einzelnen Dienstgütekriterien vorgestellt, anhand derer man die Anforderungen an das System spezifizieren, messen und steuern kann. In Kapitel 4 wird die zugrunde liegende Architektur von Aurora in Bezug auf Quality of Service untersucht. In den nächsten zwei Kapiteln (5 und 6) werden dann die beiden wichtigsten QoS-Elemente (Scheduling und Lastreduktion) erklärt. Im Anschluss betrachten wir einige weitere Ansätze, die die Dienstgüte steuern und optimieren.

2 Das DSMS Aurora

Aurora [ACC+03a, ACC+03b, CCC+02, BBC+04] wurde in Gemeinschaftsarbeit des MIT, der Brandeis Universität und der Brown Universität entwickelt. Dieses DSMS konzentriert sich auf die Bearbeitung von Datenströmen, im Gegensatz z.B. zum *STREAM* Projekt der Stanford Universität [ABB+03, BBD+02], bei dem die Integration von Data Streams in klassische Datenbanken im Vordergrund stand. Mittlerweile wurde die Forschung am Aurora Projekt eingestellt und der Nachfolger *Borealis* ins Leben gerufen. Aurora existiert jedoch weiterhin in kommerzieller Form.

Die Einsatzgebiete von DSMS sind vielfältig, die zentrale Aufgabe ist aber zumeist die Überwachung von bestimmten Datenkonstellationen. So wurde beispielsweise mit Hilfe des DSMS Aurora ein Kontrollsystem für die Überwachung der Wasserqualität erstellt oder aber ein militärisches System zur Positionsbestimmung von feindlichen Einheiten entlang der Frontlinie. Weiterhin wurden Programme entwickelt, die mittels Aurora Wertpapierinformationen auswerten und wenn nötig den Broker alarmieren.

2.1 Sprachmittel in Aurora

Anfragen in Aurora werden anhand des so genannten „Box and Arrow“ - Paradigmas erstellt, was auch bei vielen Workflowsystemen zum Einsatz kommt. Die Boxen stellen dabei die einzelnen Operatoren in der Anfrage dar. Diese werden anschließend mit Hilfe von Pfeilen miteinander verbunden, um so den Datenfluss im Anfragegraphen zu modellieren. Über diese Pfeile „fließen“ zur Laufzeit die Tupel der Eingabeströme. Die Formulierung einer Anfrage geschieht für den Benutzer komfortabel in

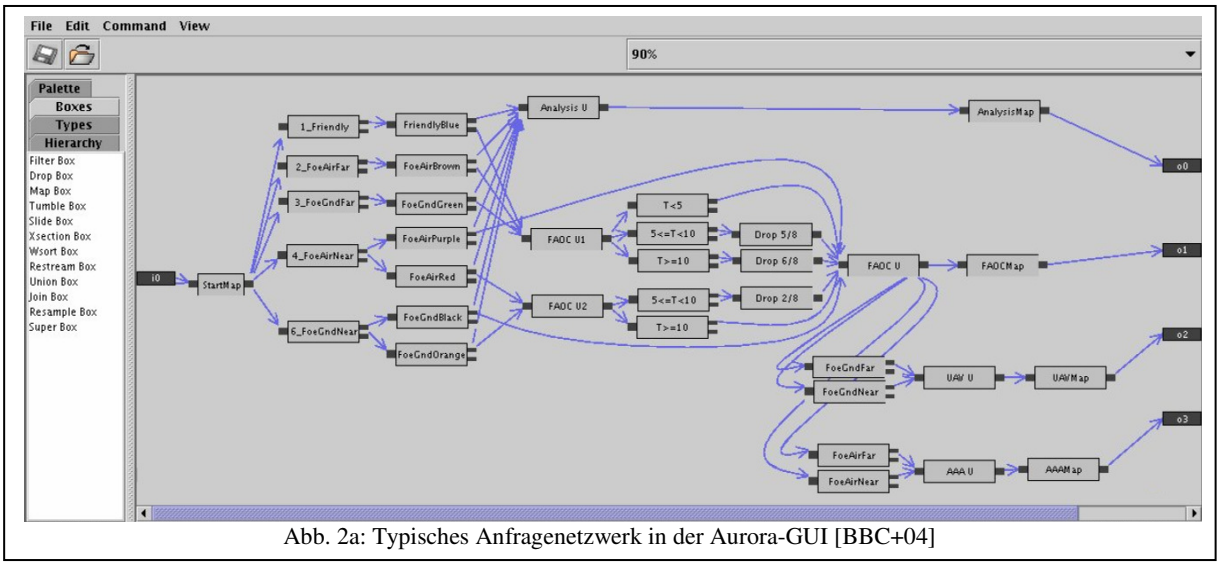


Abb. 2a: Typisches Anfragenetzwerk in der Aurora-GUI [BBC+04]

einer graphischen Oberfläche (vgl. Abb. 2a), in der er die einzelnen Boxen und Pfeile mit der Maus platzieren kann [ACC+03a, BBC+04]. Diese imperative Art der Anfragekonstruktion unterscheidet sich von anderen DSMS, die dazu eine deklarative Sprache (vgl. [BBD+02]), ähnlich zu SQL, benutzen.

Insgesamt existieren in Aurora neun verschiedene Anfrageoperatoren (nach [BBC+04]; [ACC+03b] nennt nur die ersten sieben): Filter, BSort, Map, Aggregate, Union, Join, Resample, Read und Update. Der Ursprung dieser Boxen/Operatoren beruht teilweise auf den Operatoren der relationalen Algebra; die so gewonnene Algebra wird als SQuAl (Stream Query Algebra) bezeichnet. Man unterscheidet hier zwischen Operatoren, die man als *order-agnostic* (reihenfolgeunabhängig) bezeichnet und Operatoren, die *order-sensitive* (reihenfolgeabhängig) genannt werden:

- Reihenfolgeunabhängige Operatoren (Filter, Map und Union) benötigen keine Zusatzinformation über die Reihenfolge der Eingabetupel und können direkt auf jedem einzelnen Tupel ausgewertet werden.
- Reihenfolgeabhängige Operatoren (BSort, Aggregate, Join, Resample) müssen von einer bestimmten Reihenfolge der Eingabetupel ausgehen. Ohne diese Informationen ist es nicht möglich, diese Operatoren innerhalb einer endlichen Zeit und mit endlichem Speicherplatzbedarf auszuführen [ACC+03b]. Typisch für diese Operatoren ist, dass sie erst eine gewisse Anzahl an Eingabetupel ansammeln müssen, um ein Ergebnis liefern zu können.
  Soll beispielsweise mit Hilfe des Aggregate-Operators der Durchschnittswert einer Aktie über die letzten 10 Stunden berechnet werden, so setzt dieser z.B. voraus, dass die Aktienwerte in der richtigen zeitlichen Reihenfolge eintreffen. Wäre dem nicht so, d.h. die Tupel mit den Aktienwerten kämen wahllos mit variabler Zeitverzögerung und ohne bestimmte Reihenfolge an, so könnte der Operator zu keiner Zeit davon ausgehen, dass alle Tupel dieses Zeitraums angekommen sind. Der Durchschnittswert könnte also nicht gebildet werden, da eventuell entscheidende Werte fehlen.
  Bei Aggregate kommt das so genannte *sliding window* - Konzept zum tragen. Dabei werden immer alle Tupel betrachtet, die (nach obigem Beispiel) innerhalb der letzten 10 Stunden angekommen sind und so ständig der Durchschnitt gebildet. Das Zeitfenster verschiebt sich dabei mit und enthält so immer die aktuellsten Daten. Oftmals ist es aber auch nötig, dieses Konzept auf approximative Weise anzuwenden, etwa wenn es unmöglich ist alle Tupel zu betrachten. So muss z.B. bei Aggregatfunktionen oder Joins ein Zeitfenster benutzt werden.

Dieses Kapitel soll nur als Übersicht über die Anfragesprache dienen und ist nicht Schwerpunkt dieser Arbeit. Für weitere Details sei an dieser Stelle auf die Ausarbeitung mit dem Thema „Anfragesprachen“ verwiesen.

2.2 Anfragen in Aurora

Eine typische Anfrage in Aurora besteht aus einem oder mehreren Eingabeströmen, einem Netzwerk aus Operatoren und aus Ausgabeströmen. Weiter ist jeder Ausgabestrom mit einer Applikation verbunden (Ausnahme Views, siehe unten). Es existieren insgesamt drei verschiedene Anfragetypen:

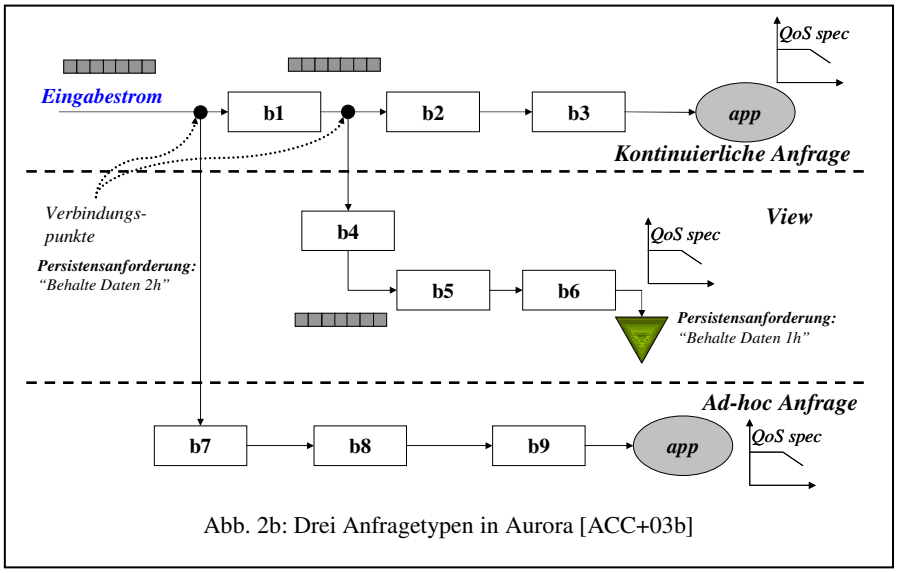


Abb. 2b: Drei Anfragetypen in Aurora [ACC+03b]

● *Kontinuierliche Anfragen* werden vor der eigentlichen Laufzeit in das System eingegeben und ständig abgearbeitet. Dabei liefern solche Anfragen nicht ein einziges Ergebnis, sondern werden fortlaufend ausgeführt und produzieren so neue Ausgabetupel aufgrund der kontinuierlich durch die Eingabeströme bereitgestellten Daten. Das Ergebnis einer kontinuierlichen Anfrage stellt damit selbst wieder einen Datenstrom dar [ACC+03b, CCC+02].

● *Ad-Hoc Anfragen* werden zur Laufzeit vom Benutzer an Aurora gestellt. Entscheidend für diesen Anfragetyp sind die Verbindungspunkte (*connection points,* vgl. Abb. 2b) im Anfragenetzwerk. An diesen vorher festgelegten Punkten können zur Laufzeit neue Boxen hinzugefügt werden, um so das Anfragenetz zu erweitern. Dabei hat jeder Punkt eine im Vorfeld eingestellte Persistenzgrenze. Es wird also festgelegt, wie viele Tupel und wie lange sich Aurora diese Tupel an diesen Punkten merken soll. Dabei kann man zeitliche Grenzen setzen, z.B. könnte man festlegen, dass alle Tupel der letzten 5 Stunden festgehalten werden sollen. Ebenso ist es auch möglich, dass man immer eine feste Anzahl an Tupeln festhält, etwa immer die letzten 50 Tupel die den Punkt passieren [ACC+03b, CCC+02]. Die festgehaltenen Daten können dann von den neu angegliederten Ad-Hoc Anfragen benutzt werden, um historische Daten in die Anfrageberechnung zu integrieren. Wenn zum Beispiel eine Ad-Hoc Anfrage den Durchschnittswert einer Aktie über die letzten *2h* errechnen soll, so könnte sie im Netzwerk aus Abb. 2b sofort nach ihrer Anbindung an einen der beiden Verbindungspunkte Ergebnisse liefern. Wären die Daten nicht gespeichert worden, so müsste die Anfrage *2h* warten, bis genügend Daten zusammengetragen wurden.

● Die letzte Anfrageart sind die *Views* (Sichten). Diese sind im Gegensatz zu den anderen Anfragetypen nicht direkt mit einem Anwendungsprogramm verknüpft, vielmehr werden die Ergebnisse dieser Anfrage für eine bestimmte Zeit gespeichert und können von verschiedenen Programmen bei Bedarf abgerufen werden [ACC+03b, CCC+02].

Der prinzipielle Ablauf zur Berechnung von Anfrageergebnissen ist nun so: Die Tupel erreichen über den Eingabestrom das Anfragenetzwerk und werden dann zu den Operatoren weitergeleitet, welche die Tupel auswerten. Das Antworttupel eines Operators erreicht dann über das Netzwerk den nächsten Operator usw. bis ein Ausgabestrom erreicht ist. Weitere Details sind auch hier wieder im Seminar „Anfragesprachen“ zu finden.

3 Verschiedene QoS-Kriterien

Da für jedes Anwendungsprogramm und im Extremfall auch für jeden einzelnen Benutzer andere Anforderungen an das DSMS gelten, ist es nötig Kriterien festzulegen anhand derer die Dienstgüte spezifiziert und überwacht werden kann. In den hier betrachteten Ansätzen werden im Wesentlichen vier grundlegende QoS-Kriterien unterschieden:

- **Antwortzeit** (*Response times* [ACC+03b], *Tuple delays* [TuXP04])
- **Datenverlustrate** (*Tuple drops* [ACC+03b], *Data loss ratio* [TuXP04])
- **Attributwerte** (*Values produced* [ACC+03b])
- **Update-Rate** (*Updating frequency* [TuXP04])

Natürlich ist es auch mögliche andere Kriterien als die vier genannten zu verwenden. So ist z.B. auch der aktuelle Durchsatz ein möglicher Maßstab [ACC+03b]. Auf diese wird jedoch in den betrachteten Systemen aus [ACC+03b] und [TuXP04] nicht näher eingegangen. In Aurora [ACC+03b] können jedoch beispielsweise auch Kombinationen – wie etwa gewichtete Funktionen – aus den genannten Kriterien als QoS-Kriterium angewendet werden.

In den nachfolgenden Textabschnitten und Abbildungen werden die Begriffe Dienstgüte, Qualität und Nutzen, sowie die Abkürzung QoS synonym verwendet.

3.1 Antwortzeit

Jedes Tupel, das über einen Eingabestrom das Anfragenetzwerk betritt, wird mit einem Zeitstempel versehen. Gelangt das Tupel an einen Operator und wird (mglw. zusammen mit anderen Tupeln) verarbeitet, so wird das Ergebnistupel mit dem Zeitstempel des ältesten an der Operation beteiligten Eingabetupels versehen. Wenn das Tupel einen Ausgabestrom erreicht, wird der aktuelle Zeitstempel mit dem Zeitstempel des Tupels verglichen. Auf diese Weise ist es möglich, eine Aussage über die Dauer zu machen, die ein Tupel von der Eingabe in das Anfragenetzwerk bis zur Ausgabe über einen Ausgabestrom braucht. Dieser Wert wird als *Antwortzeit* bezeichnet. Diese Antwortzeit ist je nach Systemlast unterschiedlich lang, da es (zusätzlich zu den Bearbeitungszeiten durch die Operatoren) aufgrund der begrenzten Ressourcen des Systems (speziell Prozessorzeit) zu Wartezeiten kommt und Tupel in den Queues der betreffenden Operatoren temporär zwischengelagert werden müssen [ACC+03b, TuXP04].

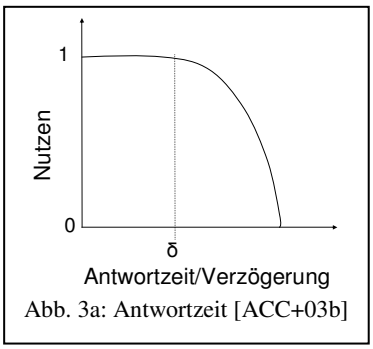


Abb. 3a: Antwortzeit [ACC+03b]

Deshalb stellt die Antwortzeit einen wichtigen Dienstgüte-Aspekt dar. Jedes Eingabetupel, welches in das Anfragenetzwerk gelangt, sollte innerhalb einer akzeptablen Zeitspanne bearbeitet werden. Ist dies nicht der Fall, so kommt es häufig vor, dass die auf eine Anfrage generierten Ergebnisse an Aussagekraft bzw. an Nutzen verlieren, da die verwendeten Daten nicht mehr aktuell sind. Das ist besonders bei zeitkritischen Anwendungen von Belang, wie z.B. bei der Überwachung von Aktienkursen oder bei Navigationssystemen. Es ist nicht akzeptabel, dass die aktuelle Positionsangabe erst kommt, wenn man schon an der gewünschten Autobahnausfahrt vorbeigefahren ist. Aber vor allem im Bereich von Echtzeitanwendungen wird dieses Kriterium als Wichtigstes angesehen. Natürlich haben nicht alle Anwendungen so hohe Anforderungen. Abb. 3a zeigt eine typische QoS Kurve. Das Intervall [0; δ] stellt den optimalen Bereich dar, während sich nach δ der Nutzen rapide verschlechtert. Das DSMS wird also im laufenden Betrieb versuchen, die Dienstgüte in dieser Region anzusiedeln [ACC+03b, TuXP04].

3.2 Datenverlustrate

Zur Bewältigung von großen Datenmengen, die innerhalb kurzer Zeit eintreffen, kann es nötig sein manche Tupel zu verwerfen bzw. nicht zu berücksichtigen, da sonst die Prozessorbelastung zu hoch wird und z.B. das Kriterium der Antwortzeit nicht mehr im optimalen Bereich gehalten werden kann. Je mehr Daten verworfen werden müssen, desto schlechter wird die Dienstgüte der Datenverlustrate. Die Antwortzeit steht oftmals in einem umgekehrt proportionalen Verhältnis zu der Datenverlustrate, d.h. je mehr Tupel in einer Überlastungssituation bearbeitet werden, desto schlechter wird die Antwortzeit und desto besser/niedriger wird die Datenverlustrate und umgekehrt.

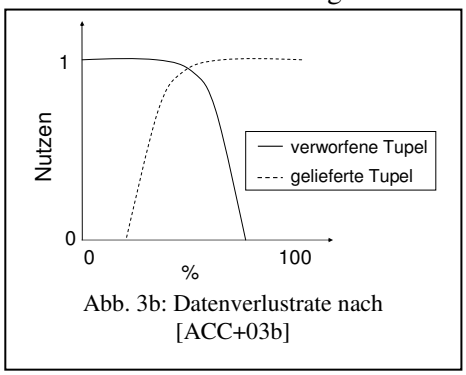


Abb. 3b: Datenverlustrate nach [ACC+03b]

Im Beispielgraph erkennt man einen Plateaubereich bei maximalem Nutzen im Bereich von 0 bis ca. 50 % verworfene Tupel, darüber verschlechtert sich die Dienstgüte schnell und erreicht bei 70-80% den Wert 0. Das DSMS wird also versuchen, nicht mehr als etwa 50% der ankommenden Tupel fallen zu lassen. Manche DSM-Systeme verlangen, im Gegensatz zu Aurora, die Eingabe von monoton steigenden Graphen (z.B. System aus Kapitel 7.1). Dazu trägt man schlicht anstatt des relativen Anteils der verworfenen Tupel, den prozentualen Anteil aller gelieferten Tupel auf der x-Achse ein (siehe Abb. 3b) [ACC+03b, TuXP04].

3.3 Attributwerte

Damit ist nicht etwa gemeint, dass das DSMS Einfluss auf den Wert bestimmter Tupelattribute nimmt, sondern dass Werte von Tupel, die innerhalb einer bestimmten Grenze liegen mit hoher Priorität behandelt werden [ACC+03b]. Diese werden dann in einer Überlastsituation nicht oder erst später verworfen. Dies ist vor allem bei Anwendungen wichtig, bei denen bestimmte Werte ein bestimmtes Ereignis auslösen sollen. Vorstellbar z.B. in einem Kernkraftwerk, bei dem kontinuierlich der Zustand des Reaktors abgefragt wird und eine zu hohe Temperatur die Abschaltung des Kerns zur Folge haben soll. Dabei ist natürlich sehr wichtig, dass die Tupel, die eine zu hohe Temperatur enthalten, nicht verworfen werden dürfen. Man sieht in Abb. 3c dass der Nutzen/die Wichtigkeit der Tupel ab einer bestimmten Temperatur schlagartig auf höchstes Niveau ansteigt. Das bedeutet, dass diese Tupel bei einer Überlastung des DSMS nicht verworfen werden dürfen. Es sind aber auch ganz andere Graphen vorstellbar.

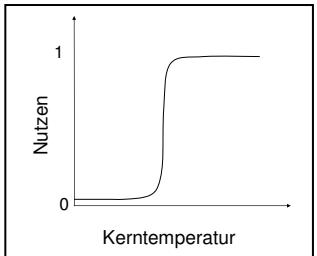


Abb. 3c: [ACC+03b]

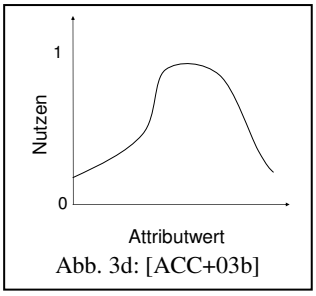


Abb. 3d: [ACC+03b]

3.4 Update-Rate

Dieses Kriterium beschreibt, wie oft neue Tupel in die Berechnung von Anfragen einfließen. Man unterscheidet prinzipiell zwei Arten.

- **Zeitintervalle:** Nach einer bestimmten Zeitspanne kommt ein neues Tupel, z.B. ein neues Tupel alle 2 Sekunden. Diese Option ist vor allem für Anwendungen interessant, die auf kontinuierliche Informationsversorgung angewiesen sind, z.B. Positionsbestimmung via GPS oder aber auch Videoübertragungen. Es gibt in diesen Anwendungen häufig keine besonders bedeutenden Werte, vielmehr sind alle Tupel gleich wichtig.

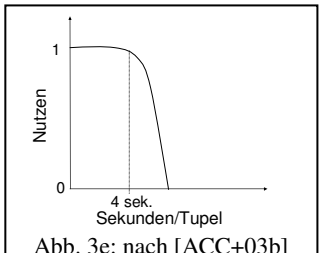


Abb. 3e: nach [ACC+03b]

- **Wertintervalle:** Nachdem sich ein bestimmter Wert um einen gewissen Prozentsatz geändert hat, wird ein neues Tupel erzeugt. Bei diesem Verfahren kann es zu Burst-Situationen kommen, d.h. es kommen sehr viele Tupel innerhalb kürzester Zeit in das DSMS. Dieses Kriterium ist für Anwendungen wichtig, bei denen bestimmte Tupelwerte eine besonders hohe Wichtigkeit besitzen, wie etwa bei dem erwähnten Kernkraftwerk aber auch z.B. bei der medizinischen Überwachung eines Patienten. Dabei ist z.B. wichtig, dass bei einer zu hohen Herzfrequenz möglichst viele Werte an den Arzt übermittelt werden; ist der Herzschlag jedoch normal, so ist eine lückenlose Aufzeichnung nicht von Nöten.

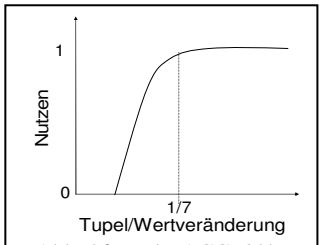


Abb. 3f: nach [ACC+03b]

Dieses Kriterium kann auf zweierlei Art betrachtet werden. Zum einen ist es möglich, wie in [TuXP04], dass die Update-Rate mit den Eingabeströmen auf regulierende Art und Weise verbunden ist. D.h. man benutzt die Update-Rate als Stellschraube, um die Systemlast zu regulieren. Dieser Vorgang geschieht entweder, indem den angeschlossenen Sensoren mitgeteilt wird, weniger Daten zu produzieren oder, indem einfach solange Tupel aus den Eingabeströmen gefiltert werden bis die Vorgabe erreicht wurde.

Andererseits ist es möglich die Update-Rate, wie bei den anderen erwähnten Kriterien, als Kriterium mit reiner Beobachterfunktion anzusehen. Dabei geht man davon aus, dass das DSMS keinen Einfluss darauf hat, wann die angeschlossenen Ströme Daten produzieren sollen. Jedoch kann man mit Hilfe dieses Kriteriums dann sicherstellen, dass nicht aus einem Strom zu lange keine Tupel mehr berücksichtigt werden. Wurde z.B. in den letzten 4 Sekunden kein Tupel mehr bearbeitet, so wird das DSMS versuchen dies in nächster Zeit zu tun, da sich ansonsten die Dienstgüte zu sehr verschlechtert (vgl. Abb. 3e). In Abb. 3f ist es beispielsweise so, dass bei einer Wertänderung des beobachteten Attributes um 7% zumindest ein neues Tupel bearbeitet werden muss.

Auch bei diesem Kriterium kann es nötig sein –je nach verwendetem DSMS– entweder einen monoton fallenden (Abb. 3e) oder einen monoton steigenden Graphen einzugeben (diese lassen sich aber durch Kehrwertbildung der x-Werte problemlos ineinander überführen).

3.5 Festlegen der QoS-Anforderungen

Die gängigste Methode, QoS-Anforderungen für das jeweilige Kriterium zu beschreiben, ist die Angabe von zweidimensionalen Graphen ([ACC+03b, CCC+02]), wie sie hier auch zur Illustration in den Abbildungen verwendet werden. In Aurora etwa, kann der Administrator dem System solch einen Graphen für die Antwortzeit übergeben (pro Ausgabestrom) und zusätzlich zwei zweidimensionale Graphen für die Datenverlustrate und die Attributwerte (global, für alle Ausgabeströme geltend). Aus diesen Angaben setzt Aurora einen multidimensionalen Graphen zusammen. Dabei werden die eingegeben Graphen normalisiert, um sie besser vergleichen zu können. Weiterhin ist bei der Eingabe der Kriterien möglicherweise gefordert, dass die Graphen alle in monoton fallender und konvexer Form (Ausnahme: Attributwerte) gestaltet sind. Das ermöglicht die Anwendung von optimierten Algorithmen, wie etwa der des „Gradient Walking“, bei der die Veränderung der Steigung des QoS-Graphen zur Ermittlung der nächsten QoS-Anpassung herangezogen wird. Ist die Steigung der Funktion an der aktuellen Stelle negativ und ist eine Verschiebung nach rechts abzusehen, so wird das System Last verringern, um die Dienstgüte nicht zu verschlechtern (vgl. Abb. 3g bei monoton fallenden Graphen; oft die Standardsituation).

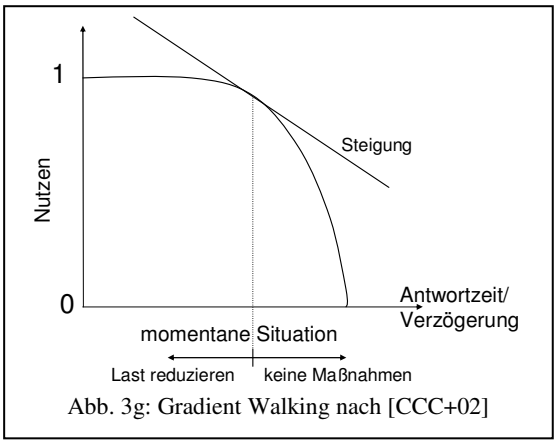


Abb. 3g: Gradient Walking nach [CCC+02]

Wäre die Steigung hingegen positiv, so würde das System mehr Last zulassen, um eine Verschiebung nach rechts zu veranlassen und den Nutzen zu verbessern.

Zusätzliche Möglichkeiten zur Definition der Anforderungen einer Anwendung ergeben sich durch eine Gewichtung der einzelnen QoS-Kriterien. Zwar besteht die Möglichkeit, diese Gewichtung indirekt über die Graphen an sich zu machen, in dem man z.B. wichtigeren Kriterien insgesamt eine höhere Dienstgüte über der gesamten x-Ordinate zuordnet und weniger wichtigen Kriterien eine entsprechend niedrigere Dienstgüte. Diese Vorgehensweise erfordert aber bei Änderung der Anforderung eine komplette Neukonstruktion der Graphen und geht deshalb weniger schnell von der Hand. In der Regel verteilt der Benutzer pro Kriterium ein Gewicht in Form von Koeffizienten. Diese Koeffizienten summieren sich zu 1 auf und bescheren demjenigen Kriterium eine höhere Beachtung, welches das höhere Gewicht hat.

In anderen Systemen, z.B. in [TuXP04] (siehe Kapitel 7.1), wird vom Benutzer verlangt, einen Vektor $\vec{q}=(q_1,q_2,\ldots,q_n)$ zu spezifizieren, welcher die bevorzugten Werte q_i für die n betrachteten Kriterien widerspiegelt. Anhand dieser diskreten Werte wird dann, mittels einem, vom Administrator vorher eingegebenen, Repertoire an Dienstgüte-Funktionen ($u_i() \; 1 \leq i \leq n$, für jedes Kriterium eine Funktion), die individuelle QoS-Funktion des Benutzers erzeugt:

$$U(\vec{q}) = \sum_{i=1}^{n} w_i u_i(q_i)$$

Die w_i sind die Gewichte, die – wie im vorherigen Abschnitt erläutert – die Priorität der einzelnen Kriterien für den Benutzer reflektiert und zusammen mit dem Vektor eingegeben werden.
Diese Vorgehensweise erlaubt eine einfachere Anpassung der QoS-Anforderungen an einzelne Benutzer, da lediglich ein Vektor angepasst werden muss. Der Benutzer gibt hierbei nicht die Dienstgüte an sich an, sondern z.B. die maximale Antwortzeit oder die Anzahl der zulässigen Tupelverluste, die er noch vertreten kann. Das hat den Vorteil, dass der Benutzer nicht mit abstrakten Werten arbeitet (also Werte von 0 bis 1), sondern mit ihm besser vertrauten Angaben.

3.6 QoS-Garantien

Wie bereits erläutert, werden bei der Spezifikation der QoS-Anforderungen gewisse Annahmen gemacht, z.B. in Bezug auf die Datenrate der Eingabeströme oder auf die Selektivität einzelner Operatoren und deren Kosten bei der Abarbeitung. Diese Werte sind jedoch im laufenden Betrieb hochgradig dynamisch und im Vorfeld schlecht abschätzbar. So kommt es in der Praxis häufig zu Datenschüben (bursts), das heißt, dass innerhalb eines kurzen Zeitabschnittes sehr viele Daten auf einmal eintreffen. Des Weiteren ist zu berücksichtigen, dass es bei der Angabe von mehreren QoS-Kriterien häufig zu Wechselwirkungen zwischen denselben kommt. So kann es beispielsweise unmöglich für das System sein sowohl die geforderte Dienstgüte für die Antwortzeit, als auch die verlangte Dienstgüte für die Datenverlustrate einzuhalten, da diese in den meisten Fällen ein antagonistisches Verhältnis zueinander haben. Soll nun z.B. weiterhin eine gute Antwortzeit erreicht werden, so ist es nötig, dass man einige der eintreffenden Daten verwirft. Dies wiederum führt zu einer Verschlechterung des Datenverlustrate-Kriteriums. Diese gegenseitigen Beeinflussungen und deren Ausmaß sind ebenfalls a priori schwer zu ermitteln, da unter Umständen sehr viele Faktoren aufeinander einwirken. Werden zu hohe Anforderungen an das System gestellt, so wirkt sich das negativ auf die Dienstgüte aus, da die verwendeten Algorithmen zur Sicherstellung der QoS von erreichbaren Werten ausgehen. Sind die Anforderungen hingegen zu niedrig, so bleibt das System womöglich hinter seinen Möglichkeiten.

Aus diesen Gründen ist es in der Regel unmöglich, feste Garantien für die verlangte Dienstgüte zu geben. Es kann lediglich versucht werden, im Mittel die Anforderungen zu erfüllen und im Normalbetrieb gute Resultate zu erbringen. Traditionelle Ansätze von QoS (z.B. bei DFÜ-Verbindungen, speziell im Bereich Multimediaübertragung), die durch Reservierung von Ressourcen feste Zusicherungen an die Dienstgüte machen können, greifen nicht, da die Dynamik der Systemlast während der

Laufzeit nicht vorhergesagt werden kann und die verwendete Lösung vom Betriebsystem unabhängig funktionieren soll. Weiterhin soll zur Laufzeit auf bestimmte Kriterien besondere Rücksicht genommen werden, denn z.B. bei Echtzeitanwendungen will man rasch Ergebnisse haben (Kriterium Antwortzeit wird bevorzugt) wohingegen bei Überwachungsanwendungen kein Wert vergessen werden darf (Kriterium Datenverlustrate oder Attributwert wird bevorzugt behandelt). In jedem Fall ist es nötig, einen Kompromiss zwischen den verschiedenen Kriterien einzugehen.

4 QoS Framework in Aurora

Aurora übernimmt in Bezug auf QoS in DSMS eine Art Vorreiterrolle. So wurden in Aurora verschiedene Konzepte zur Spezifizierung und Einhaltung der Dienstgüte von Anfang an integriert und diese stellen einen wesentlichen Bestandteil des Systems dar. Die Dienstgüteanforderungen werden, wie bereits erläutert, vom Administrator in Form von zweidimensionalen Graphen in das System eingegeben. Die zwei bestimmenden Mechanismen zur Erhaltung der Qualität der Anfragen in Aurora sind: Lastverringerung und Scheduling.

Die Architektur von Aurora [ACC+03b, CCC+02] beruht auf sieben Bestandteilen (vgl. Abb. 3a): dem Storage Manager, dem Router, dem Scheduler, dem Load Shedder, dem QoS-Monitor, den Catalogs und den Box Processors. Die wesentlichen Elemente im QoS-Framework von Aurora sind der Load Shedder, der QoS Monitor, der Metadaten-Katalog (Catalogs) und der Scheduler.

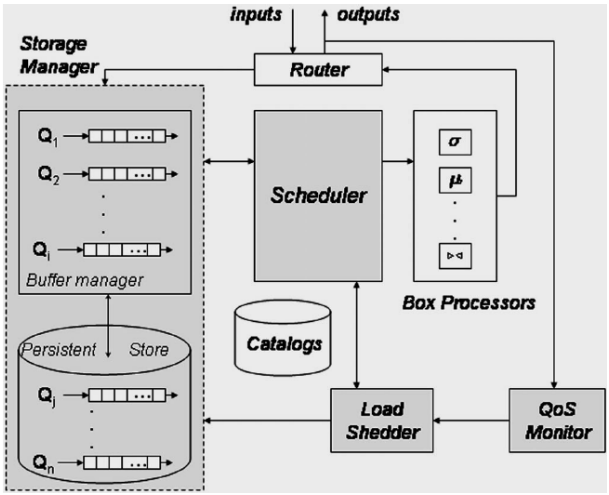


Abb. 3a: Architektur von Aurora [ACC+03b]

Der Load Shedder ist dafür zuständig, dass bei einer Überlastsituation Teile des ankommenden Datenstroms ignoriert bzw. verworfen werden. Der Scheduler ist dafür zuständig, die verfügbare Prozessorzeit an die Operatoren zu verteilen. Bei beiden genannten Einheiten werden jeweils die Dienstgütevorgaben des Benutzers berücksichtigt. Sie sind sozusagen die wichtigsten Exekutiven zur Aufrechterhaltung der QoS. Der QoS Monitor dient dazu, ständig die aktuelle Dienstgüte entsprechend der eingegebenen Kriterien zu überwachen und bei einer drohenden Verschlechterung entsprechende Maßnahmen einzuleiten. In späteren Versionen von Aurora wurde der QoS-Monitor nicht mehr als eigenständige Komponente betrachtet, sondern in den Load Shedder integriert. An der Funktionsweise hat sich jedoch nichts geändert. Die Daten zur Überwachung erhält er von dem Router, der die Informationen einer jeden Query (also etwa Bearbeitungsdauer oder der Wert, falls dieser mit vom Attributwerte-Kriterium berührt wird) weiterleitet. Die eingegeben QoS Anforderungen holt sich der QoS-Monitor aus dem Metadaten-Katalog, wie auch die Informationen über Selektivität der Operatoren und sonstige Statistiken. Die letzten beiden Informationen (Selektivität und sonstige Statistiken) werden in Probeläufen vor der eigentlichen Systembenutzung ermittelt, die QoS-Anforderungen werden, wie in Kapitel 3 erläutert, vom Systemadministrator - ebenfalls vorher - festgelegt. Erkennt nun der QoS-Monitor eine Verschlechterung der Qualität, so benachrichtigt er den Load Shedder.
Der Ansatz, bei der Verringerung der Last QoS-Kriterien (hier speziell Attributwerte) zu berücksichtigen, wird auch *Semantic Load Shedding* (oder *Semantic Drop*) genannt. Ein weiterer Ansatz wäre, unabhängig von qualitativen Faktoren einen prozentualen Anteil aller ankommenden Tupel zu verwerfen, was auch als *Random Drop* bezeichnet wird. In beiden Ansätzen wird vom Load Shedder zur Laufzeit das Anfragenetzwerk geändert, indem je nach Lastsituation so genannte Drop-Operatoren (spezielle Filter Operatoren, die Tupel verwerfen und so die Last reduzieren) eingefügt oder wieder entfernt werden. Der Scheduler arbeitet dann auf dem neu entstandenen Anfragenetzwerk und der Prozessor bearbeitet so die neu eingefügten Drop-Boxen wie „gewöhnliche“ Operatoren. Demnach findet die eigentliche Tupelverwerfung erst statt, nachdem die Drop-Operatoren vom Scheduler ausgewählt und vom Prozessor abgearbeitet wurden.

Die QoS Architektur von Aurora berücksichtigt nur drei der vier beschriebenen QoS-Kriterien, nämlich *Antwortzeit*, *Datenverlustrate* und *Attributwerte*.

5 QoS-beeinflusstes Scheduling in Aurora

Da in Aurora immer nur eine Box zur selben Zeit bearbeitet werden kann, ist es nötig, dass den Boxen nacheinander Prozessorzeit zugeordnet wird. Das generelle Ziel dabei ist die Maximierung der Dienstgüte. Die Scheduling-Methode, die in Bezug auf QoS in Aurora zum Einsatz kommt, wird als *priority assignment* beschrieben. Bei diesem Verfahren werden den Boxen Prioritäten zugewiesen und die Box mit der höchsten Priorität bekommt dann die notwendige Prozessorzeit zugeordnet. Im Folgenden soll nun beschrieben werden, wie die Prioritäten mit Hilfe der QoS-Spezifikationen berechnet werden. Dabei wird aber lediglich das Antwortzeit-Kriterium berücksichtigt. Zunächst wird die Latenzzeit einer jeden Box *b – latency(b)* – berechnet. Dieser Wert bildet sich aus der durchschnittlichen Latenzzeit aller Tupel (also die Zeitspanne, die von der Ankunft eines Tupels in der Queue, über die Verarbeitung, bis hin zum Verlassen des Operatorbox, vergeht) in der/den Eingabequeue/s der betreffenden Box *b*.

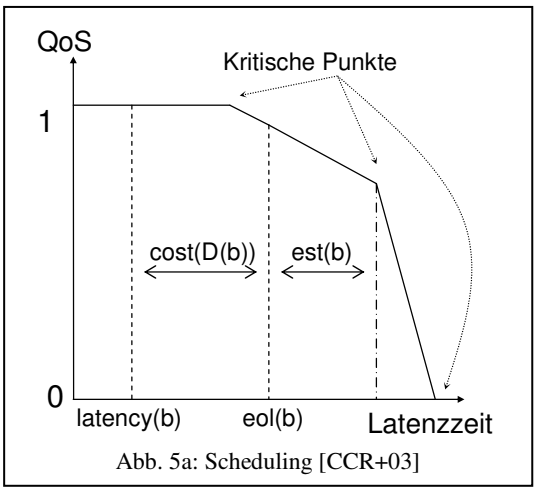


Abb. 5a: Scheduling [CCR+03]

Die Priorität jeder Box wird anhand von zwei Faktoren berechnet: des *Nutzens* und der *Dringlichkeit*. Als Nutzen wird in diesem Zusammenhang die Änderung der Dienstgüte beschrieben, die nach Abarbeitung der Box entsteht. Dieser Nutzen der Box *b utility(b)* wird durch die Steigung des QoS-Graphen an der Stelle *eol(b)* gekennzeichnet, wobei *eol(b)* die erwartete Ausgabeverzögerung der Box *b* darstellt (*expected output latency*). Mathematisch ausgedrückt: *utility(b) = gradient(eol(b)),* wobei die Funktion *gradient* die Steigung der Dienstgütefunktion ausgibt. Dabei ist *eol(b) = latency(b) + cost(D(b))* (vgl. Abb. 5a), wobei *cost(D(b))* eine Abschätzung der Dauer ist, die für die Abarbeitung der nach *b* folgenden Boxen *D(b)* benötigt wird. Je höher der Nutzen (also je positiver *utility(b)*) ist, desto geringer ist das Gefälle (für die Antwortzeit wird ein monoton fallender Graph angenommen) des QoS-Graphen an der betreffenden Stelle. Würde man also eine Box mit flachem Gefälle (entspricht hoher QoS) nicht auswählen, so würde viel Dienstgüte verloren gehen.
Die Dringlichkeit einer Box ist ein Maßstab, wie tolerabel eine längere Abarbeitungszeit im Vergleich mit der geschätzten Zeit *eol(b)* ist. Sie wird durch den kleinsten zeitlichen Abstand von *eol(b)* zu einem zeitlich nachfolgendem, so genannten *kritischen Punkt*, also eine Stelle, ab der sich die Steigung des QoS-Graphen stark verändert, beschrieben. Als Indikator für die Dringlichkeit dient die „Zeitreserve“ (*expected slack time*) *est(b)*, die alle möglichen weiteren Verzögerungen bei der Berechnung berücksichtigt. Sie ist der Abstand zu dem nächsten kritischen Punkt (vgl. Abb. 5a).

Nachdem diese beiden Werte für alle Boxen bestimmt wurden, können nun die Prioritäten errechnet werden. Dazu werden die Boxen zuerst nach ihrem Nutzen und anschließend nach ihrer Dringlichkeit sortiert, wobei sich die Prioritäten der Boxen als 2-Tupel-Liste wie folgt darstellen:

$$priority(b) = (utility(b), -est(b)).$$

Die Boxen werden also zunächst nach ihrem Nutzen ausgewählt (je höher desto besser) und anschließend nach ihrer Dringlichkeit (je niedriger desto besser, wird zwecks besserer Sortierbarkeit durch „-“ kompensiert).

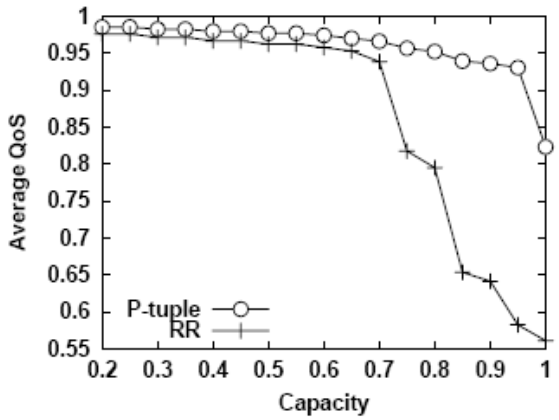


Abb. 5b: Dienstgüte in Abhängigkeit von der Systemauslastung (Capacity) [CCR+03]

Im Vergleich zu einer normalen Round-Robin Strategie, bei der die Operatoren gleichberechtigt der Reihe nach ausgewählt werden, bleibt das QoS-orientierte Scheduling auch bei sehr vielen Tupel in einer guten Dienstgüte (siehe Abb. 5b). Man erkennt, dass bei Round-Robin die Qualität bei voller Auslastung (*Capacity = 1*) nur noch die Hälfte beträgt, während Priority Assignment bei gleicher Auslastung noch einen Nutzen von über 80% erreicht [CCR+03].

6 Lastreduktion in Aurora

Ein DSMS muss aktiv bemüht sein, die Qualitätsanforderungen seiner Benutzer zu erfüllen. Dabei kommt es immer wieder zu Ressourcen-Engpässen, die z.B. durch Datenschübe ausgelöst werden und so die Anfrageberechnung verlängern. Bei solchen Überlastsituationen muss das DSMS Maßnahmen ergreifen, damit die gewünschte Dienstgüte (z.B. die Antwortzeit) in einem vertretbaren Rahmen bleibt. Dies wird durch Lastreduktion erreicht, d.h. Teile der ankommenden Daten werden vom System ignoriert, also werden nicht bei der Berechnung von Anfrageergebnissen berücksichtigt. Doch eine schlichte Verwerfung irgendwelcher Tupel wäre ebenfalls ungünstig für die Dienstgüte, da vielleicht sehr wichtige Informationen verloren gehen (siehe Attributwertkriterium). Das hauptsächliche Problem dabei ist, einen guten Kompromiss zwischen Lastreduktion und Erhaltung der Dienstgüte zu finden. Bei der Lastverringerung existieren vier zentrale Aufgaben [ACC+03b, CCC+02, TCZ+03a, TCZ+03b]:

1. Ermitteln, **wann** die Last zurückgenommen werden soll
2. Ermitteln, **wie viel** Last verringert werden soll
3. Ermitteln, **wo** die Last eingeschränkt werden soll
4. Ermitteln, **welche Tupel** verworfen werden dürfen

Die Arbeitsweise ist nun so, dass in Abhängigkeit von der Last, die permanent über die Eingabeströme eintrifft und der Last, die sich in den Queues der Operatoren ansammelt, überprüft wird, wann Last zurückgenommen werden muss. Dabei wird auch das Ausmaß der benötigten Lastreduktion bestimmt. Zum Abschluss muss noch ermittelt werden, wo im Anfragenetzwerk Last verringert werden soll und welche Tupel – unter Beachtung des Attributwertkriteriums – verworfen werden dürfen. Viele dieser einzelnen Arbeitsschritte können schon vorab durchgeführt werden und Informationen daraus stehen dem System statisch zur Laufzeit zur Verfügung. Der genaue Ablauf soll nun in den folgenden Abschnitten erläutert werden.

6.1 Wann und wie viel Last soll angepasst werden?

Punkt 1 und 2 wird üblicherweise vom QoS-Monitor bei einer Überlastung oder schon bei einer drohenden Überlastung an den Load Shedder gemeldet. Dabei vergleicht der QoS-Monitor die Soll-Daten der QoS-Kriterien aus dem Metadatenkatalog mit den vorliegenden Daten. Andererseits muss der QoS-Monitor auch sofort melden, falls zu viele Tupel fallengelassen werden, da sonst die Aussagekraft der bearbeiteten Anfragen erheblich leiden kann. Diese Aufgabe wird mit Hilfe einer dauernden aktiven Überprüfung der Systembelastung gelöst.

Um die Last des Systems festzustellen und korrekt anzupassen, ist es notwendig zu wissen, wie viel Last das System maximal bewältigen kann. Dazu werden vor der Systemlaufzeit Analysen durchgeführt, die die Belastung des Systems durch Tupel aus den Eingabeströmen zur Laufzeit abschätzen. Hierbei wird die mittlere Datenrate der Eingabeströme untersucht und die daraus entstehenden Bearbeitungskosten im Anfragenetzwerk ermittelt. Anhand dieser Werte wird die Hardware der Aurora-Plattform dimensioniert und dementsprechend die Systemkapazität C festgelegt. So stellt man sicher, dass das System im Regelfall die ankommende Last bewältigen kann (man betrachtet zur Dimensionierung die *mittlere* Datenrate), hin und wieder jedoch in eine Überlast geraten kann (z.B. bei *bursts*). Ausschlaggebend für die Größe von C ist, neben der zugrunde liegenden Hardware und Betriebsumgebung, auch der Prozentsatz an Ressourcen, die das DSMS vom Betriebssystem zugeteilt bekommt. Maßeinheit von C ist „Zyklen/Zeiteinheit“, also ein Maß, wie viele Berechnungszyklen pro Zeiteinheit maximal geschafft werden können. Die zulässige Belastbarkeit des Systems ergibt sich dann aus $C * H$, wobei die Konstante H als *headroom* bezeichnet wird und für die Lastreduktion einen gewissen Spielraum einräumt, damit frühzeitig auf Überlastung reagiert werden kann (es gilt also $0 < H \leq 1$).

Ausgehend von dieser Belastungsgrenze $C * H$ wird mit Hilfe der momentanen Systembelastung T die notwendige Lastanpassung ermittelt. Diese momentane Gesamtlast T des Systems besteht aus zwei

Teilen: zum einen aus der Last, die aus den Tupeln der Eingabeströme entsteht, welche die Operatorboxen durchlaufen müssen, bezeichnet als Datenstromlast (*stream load*) *S*. Zum anderen aus der Queuelast (*queue load*) *Q*, die durch die Tupel entsteht, die aufgrund von Verzögerungen in den Queues der Operatoren/Boxen vorhanden sind und bisher noch nicht abgearbeitet wurden. Es gilt: $T = S + Q$; Maßeinheit von *T* (und daher auch von *S* und *Q*) ist wieder „Zyklen/Zeiteinheit“. Lastverringerung muss nun durchgeführt werden, wenn $T > C * H$, also falls die Gesamtlast größer ist als das Leistungsvermögen des Systems. Entsprechend muss die Last des Systems wieder hochgefahren werden, wenn $T < C * H$ gilt. Die aktuelle Über-/Unterlastung des Systems ist gegeben durch die Differenz $T-(C * H)$; anhand dieses Wertes wird dann im laufenden Betrieb über die zu ergreifenden Maßnahmen entschieden (siehe Abschnitt „LSRM“).

Um nun den Wert *T* zu bestimmen, muss man nacheinander *S* und *Q* berechnen.

6.1.1 Bestimmung der Datenstromlast *S* (stream load)

Zunächst wird der Lastkoeffizient *L(I)* des Eingabestroms *I* berechnet. Hierbei sei, der Einfachheit halber, zunächst ein simples Anfragenetzwerk mit nur einem Eingabestrom *I* angenommen (vgl. Abb. 6a). Dieser Lastkoeffizient ist ein Maß darüber, wie teuer die Berechnung wird (d.h. wie groß die Anzahl an benötigten Zyklen pro Tupel ist), wenn ein neues Tupel über *I* in das System eintrifft und bis zu einem Ausgabestrom weitergegeben wird. Maßeinheit von *L(I)* ist „Zyklen/Tupel“ Dabei wird *L(I)* wie folgt berechnet:

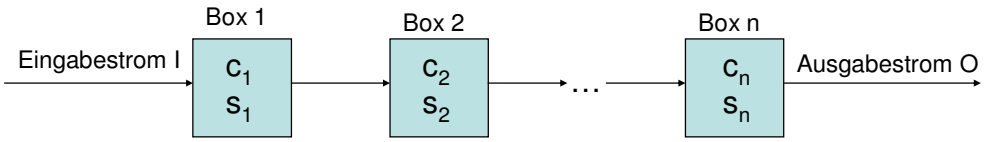


Abb. 6a: Fluss der Tupel über n Operatoren [TCZ+03b]

$$L(I) = \sum_{i=1}^{n} (\prod_{j=1}^{i-1} s_j) * c_i$$

Wobei s_i die Selektivität und c_i die Kosten (Anzahl der benötigten Prozessorzyklen pro Tupel) für eine Operatorbox *i* ist. Die genaue Größe von Selektivität und Kosten werden statistisch, in repräsentativen Probeläufen, vor der eigentlichen Systemlaufzeit ermittelt. Die Konstante *n* entspricht hierbei der Anzahl der Operatoren (vgl. Abb. 6a), die ab dem Eingabestrom bis zum Ausgabestrom durchlaufen werden. Im Lastkoeffizienten werden alle vorhergehenden Selektivitäten einer Box multipliziert und mit dem Kostenfaktor malgenommen.
Dieser Lastkoeffizient $L(I_i)$ (wobei $1 \leq i \leq m$) wird nun für alle *m* Eingabeströme des Systems berechnet. Die Datenstromlast *S* ergibt sich dann, wenn man die Lastkoeffizienten $L(I_i)$ aller *m* Ströme je mit der mittleren Aktualisierungsrate r_i (diese gibt an, wie häufig neue Tupel am Eingabestrom *i* eintreffen, Maßeinheit „Tupel/Zeiteinheit“) der Ströme multipliziert:

$$S = \sum_{i=1}^{m} L(I_i) * r_i$$

Man erhält also die mittleren Kosten (Zyklen pro Zeiteinheit) für die Abarbeitung der Tupel, welche über die m Datenströme in das System gelangen, das Anfragenetzwerk durchlaufen und über einem Ausgabestrom das System wieder verlassen [TCZ+03b].
(*Anmerkung:* die hier verwendete Schreibweise der Formeln unterscheidet sich, aufgrund der besseren Verständlichkeit, etwas von der Schreibweise in [TCZ+03b]. Hier wird der Lastkoeffizient von einem Eingabestrom *I* als Funktion *L(I)* dargestellt, während der, im nächsten Abschnitt erklärte, Lastkoeffizient einer Operatorbox *k*, als L_k bezeichnet wird. [TCZ+03b] verwendet für beide Koeffizienten die Schreibweise L_k, was zu Missverständnissen führen kann.)

6.1.2 Bestimmung der Warteschlangenlast Q (queue load)

Nachdem *S* nun bestimmt ist, gilt es die potentielle Last *Q* zu bestimmen, die in den Warteschlangen (Queues) der Operatorboxen wartet. Dazu muss zunächst, auf ähnliche Weise wie *L(I)*, die Lastkoeffizienten der Operatorboxen L_k berechnet werden. L_k ist dabei die Anzahl der Zyklen, die benötigt wird, wenn ein Tupel ab der Box *k* zu einem Ausgabestrom transportiert wird (Maßeinheit wieder „Zyklen/Tupel“). Aber im Unterschied zur Berechnung von *L(I)* werden hier nur die nach der Box *k* folgenden Boxen mitbetrachtet (Laufvariablen *i* und *j* beginnen ab *k*):

$$L_k = \sum_{i=k}^{n} (\prod_{j=k}^{i-1} s_j) * c_i$$

Nun wird weiter eine Variable definiert, die so genannte *MELT_RATE* (zu Deutsch: „Schmelzrate“), die den prozentualen Anteil festlegt, der pro Zeiteinheit in jeder Queue mindestens abgebaut werden soll (Maßeinheit „1/Zeiteinheit“). Der Wert $MELT\_RATE*L_i*q_i$ gibt dann an, wie viele Zyklen pro Zeiteinheiten benötigt werden, um alle der an Box *i* wartenden Tupel abzuarbeiten, wobei q_i (Maßeinheit „Tupel“) der Anzahl der wartenden Tupel entspricht und durch L_i auch die Kosten alle nachfolgenden Boxen berücksichtigt werden. Der Gesamtaufwand für alle wartenden Tupel im gesamten Netzwerk ergibt sich dann aus:

$$Q = \sum_{i=1}^{n} MELT\_RATE * L_i * q_i$$

Nun kann die aktuelle Gesamtbelastung *T* des Systems aus der Summe von *Q* und *S* ermittelt werden. Falls nun die Differenz $T-(C*H) < 0$ ist (d.h. die aktuelle Auslastung *T* ist kleiner als die zulässige Gesamtkapazität *C*H*), so sind noch Leistungsreserven vorhanden, ist die Differenz positiv, so muss Last reduziert werden und falls das Ergebnis 0 lautet, so ist das System im Gleichgewicht. Damit ist die Frage „wann?“ beantwortet, ebenso wie die Frage „wie viel?“, nämlich mit der Differenz *T-(C*H)* [TCZ+03b].

6.1.3 Beispiel

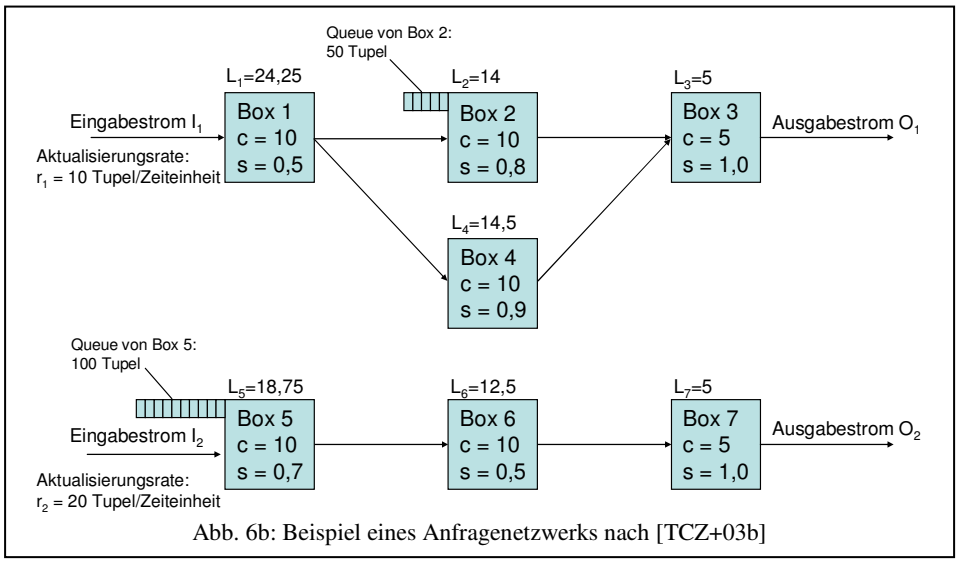


Abb. 6b: Beispiel eines Anfragenetzwerks nach [TCZ+03b]

In Abb.6b gilt:

$$L(I_1) = L_1 = c_1 + \underbrace{(s_1 * c_2 + s_1 * c_4)}_{(1)} + \underbrace{(s_1 * s_2 * c_5 + s_1 * s_4 * c_5)}_{(2)}$$

$$= 10 + (0{,}5 * 10 + 0{,}5 * 10) + (0{,}5 * 0{,}8 * 5 + 0{,}5 * 0{,}9 * 5) = 24{,}25$$

Zu beachten ist, dass *Box 1* mit **zwei** nachfolgenden Boxen verbunden ist. Dabei werden die Kosten von beiden Boxen (1) entsprechend addiert, da ja an beide Boxen Tupel weitergegeben werden und diese somit an beiden Boxen Kosten verursachen. Weiterhin muss darauf geachtet werden, dass nun

beide Wege weiterverfolgt werden (Nachfolger von *Box 2* und *Box 4*) und diese Kosten hinzuaddiert werden (2).
Analog berechnet man:

$L_2 = c_2 + s_2 * c_3 = 10 + 0{,}8 * 5 = 14$
$L_3 = c3 = 5$
$L_4 = c_4 + s_4 * c_3 = 10 + 0{,}9 * 5 = 14{,}5$

$L(I_2) = L_5 = c_5 + s_5 * c_6 + s_5 * s_6 * c_7 = 10 + 0{,}7 * 10 + 0{,}7 * 0{,}5 * 5 = 18{,}75$
$L_6 = c_6 + s_6 * c_7 = 10 + 0{,}5 * 5 = 12{,}5$
$L_7 = c_7 = 5$

Die Stromlast *S* ist dann:

$S = L(I_1) * r_1 + L(I_2) * r_2 = 24{,}25 * 10 + 18{,}75 * 20 = 617{,}5$

Man braucht also *617,5* Berechnungszyklen, um alle Tupel der Eingabeströme, die in einer Zeiteinheit ankommen, zu bewältigen.

Unter der Annahme, dass *MELT_RATE = 0,1* (also pro Zeiteinheit werden 10% der Tupel in den Queues verarbeitet), und sich wie in Abb. 6b nur Tupel in den Queues von Box 2 und Box 5 befinden, gilt für die Queuelast Q:

$Q = MELT\_RATE * (L_2 * (\#\ Tupel\ in\ Queue\ von\ Box\ 2) + L_5 * (\#\ Tupel\ in\ Queue\ von\ Box\ 5))$
$= 0{,}1 * (5 * 50 + 18{,}75 * 100)$
$= 212{,}5$

Man braucht also *212,5* Zyklen pro Zeiteinheit, um die in den Queues wartenden Tupel alle zu verarbeiten.

Die Gesamtlast *T* ergibt sich dann aus der Summe von *S* und *T*, also:

$T = 617{,}5 + 212{,}5 = 830$

Es sind also im Schnitt *830* Zyklen nötig, um die anfallende Last im System zu bewältigen, die in einer Zeiteinheit anfällt. Liegt dieser Wert über der zulässigen Systemkapazität *C*H*, so müssen Maßnahmen zu Lastverringerung ergriffen werden [TCZ+03b].

6.2 Wo soll die Last angepasst werden?

Für Punkt 3 wird vor dem eigentlichen Betrieb des Systems festgelegt, wo sich im Anfragenetzwerk geeignete Stellen befinden, an denen eine Tupel-Verwerfung stattfinden kann. Diese Verwerfung von Tupeln geschieht dann im laufenden Betrieb mit Hilfe von Drop-Operatoren, die bei Bedarf an die

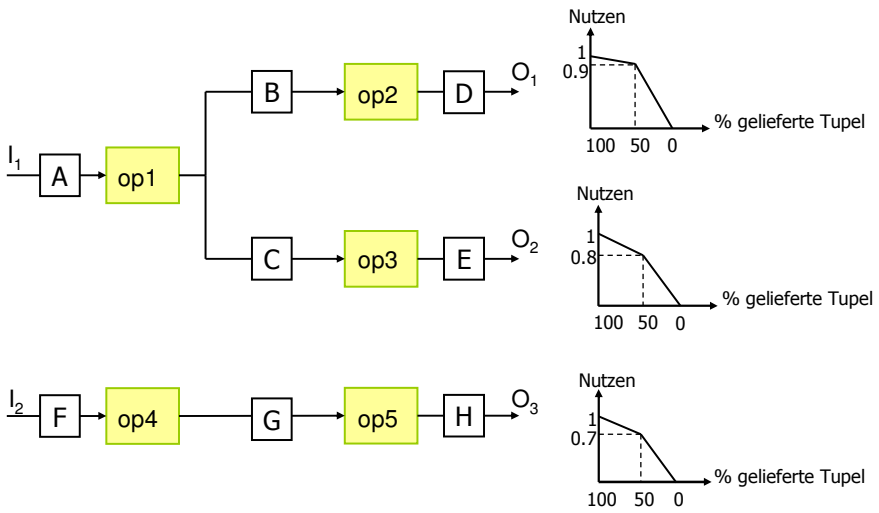


Abb. 6c: Auswirkungen der Wahl des Verwerfungspunktes auf die Qualität der Ausgaben; nach [TCZ+03b]

vorher berechnete Stelle ins Netzwerk eingefügt werden. Dabei handelt es sich im Grunde um Filter-Operatoren, die bei Semantic Drop ein bestimmtes Filterprädikat besitzen und bei Random Drop einfach einen bestimmten prozentualen Anteil aller Tupel aussortieren. Bei der Wahl solcher Verwerfungsstellen muss man zwischen zwei Aspekten abwägen: (1) man erreicht eine maximale Lastreduktion und (2) man minimiert den Verlust der Qualität/des Nutzens. Wählt man z.B. den Verwerfungspunkt ganz am Anfang eines Netzwerkes (Abb. 6c, Position A), so erhält man mit Sicherheit eine sehr hohe Lastreduktion, da an alle folgenden Operatoren (op1, op2 und op3) weniger Tupel weitergegeben werden, was viel Rechenzeit erspart. Dafür wird aber in Kauf genommen, dass Ausgabeströme, die eine hohe QoS-Anforderung in Sachen Datenverlustrate haben, sich in ihrer Qualität und Aussagekraft verschlechtern. Setzt man dagegen den Verwerfungspunkt sehr spät in der Hierarchie (Abb. 6c, Position B oder C), so erhält man zwar eine entsprechend höhere Qualität der Anfragen aber es wird nur sehr wenig Last reduziert (nur an op3 oder op2 kommen weniger Tupel an). In einem Anfragenetzwerk kann es mehrere solche Stellen geben, die einen guten Kompromiss zwischen den beiden Parametern ergeben. Wichtig ist jedoch, dass die Entscheidung über das Ausmaß der Lastreduktion immer auch eng mit der Entscheidung, wo reduziert werden soll, gekoppelt ist. Denn jeder Verwerfungspunkt erlaubt nur ein gewisses Maß an Tupeln, die verworfen werden. Alles was darüber hinaus geht beeinflusst zu sehr den Endnutzen. Diesen Vorgang gilt es nun zu automatisieren:

Um herauszufinden, welche Punkte besonders vorteilhaft sind, berechnet man das Verlust/Gewinn Verhältnis in Abhängigkeit von der relativen Anzahl der zu ignorierenden Tupel, also das Verhältnis vom erwarteten Dienstgüteverlust zu der erwarteten Einsparung an Prozessorzeit. Die Position mit dem niedrigsten Verlust/Gewinn-Verhältnis wird dann entsprechend ausgewählt. Um den Verlust an Dienstgüte zu bestimmen, ermittelt man den Prozentsatz an Tupel, die nicht an der Ausgabe ankommen würden. Zusammen mit dem Datenverlustrate-Graphen kann man dann die neue Dienstgüte und den Verlust an Dienstgüte *U(x)* berechnen. Um den Gewinn an Prozessorzyklen zu berechnen, muss man die Berechnungskosten *L* aller Operatoren wissen, die hinter dem Drop-Operator folgen. Zusammen mit der Selektivität *R* kann man dann die Einsparung *G(x)* für den Verwerfungsanteil *x* (also den Prozentsatz der verworfenen Tupel) ermitteln:

$$G(x) = \begin{cases} R*(x*L-D) & falls\ x>0 \\ 0 & sonst \end{cases}$$

Nun wird diejenige Drop-Position ausgewählt, an der der Quotient *U(x)/G(x)* am kleinsten ist, d.h. der Verlust an Dienstgüte im Verhältnis zur gewonnenen Prozessorzeit am besten ist. Da zur Berechnung dieser Verhältnisse alle Werte im Vorfeld ermittelt werden können, werden diese auch vor der Laufzeit bestimmt. Dabei ist zu beachten, dass dieses Verhältnis immer in Abhängigkeit von *x* ausgerechnet wird. Um nicht zu viele Werte berechnen zu müssen, wird eine feste Schrittweite vorgegeben, um die sich *x* verändern kann. Bei dieser Positionsbestimmung gilt es jedoch immer zu berücksichtigen, dass sich eventuell schon Drop-Operatoren im Netzwerk befinden. Dann muss auch eventuell in Erwägung gezogen werden, die Drop-Rate dieser Operatoren zu erhöhen, um die gewünschte Lastverringerung zu erreichen. Aber auch andere Optionen, wie z.B. das Vorziehen von Drops oder die Ersetzung durch Drop-Operatoren an anderen Positionen ist zu beachten. Diese Betrachtungen verkomplizieren die Berechnung natürlich enorm, es sei deshalb nur der Vollständigkeit halber darauf hingewiesen [TCZ+03b].

6.3 Welche Tupel dürfen verworfen werden?

Für Punkt 4 muss man unterscheiden, nach welcher Strategie man vorgeht. Für die Random Drop Methode wird lediglich der prozentuale Anteil der fallengelassenen Tupel angepasst. Diese Aufgabe lässt sich mit *T-(C*H)* als Maßstab bereits lösen. Verfolgt man hingegen die QoS-orientierten Methode (Semantic Drop), so ist zusätzlich noch zu bestimmen, welche Tupel verworfen werden dürfen und welche nicht. Problematisch ist, dass man bei der semantischen Methode keine direkte Aussage drüber machen kann, welche Tupel das im Einzelnen sind, da bisher nur ein prozentualer Wert bekannt ist, wie viele Tupel verworfen werden sollen. Die vom Administrator als besonders wichtig gekennzeich-

nete Werte (Kriterium Attributwerte) müssen schließlich erhalten bleiben. Der genaue Vorgang für das Semantic Drop Verfahren ist nun wie folgt (vgl. Abb. 6d): Nachdem der exakte Ort des Drop-Operators bestimmt wurde, wird mit Hilfe des betroffenen Attributwert-Graphen (Abb. 6d, links) und der Häufigkeit der Attributwerte (Abb. 6d, Mitte) ein kombinierter Graph erstellt. Die relativen Häufigkeiten der Attributwerte werden statistisch ermittelt und können zur Laufzeit angepasst werden. In Abb.

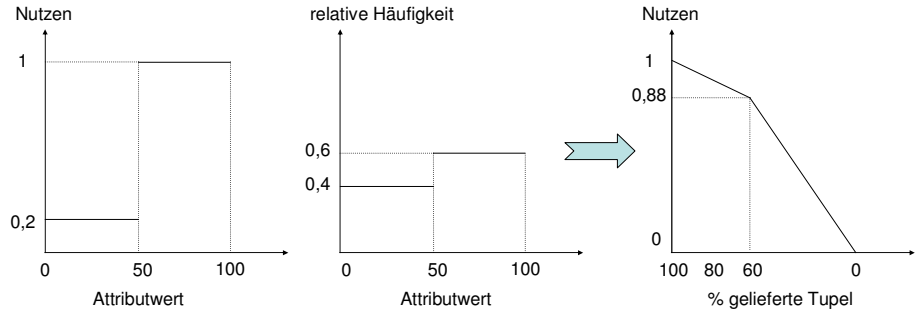


Abb. 6d: Vom prozentualem Anteil zu genauem Tupelwert (1) [TCZ+03b]

6d sieht man beispielsweise, dass die Attributwerte von [0; 50] eine niedrige Dienstgüte aufweisen und diese Werte in 40% aller Fälle auftreten. Kombiniert man diese Werte und rechnet sie auf die Gesamthäufigkeit aller Tupel um, so erhält man den in Abb. 6d dargestellten Graph (rechts). Bei der Konstruktion des Graphen wird davon ausgegangen, dass die Tupel mit dem niedrigsten Nutzen (im Beispiel also die Tupel mit Werten in [0; 50]) zuerst fallen gelassen werden. Das erklärt die abrupte Änderung der Steigung bei 60%, da bis zu diesem Punkt die „minderwertigen“ Tupel weggeworfen werden und ab dann auch Tupel mit hohem Nutzen wegfallen müssen. Mit Hilfe dieses generierten Graphen kann man nun wie beim Random Drop Verfahren weiter vorgehen; man hat nun sozusagen eine Abbildung von prozentualen Anteilen auf die Tupelwerte.

Jetzt wird also entschieden, welche Tupel mit welchen Attributwerten verworfen werden sollen. Dazu ist es nötig, ein geeignetes Filterprädikat für den Verwerfungspunkt zu finden. Diese Entscheidung ist

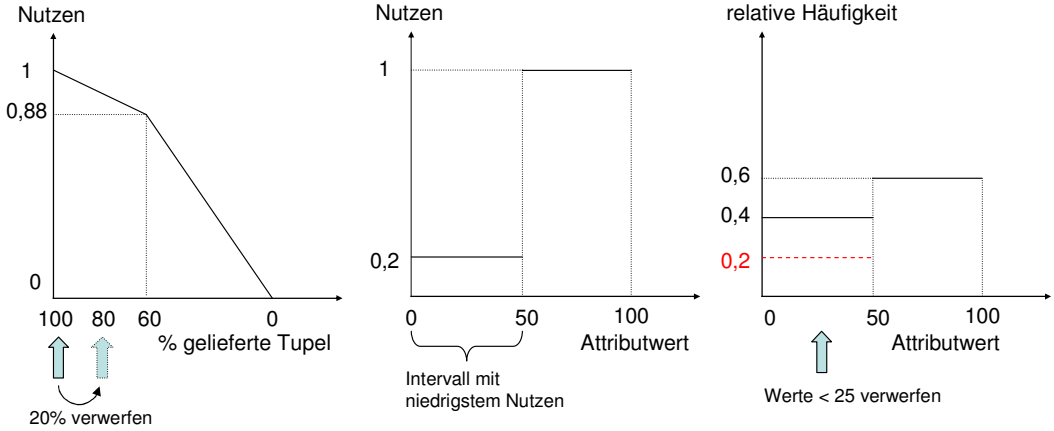


Abb. 6e: Vom prozentualem Anteil zu genauem Tupelwert (2) nach [TCZ+03b]

wie bereits erwähnt nur für die Strategie Semantic Drop relevant, da bei Random Drop die Tupel zufällig verworfen werden. Dabei werden die QoS-überwachten Werte überprüft und festgestellt, ob diese in einem Bereich von hoher Priorität liegen. Ist dies nicht der Fall, so darf das Tupel verworfen werden. Da bekannt ist, wie viel Prozent der Tupel verworfen werden sollen, muss noch ermittelt werden, welche Attributwerte dazu in Betracht kommen. Dazu benutzt man den im letzten Schritt konstruierten Graphen. Wie in Abb. 6e sei z.B. bekannt, dass 20% aller Tupel ignoriert werden sollen. Das bedeutet, dass insgesamt nur noch 80% aller Tupel bei der Berechnung benutzt werden, wenn man davon ausgeht, dass vorher keine Tupelverwerfung stattgefunden hat. Damit ist nun weiter bekannt, dass insgesamt nur Tupel mit niedrigem Nutzen fallengelassen werden müssen, da die kritische Grenze bei 40% Verwerfung liegt (Intervall [0; 50] siehe obige Erläuterung). Da nun quasi die Hälfte dieser 40%

wegfallen müssen, wählt man als Verwerfungsintervall [0; 25]. Damit steht nun auch das Filterprädikat fest; es werden nur noch Tupel durchgelassen, deren Attributwert ≥ 25 beträgt. Ein anderes Beispiel: wenn man insgesamt 70% aller Tupel eliminieren müsste, so müssten alle Tupel mit Werten aus [0; 50] (≡ 40%) und die Hälfte der Tupel aus [51; 100] (≡ 30%) verworfen werden. In diesem Fall wäre das Filterprädikat: Attributwert ≥ 75. Man benutzt quasi den konstruierten Graphen als Vermittler zwischen prozentualem Anteil und genauen Tupelwerten [TCZ+03b].

6.4 Load Shedding Roadmap (LSRM)

Die Ergebnisse der Punkte 1 bis 4 werden dann im Vorfeld der Systembenutzung verwendet, um die so genannte Load Shedding Road Map (LSRM, Abb. 6f) zu erstellen. Diese Tabelle ist statisch und wird zur Laufzeit nicht mehr verändert. Darin vermerkt ist die momentane Belastungssituation (in Abb. 6f „cursor“, steht zwischen 10 und 20% Überlastung) und die Maßnahme, um die Belastung zu verringern („Drop Insertion Plan“, kurz DIP). Je nachdem, wie hoch die Überlastung des Systems ist,

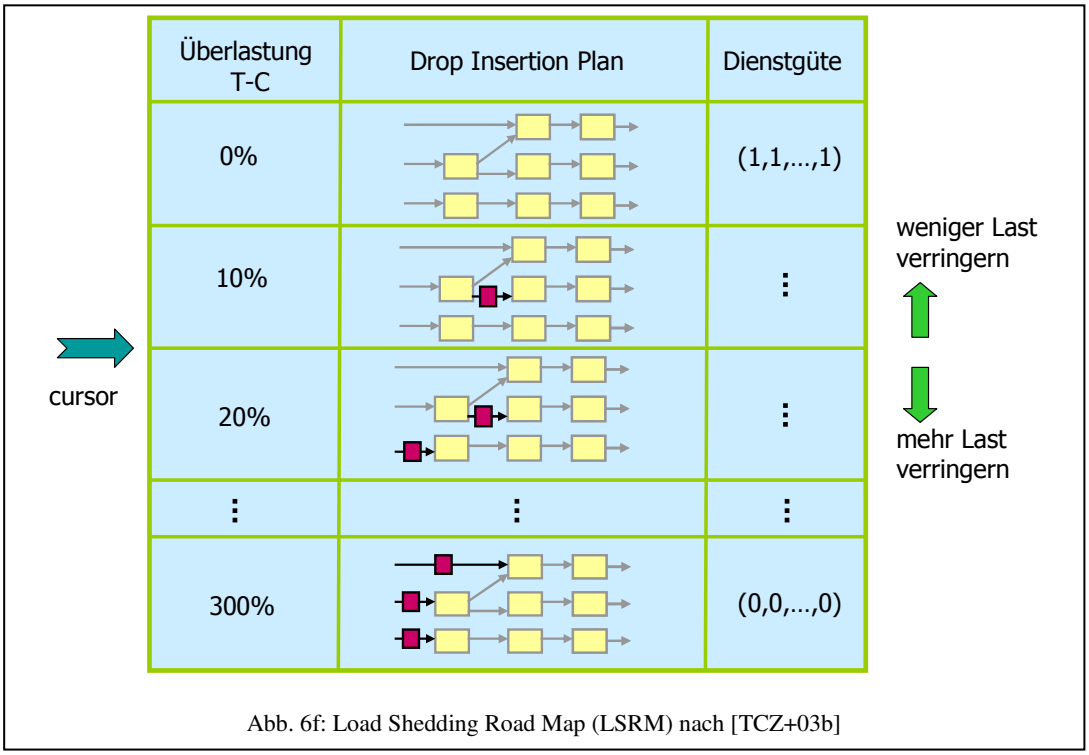


Abb. 6f: Load Shedding Road Map (LSRM) nach [TCZ+03b]

ändert sich also der auszuführende Plan. Je höher die Belastung, desto mehr Drop-Operatoren müssen platziert werden bzw. desto früher müssen die Drops in der Netzwerkhierarchie gesetzt werden, usw. Ist die Überlastung maximal (in Abb. 6f 300%), so werden praktisch alle ankommenden Tupel verworfen, die Dienstgüte aller Kriterien sinkt auf Null. Der Cursor bzw. die Überlastung wird also ständig überwacht und dabei jeweils der aktuelle DIP ausgeführt, auf die er zeigt [TCZ+03b].

Der prinzipielle Ablauf bei einer Lastreduktion sieht nun so aus:
Im Vorfeld der Systemnutzung wird wie oben beschrieben die LSRM erstellt. Im Betrieb bekommt der Load Shedder vom QoS Monitor mitgeteilt, dass eine Überlastung vorliegt. Die genauen Informationen, wann und wie viele Tupel verworfen werden müssen erhält er vom QoS-Monitor bzw. berechnet er aus den erhaltenen Werten. Der Load Shedder entscheidet nun, welche Tupelverwerfungspunkte aktiviert werden sollen. Dazu schaut er in der LSRM nach und entscheidet je nach Überlastungsgrad, welcher DIP ausgeführt werden soll. Je nachdem wie sich die Dienstgüte nach diesen Maßnahmen verändert, bekommt der Load Shedder entweder den Auftrag noch mehr Last zu verringern, oder aber mehr Tupel durchzulassen, falls die Qualität sich verbessert hat.

7 Weitere Ansätze

Neben den QoS-Konzepten in Aurora existieren noch weitere Verfahren, die entweder die oben erläuterten Problem (Scheduling, Lastanpassung) anders lösen (7.1) oder neue Mechanismen zur Regulierung und Optimierung der Dienstgüte einführen (7.1.4 und 7.2). Auf diese Konzepte soll nun eingegangen werden.

7.1 Ein weiteres QoS-Framework

Das folgende System aus [TuXP04] will zwei Punkte bestehender Systeme verbessern. Ein Kritikpunkt der Autoren aus [TuXP04] war, dass bei existierenden Systemen kein richtiges Kontrollmodell eingesetzt wird, d.h. Entscheidungen z.B. zur Lastverringerung nicht aufgrund fundierter, theoretischer Grundlagen getroffen werden. Um dies zu bereinigen, benutzen die Entwickler bestehende Erkenntnisse aus der Regelungstechnik (*feedback control theory*) und formulieren das Problem der Leistungsanpassung des DSMS entsprechend um (siehe Regelkreis Abb. 7a) in ein analytisches Modell zur QoS-Kontrolle.

Des Weiteren soll – laut den Autoren – die Unterstützung benutzerspezifischer QoS-Anforderungen erweitert werden. Denn die Spezifikationen die bezüglich QoS z.B. in Aurora gemacht werden können, betreffen nur die Systemebene und gelten global für alle Nutzer. Zwischen verschiedenen Benutzeranforderungen kann, laut [TuXP04], nicht genug differenziert werden. Um dies zu beheben muss der Benutzer – wie in Kapitel 2 erklärt – zusätzlich zu seinen Anforderungen einen Vektor angeben.

Dieses System betrachtet dabei nur die Kriterien der Antwortzeit, Datenverlustrate und der Update-Rate. Das Hauptziel dieser Architektur ist die Einhaltung von Bearbeitungs-Deadlines. Das bedeutet, dass die Bearbeitungszeit jedes Tupels (also die Zeit vom Eindringen des Tupels über einen Eingabestrom bis zu dem Zeitpunkt, an dem das Tupel nicht mehr weiter verarbeitet wird), vom System innerhalb des zeitlichen Rahmens gehalten wird, der durch das Antwortzeitkriterium jedes Eingabestrom spezifiziert wurde.

7.1.1 Architektur

Das System beruht in seiner Funktionsweise auf einem Regelkreis (feedback-control-loop) [TuXP04], in der die aktuelle Dienstgüte (vgl. Abb. 7a: Ausgabewerte) ständig mit der angestrebten Dienstgüte (Abb. 7a: Referenzwerte) verglichen wird und über die Differenz der beiden Werte Rückschluss über

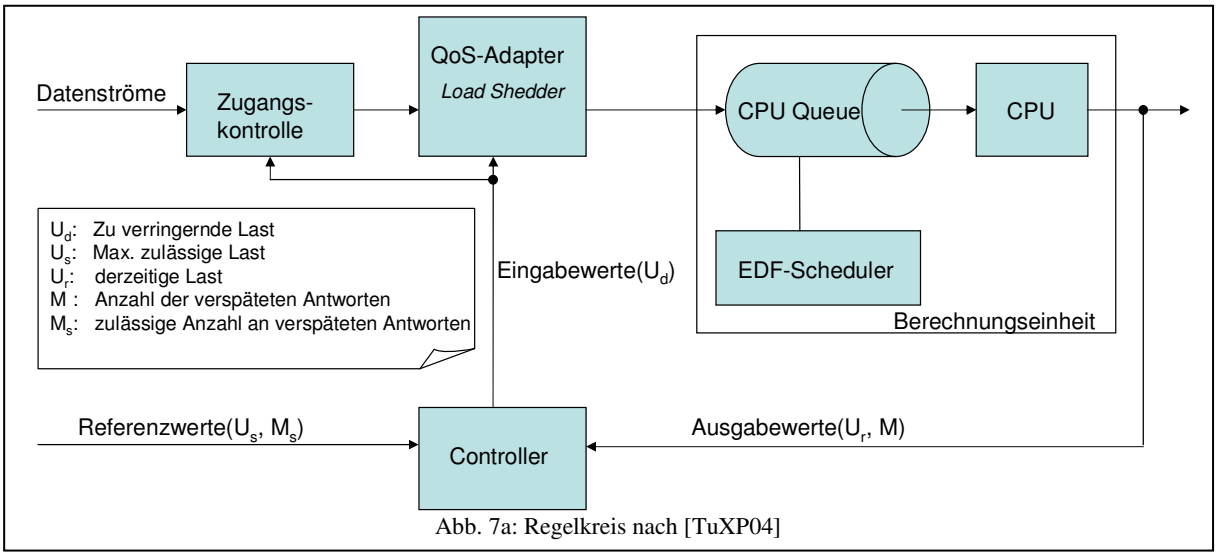


Abb. 7a: Regelkreis nach [TuXP04]

die zu ergreifenden Aktionen (Eingabewerte) gezogen wird. Nach dem Ausführen dieser Aktionen, beginnt der Zyklus erneut. Die wesentlichen Elemente der Schleife sind: die Zugangskontrolle, der QoS-Adapter, der Controller, die Berechnungseinheit und der integrierte Scheduler.
Die Zugangskontrolle dient dabei der Anknüpfung neuer Datenströme, der QoS-Adapter ist für Dienstgüte erhaltende Maßnahmen zuständig. Der Controller überwacht die Systemlast und in der Berechnungseinheit findet die eigentliche Auswertung der Anfragen statt. Der Scheduler ist für die Prozessorzuweisung an die Anfragen (bzw. *operator pipelines*) zuständig.

7.1.2 Lastreduktion

Der konkrete Ablauf bei einer Lastverringerung sieht so aus: Zunächst erhält der Controller von der Berechnungseinheit die aktuellen Daten über die Last des Systems und den prozentualen Anteil der bisherigen verpassten Deadlines. Daraus berechnet er mit Hilfe der zulässigen Last und dem zulässigen relativen Anteil an Verspätungen, wie viel Last angepasst werden soll. Um nun die Last anzupassen, wird ein Eingabestrom ausgewählt, an dem entweder die Datenverlustrate oder die Update-Rate angepasst wird. So gelangen bei Überlast weniger und bei Unterlast mehr Tupel in das System. Dabei wird immer nur ein kleiner Teil der Last angepasst und der Vorgang so lange wiederholt, bis das volle Ausmaß der Anpassung erreicht wird. Dann wird entschieden, ob noch mehr Last verringert werden soll, oder ob die ausgeführten Aktionen ausreichend waren [TuXP04].

Wann und wie viel Last soll reduziert werden?

Die Frage, wann und wie viel Last verringert werden soll, wird kontinuierlich vom Controller beantwortet. Dieser vergleicht die aktuelle Last $U(k)$ mit der maximalen zulässigen Last U_s und der prozentuale Anteil schon zu spät gekommener Tupel $M(k)$ mit dem maximal zulässigen Anteil zu spät gekommener Tupel M_s. Daraus berechnet der Controller den Anteil an Last, der angepasst (also entweder zurückgenommen oder hochgefahren) werden soll. Dieser Vorgang geschieht permanent und aktiv in einer Schleife. Dabei findet eine ständige Anpassung statt, da in jedem Schleifendurchlauf diese Überprüfung gemacht wird. Es wird nicht abgewartet, bis eine Überlastung oder eine Unterlastung eintritt, sondern kontinuierlich die Last abgestimmt. Natürlich kann dadurch aber nicht garantiert werden, dass es nie zu einer Überlastung kommt, da Änderungen an der Datenflussrate unter Umstände sehr schnell und ausgeprägt auftreten. Zur Steuerung des Datenstrom-Regelkreises bedient man sich eines sog. PID-Reglers (Proportional, Integral, Differenzial), der historische Werte zur Berechnung berücksichtigt. Der absolute Lastanteil $U_d(k)$, der geändert werden soll, wird wie folgt berechnet:

$$U_d(k) = \alpha E(k) + \beta \sum_{i=k-W}^{k-1} E(i)$$

Dabei ist $U_d(k) < 0$, wenn Last zurückgenommen werden muss, $U_d(k) > 0$, falls das System mehr belastet werden kann und $U_d(k) = 0$, falls das System ausgeglichen ist. Die Funktion E (siehe Unten) ist hierbei maßgeblich für die Ermittlung der benötigten Anpassung zuständig. Die Summe in der Gleichung berücksichtigt Anpassungen aus vorausgegangenen Schleifendurchläufen und verhindert allzu starke Wertschwankungen. Der endgültige Wert von $U_d(k)$ wird aus $E(k)$ und der Summe gebildet; das Verhältnis der Koeffizienten α und β bestimmt, wie stark die vergangenen Werte den aktuellen Wert von $U_d(k)$ beeinflussen. Die Funktion E ist dabei lediglich die Differenz zwischen Soll-Last und Ist-Last – und somit negativ –, falls eine Überlastung vorliegt ($U(k) \geq 1$). Ansonsten ist E positiv, da noch Leistungsreserven vorhanden sind. Wie groß E in dieser Situation wird, hängt davon ab, wie viele Ergebnistupel noch zu spät kommen dürfen, die Differenz $M_s - M(k)$ dient dabei als Maß. Der Korrekturfaktor m dient dazu, aus dieser Differenz Rückschlüsse auf die noch vorhandene Leistungsreserve des Systems zu ziehen. $E(k)$ berechnet sich also wie folgt:

$$E(k) = \begin{cases} U_s - U(k) & \text{wenn } U(k) \geq 1 \\ m(M_s - M(k)) & \text{sonst} \end{cases}$$

Nun kommt der QoS Adapter ins Spiel. Dieser ist dafür zuständig, dass die Dienstgüteanforderungen der Benutzer eingehalten werden. Darin integriert ist der so genannte Load Shedder, also die Einheit,

die die Last des Systems in einem zu bewältigenden Rahmen hält, in dem ankommende Tupel verworfen werden. Vom Controller erfährt der QoS-Adapter permanent, ob und wie viel Last anzupassen ist, indem er ihm den Wert $U_d(k)$ übermittelt. Aus dem aktuellen Wert von $U_d(k)$ und der aktuellen Lastanpassung kann nun entschieden werden, wie viel Last angepasst werden soll.

Eine weitere wesentliche Information fehlt allerdings noch: wo die Last reduziert werden soll. Dazu bietet das System wieder –ähnlich zu Aurora– zwei Strategien, Random Drop und Semantic Drop. Das semantische Verfahren zum Load Shedding weist jedoch eine höhere Flexibilität als in Aurora auf, da der Benutzer mehr Einfluss auf den Ort der Lastverringerung hat [TuXP04].

Wo soll die Last reduziert werden?

Die QoS-Anpassung wird in dieser Architektur etwas anders behandelt, als in Aurora. Wenn mehrere QoS-Kriterien zur Verfügung stehen, muss das System entscheiden, bei welchen Kriterien und in welchem Umfang eine Anpassung vorgenommen werden soll. Dieses Problem ist keineswegs trivial (und stellt selbst wieder ein komplexes Optimierungsproblem dar), da theoretisch unendliche viele Möglichkeiten bestehen, die Anpassung vorzunehmen. Ziel dabei ist eine Anpassung, bei der die Dienstgüte optimal ist.
Der gewählte Ansatz ist nun, dass der QoS-Adapter nach und nach mit vorher festgelegter Schrittweite die Anpassung vornimmt. Diese Schrittweite wird vom Benutzer eingegeben und kann separat für jedes QoS-Kriterium definiert werden. Dieses Vorgehen erspart das Konstruieren eines Algorithmus, welcher aus den erwähnten unendlich vielen Möglichkeiten, die Last zu reduzieren, eine auswählt. Ausgehend von einem Anfangsvektor $\{q_1, q_2\}$ - wobei q_1 der Datenfrequenz (Update-Rate) und q_2 der Datenverlustrate entspricht - geht der Adapter nun wie folgt vor, wenn die Last verringert werden soll ($U_d < 0$):

$$\{q_1, q_2\} \rightarrow \{q_1 - S_1, q_2 - S_2\} \rightarrow \{q_1 - 2S_1, q_2 - 2S_2\} \rightarrow \ldots$$

Es wird also Schritt für Schritt Last vom System genommen, bis eine akzeptable Dienstgüte erreicht wird. Soll hingegen wieder die Last erhöht werden ($U_d > 0$), so geht man entsprechend vor:

$$\{q_1, q_2\} \rightarrow \{q_1 + S_1, q_2 + S_2\} \rightarrow \{q_1 + 2S_1, q_2 + 2S_2\} \rightarrow \ldots$$

Dabei sind S_1 und S_2 die festgelegten Schrittweiten für q_1 bzw. q_2. Bei der Wahl dieser Schrittweiten spielen die vom Benutzer festgelegten Prioritäten w_1 und w_2 (vgl. Abschnitt 3.5) für die entsprechenden Kriterien eine Rolle. Dabei ist die Schrittweite umso kleiner, je höher die Priorität ist und umgekehrt. Das bedeutet, dass bei sehr wichtigen Kriterien immer nur wenig Anpassung (besonders Verringerung) stattfindet und bei weniger wichtigen entsprechend mehr. Dadurch bleibt die Dienstgüte bei wichtigen Kriterien auf hohem Niveau. Das Verhältnis zwischen Priorität und Schrittweise ist also umgekehrt proportional und lässt sich am besten anhand folgender Formel erkennen (anhand des Exponenten a lässt sich bei Bedarf das Verhältnis von w_1 und w_2 noch weiter anpassen; Standardwert ist $a = 1$):

$$\frac{w_1}{w_2} = \left(\frac{S_2}{S_1}\right)^a$$

Nachdem wir nun erläutert haben, wie die Anpassung von QoS-Merkmalen anhand der Schrittweiten durchgeführt werden kann, muss noch festgelegt werden, welchen Eingabestrom die Anpassung betreffen soll. Bisher steht lediglich der absolute Anteil $U_d(k)$ fest, um den die Last verringert werden soll. Welcher Strom von der Anpassung betroffen sein soll, steht noch nicht fest. Dazu stellt man, ähnlich wie in Aurora, eine Kosten/Nutzenanalyse auf. Dabei werden zur Anpassung nur die kompletten Ausgabeströme in Betracht gezogen.

$$E_i = \frac{h_i(f_i(\vec{q}_1) - f_i(\vec{q}_0))}{U(\vec{q}_0) - U(\vec{q}_1)}, \text{ wobei } 1 \le i \le n \text{ und } n \text{ die Zahl der Ausgabeströme ist.}$$

Dabei steht $\vec{q}_0$ für den aktuellen Dienstgütevektor und für den Vektor nach der Anpassung. Der Zähler repräsentiert die Kostendifferenz - also die gewonnene Prozessorzeit -, wobei h die Kosten pro

Tupel und *f* die Funktion ist, die für die gegebenen QoS-Vektoren die durchschnittliche Tupelanzahl berechnet. Der Nenner hingegen ist ein Maß für den zu erwarteten Dienstgüteverlust (Differenz aus der Dienstgüte vor der Anpassung $U(\vec{q}_0)$ und der Dienstgüte nach der Anpassung $U(\vec{q}_1)$, dabei ist *U()* die in Abschnitt 3.6 erläuterte Dienstgütefunktion). Dieser Gewinn/Verlust-Quotient wird für jede der *n* Anfragen (bzw. jeden Eingabestrom) berechnet und dann derjenige Eingabestrom zur Anpassung ausgewählt, welcher das größte *E* besitzt.
Der Ablauf ist nun so, dass ausgehend von einem bestimmten Ziel $U_d(k)$ iterativ anhand der Kosten/Nutzenanalyse ein Eingabestrom zur Anpassung ausgewählt wird. Sei *i* dieser ausgewählte Strom (E_i ist am größten) und *k* der aktuelle Zyklus. Dann wird mit der Schrittweite S_i die Eingabefrequenz des Stroms *i* angepasst. Dabei wird wegen der geringen Schrittweite S_i immer nur wenig Last angepasst. Das ganze wird so lange wiederholt, bis das Ziel $U_d(k)$ erreicht ist. Schließlich beginnt ein neuer Zyklus *(k+1)*, bei dem sich die beschriebene schrittweise Iteration wiederholt; diesmal jedoch bis man das neu bestimmte Ziel $U_d(k+1)$ erreicht hat.

7.1.3 Scheduling

Der Scheduler des Systems aus [TuXP04] verfolgt die Strategie *Earliest Deadline First* (EDF). Bei dieser Methode werden die absoluten Grenzen der Deadlines für Anfragen berücksichtigt. Erreicht ein für die entsprechende Anfrage relevantes Tupel das System zum Zeitpunkt *s* und besitzt die Anfrage eine zulässige Antwortzeit von *d* (d.h. ein Eingabetupel der Anfrage darf nicht länger als *d* Zeiteinheiten brauchen, um dem Anfragegraphen zu durchlaufen), so werden nach dieser Strategie die Anfragen nach $s + d$ sortiert und jeweils die dringlichste Anfrage erhält als nächstes den Prozessor. Eine andere mögliche Strategie wäre *Relative Deadline Monotonic* (RDM), bei der die Anfragen nach der zulässigen Antwortzeit *d* sortiert werden. Diese Strategie hat zwar im Gegensatz zu EDF eine bessere Laufzeitkomplexität (EDF: logarithmisch vs. RDM: konstant), ist aber aufgrund von Scheduling-Fairness-Gesichtspunkten zu vernachlässigen. So kann es bei RDM zu „starvation“ kommen, da immer die Operatoren ausgewählt werden, die die niedrigste zulässige Anfragedauer *d* besitzen und unabhängig davon, wie lange sie schon warten (*s*). Neuankommende Operatoren mit niedrigem *d* können somit schon lange wartenden Operatoren (größeres *d*) ständig den Prozessor wegnehmen.

7.1.4 QoS-überwachtes Anbinden neuer Streams

Hin und wieder kann es zur Laufzeit vorkommen, dass neue Streams an das System angebunden werden sollen. Dieser Vorgang kann die Auslastung des Systems unter Umständen sehr belasten, falls der Strom eine sehr hohe Aktualisierungsrate besitzt. Deshalb wird in [TuXP04] eine Möglichkeit geboten, schon vor der Anknüpfung des Stroms die Auswirkungen auf das System zu überprüfen.
Dafür zuständig ist die Zugangskontrolle. Diese Komponente überprüft, ob das System mit einem neu eingebundenen Strom in der momentanen Situation fertig werden kann. Dazu wird lediglich berechnet, ob die folgende Bedingung gegeben ist:

$$U_r + h_i r_i \leq U_s$$

Dabei ist U_r die aktuelle Systembelastung, wobei h_i die Kosten pro Tupelauswertung und r_i die Update-Frequenz darstellt. So kann im Vorfeld der Einbindung entschieden werden, ob dieser Vorgang ohne weiteres zulässig ist, oder ob zugunsten der Dienstgüte Lastverringerungsmaßnahmen ergriffen werden müssen bzw. ganz auf die Anbindung des neuen Stroms verzichtet werden muss.

7.1.5 Vergleich mit Aurora

Das Grundprinzip beider Konzepte ist gleich: Es wird ständig die Dienstgüte überwacht und bei einer Abweichung werden Lastverringerungsmaßnahmen eingeleitet. Jedoch existieren einige signifikante Unterschiede:

- Das hier vorgestellte System verwendet Konzepte aus der Regelungstechnik, um eine Lastanpassung vorzunehmen. Die Informationen zur Bestimmung der Anpassung werden dynamisch - zur Laufzeit - gewonnen. Im Gegensatz dazu, verwendet Aurora zum Teil statische Werte bei der Lastanpassung, die im Vorfeld der Systembenutzung in Probeläufen ermittelt wurden.
- Weiterhin werden bei der Lastanpassung im [TuXP04]-System die Werte aus vergangenen Anpassungsvorgängen berücksichtigt. So kann besser auf tendenzielle Lastveränderungen reagiert werden (z.B. kann so erkannt werden, dass bei schwankenden Werten die Last dennoch insgesamt steigt). In Aurora wird die Last ausschließlich anhand der momentanen Dienstgüte und der zu erreichenden Dienstgüte angepasst.
- Im zweiten System ([TuXP04]) wird nicht explizit Rücksicht auf das Attributwertkriterium genommen, wohingegen Aurora dieses Kriterium sehr wohl unterstützt und sogar speziell darauf abgestimmte Algorithmen zur Lastreduktion besitzt. Aurora bietet sogar den Vorteil, theoretisch auf beliebige Kriterien zurückzugreifen zu können. Das System aus [TuXP04] ist bisher auf die Antwortzeit, Update-Rate und Datenverlustrate beschränkt. Ob dies von Nachteil ist, ist von Anwendung zu Anwendung sicher unterschiedlich; in Einzelfällen, z.B. bei der erwähnten Reaktorüberwachung jedoch schon.
- Im System [TuXP04] kann jeder Benutzer seine Präferenzen in Vektorform spezifizieren. Somit ist es leichter möglich, die Anforderungen zu ändern und für jeden Benutzer getrennt anzupassen. Aurora bietet lediglich die Möglichkeit, an jeder Anfrage die Dienstgüte in Form von Graphen zu spezifizieren.
- Die Lastanpassung – speziell die Tupelverwerfung – findet im zweiten System auf einer gröberen Ebene statt, da lediglich zwischen den einzelnen Eingabeströmen gewählt wird und nicht wie bei Aurora auch noch zwischen allen mögliche Positionen zwischen den Operatoren des Anfragenetzwerks.

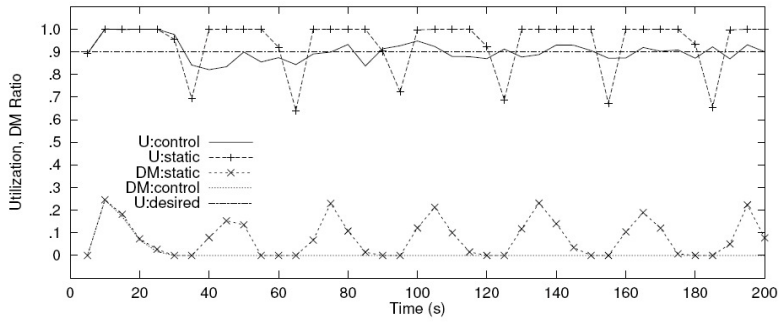


Abb. 7b: Vergleich (Simulation) mit Auora [TuXP04]

Das in Abb. 7b gezeigte Diagramm zeigt, dass bei statischen Verfahren zur Lastanpassung (U:static, entspricht dem Aurora System) die Dienstgüte relativ starken Schwankungen unterworfen ist. Im Gegensatz dazu pendelt die Dienstgüte des [TuXP04] Systems (U:control) nur schwach um die gewünschte Dienstgüte herum (U:desired). Bei den Verspätungen von Antworttupeln (Deadline Misses bzgl. Antwortzeit) erkennt man, dass das statische Verfahren (DM:static) periodisch bis zu 25% verspätete Ergebnistupel liefert, während das [TuXP04] System schon nach kurzer Zeit keine verspäteten Tupel mehr liefert (DM:control).

Es ist jedoch unklar, inwieweit Abb. 7b repräsentativ, als Vergleich zwischen Aurora und [TuXP04], ist. Denn Aurora berücksichtigt zum Beispiel auch das Kriterium Attributwert, was das [TuXP04]-System nicht tut.

7.2 QoS-Vorhersage bei Kontinuierlichen Anfragen

Der naive Ansatz in vielen DSMS ist es, laufend die aktuelle Dienstgüte zu bestimmen und bei einer Verletzung der Anforderungen Maßnahmen einzuleiten. Dabei wird die Dienstgüte ständig und aktiv mit Hilfe einer Überwachungsschleife geprüft und bei einer Abweichung wird entweder die Scheduling Strategie geändert, um kritische Tupel bevorzugt zu bearbeiten, oder es werden Schritte zur Lastverringerung eingeleitet (vgl. Abb. 7b). Ist die Qualität immer noch ungenügend, so wird der Vorgang wiederholt, bis ein akzeptabler Wert erreicht wird.
Diese Vorgehensweise hat jedoch entscheidende Nachteile: Durch die ständige Überwachung der QoS werden relativ viele Systemressourcen verbraucht, da ständig Prozessorzeit benötigt wird. Weiterhin werden Maßnahmen erst eingeleitet, wenn die Dienstgüteanforderungen bereits verletzt sind. Wird die Abweichung also nicht rechtzeitig erkannt, so kann die Qualität der Anfragen erheblich leiden. Auch die Entscheidung, wann die getroffenen Maßnahmen wieder zurückgenommen werden können, ist sehr ungenau. So kann es vorkommen, dass noch Last reduziert wird, obwohl eigentlich noch genügend Ressourcen für die Bearbeitung der Anfragen vorhanden wären.

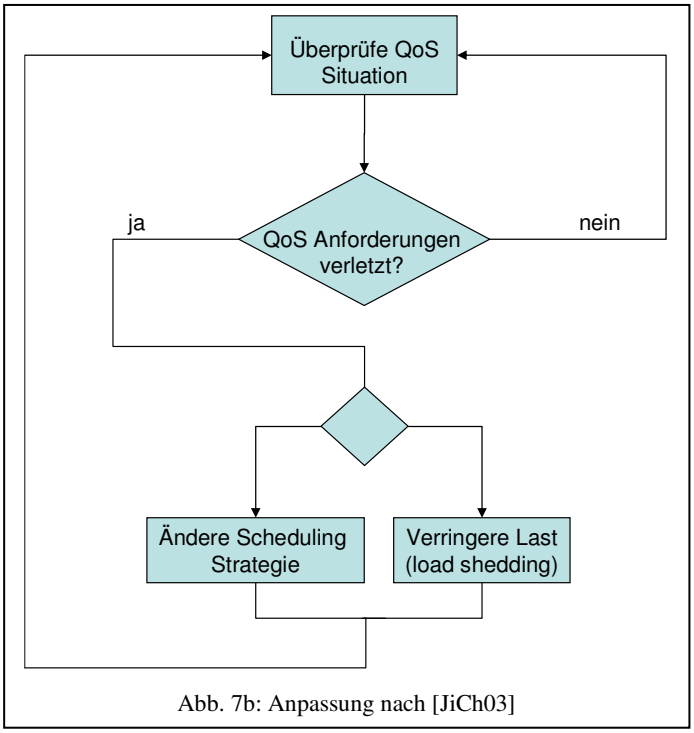


Abb. 7b: Anpassung nach [JiCh03]

Aus diesen Gründen wäre es sinnvoll, wenn eine Vorhersage über die Dienstgüte getroffen werden könnte. Ein System, dass die Dienstgüte zumindest für kontinuierliche Anfragen voraussagen kann, wird in [JiCh03] beschrieben. Dabei wird ein theoretisches Modell entwickelt, welches aus bekannten Parametern eine Aussage über die zukünftige Dienstgüte ermöglicht. Dadurch können Überlastungssituationen schon vor ihrer Entstehung erkannt und umgangen werden.
Die Autoren betrachten das DSMS als ein vernetztes System aus Warteschlangen, wobei eine Warteschlange zu je einem Operator gehört. Sie unterscheiden, ob die Warteschlangen interne, externe oder beide Eingaben erhalten. Hierbei sind interne Eingaben die von anderen Warteschlangen bzw. Operatoren erzeugte Tupel, externe Eingaben erfolgen durch die Eingangsströme. Das beschriebene Verfahren kann dabei je nach Eingabeart für jeden Operator bzw. für jede Warteschlange voraussagen, wie groß die Warteschlange ist und wie lange ein Tupel durchschnittlich an einem Operator zu einem bestimmten Zeitpunkt warten muss. Mit Hilfe dieser Werte werden Optimierungen vorgestellt. Der Ansatz ist, die vorausgesagte Berechnungszeit einer Anfrage mit der verlangten Deadline des Benutzers zu vergleichen. Erkennt man, dass die Anfrage zu spät für den Benutzer kommen wird, so räumt man dieser eine höhere Priorität beim Scheduling ein. Entsprechend erhält die Anfrage eine niedrigere Priorität, wenn man erkennt, dass sie vor dem einzuhaltenden Termin fertig wird. Danach sagt man erneut die Bearbeitungszeit voraus. Erkennt man weiter Abweichungen, so geht man davon aus, dass die Aufgabe nicht mit den zur Verfügung stehenden Ressourcen zu lösen ist und man benutzt Load Shedding, um die Überlast zu reduzieren. Ansonsten wird der Scheduling Plan so ausgeführt.

8 Zusammenfassung

Der Aspekt der Dienstgüte wird in den meisten existierenden DSMS von Haus aus überhaupt nicht unterstützt (z.B. STREAM oder Gigascope von AT&T). Zwar gibt es auch bei diesen Systemen ein Ressourcen Management, dieses hat aber den Fokus eher auf der Speichernutzung, Bandbreite des Netzwerks und Prozessorauslastung. Diese Maßnahmen sind dabei sehr allgemein gehalten, eine explizite Beachtung der Benutzer- bzw. Administratorbedürfnisse ist nicht vorgesehen. Eine Anpassung der Leistung ist aber in jedem Fall auch bei diesen Architekturen notwendig. Somit entscheidet das System – zumeist zufällig – welche Daten verworfen werden und somit nicht zwangsläufig im Sinne des Anwenders. Das führt dazu, dass solche Systeme in vielen Einsatzgebieten nicht benutzt werden können, da das Spektrum der Anforderungen sehr groß und die Möglichkeit der Anpassung des Systems sehr gering ist. Von Universalität, wie sie bei klassischen Datenbankmanagementsystemen existiert, kann also längst nicht gesprochen werden.
Dennoch gibt es viel versprechende Ansätze wie z.B. Aurora, bei dem Quality of Service ein zentraler Teil der Systemarchitektur darstellt. Aber auch das vorgestellte Konzeptsystem für STREAM geht in diese Richtung und setzt QoS-Elemente zur Sicherstellung der vom Benutzer geforderten Eigenschaften ein. Insgesamt kann man sagen, dass die Forschung in Bereich von DSMSen noch sehr jung ist, gerade im Bezug auf Quality of Service. Es wird sich zeigen, ob sich QoS-orientierte System, die mehr oder weniger universell einsetzbar sind, durchsetzten oder aber der Trend in Richtung spezialisierte DSMS geht, die jeweils nur sehr begrenzte Einsatzgebiete abdecken.

9 Literaturverzeichnis

[ABB+03] Arasu, A.; Babcock, B.; Babu, S.; Datar, M.; Rosenstein, J.; Ito, K.; Nishizawa, I.; Widom, J.
Query Processing, Resource Management, and Approximation in a Data Stream Management System.
In Proceedings of 1st CIDR Conference, January 2003.
www-db.cs.wisc.edu/cidr2003/program/p22.pdf

[ACC+03a] Abadi, D.; Carney, D.; Cetintemel, U.; Cherniack, M.; Convey, C.; Erwin, C.; Galvez, E.; Hatoun, M.; Hwang, J.; Maskey, A.; Rasin, A.; Singer, A.; Stonebraker, M.; Tatbul, N.; Zing, Y.; Yan, R.; Zdonik, S.
Aurora: A Data Stream Management System (demo description).
In Proceedings of the 2003 ACM SIGMOD Conference on Management of Data, San Diego, CA, 2003.
www.cs.brandeis.edu/~mfc/papers/AuroraDemo.pdf

[ACC+03b] Abadi, D.; Carney, D.; Cetintemel, U.; Cherniack, M.; Convey, C.; Lee, S.; Stonebraker, M.; Tatbul, N.; Zdonik, S.
Aurora: A New Model and Architecture for Data Stream Management.
In VLDB Journal, 2003.
www.cs.brandeis.edu/~mfc/papers/vldb095.pdf

[BBC+04] Balakrishnan, Hari; Balazinska, Magdalena; Carney, Don; Çetintemel, Uğur; Cherniack, Mitch; Convey, Christian; Galvez, Eddie; Salz, Jon; Stonebraker, Michael; Tatbul, Nesime; Tibbetts, Richard; Zdonik, Stan
Retrospective on Aurora.
VLDB Journal: Special Issue on Data Stream Processing, 2004
nms.lcs.mit.edu/~mbalazin/vldbj_retrospective.pdf

[BBD+02] Babcock, Brian; Babu, Shivnath; Datar, Mayur; Motwani, Rajeev; Widom, Jennifer.
Models and Issues in Data Stream Systems.
In Proceedings of 21st ACM Symposium on Principles of Database Systems (PODS 2002)
http://dbpubs.stanford.edu/pub/showDoc.Fulltext?lang=en&doc=2002-19&format=pdf&compression=&name=2002-19.pdf

[CCC+02] Carney, D.; Cetintemel, U.; Cherniack, M.; Convey, C.; Lee, S.; Seidman, G.; Stonebraker, M.; Tatbul, N.; Zdonik, S.
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, 2002.
http://www.cs.brown.edu/research/aurora/vldb02.pdf

[CCR+03] Carney, Don; Çetintemel, Uğur; Rasin, Alex; Zdonik, Stan; Cherniack, Mitch; Stonebraker, Mike.
Operator Scheduling in a Data Stream Manager.
In Proceedings of the 29th VLDB Conference, Berlin, Germany, 2003
www.cs.brown.edu/~dpc/publications/vldb2003.pdf

[JiCh03] Jiang, Qingchun; Chakravarthy, Sharma.
QoS Prediction for Continuous Queries over Data Streams.
CSE Technical Report CSE-2003-7, University of Texas at Arlington.
berlin.uta.edu/~qingchun/Papers/queryPlansTR.pdf

[TCZ+03a] Tatbul, Nesime; Cetintemel, Ugur; Zdonik, Stan; Cherniack, Mitch; Stonebraker, Michael.
Load Shedding on Data Streams.
In Proceedings of the Workshop on Management and Processing of Data Streams (MPDS 03), San Diego, CA, USA, June 8, 2003.
http://www.cs.brandeis.edu/~mfc/papers/mpds03b.pdf

[TCZ+03b] Tatbul, Nesime; Cetintemel, Ugur; Zdonik, Stan; Cherniack, Mitch; Stonebraker, Michael.
Load Shedding in a Data Stream Manager.
In Proceedings of the 29th International Conference on Very Large Data Bases (VLDB), September, 2003.
wwwdb.informatik.uni-rostock.de/ vldb2003/slides/S10F03.pdf

[TuXP04] Tu, Yi-Cheng; Xia, Yuni; Prabhakar, Sunil.
Quality of Service Adaptation in Data Stream Management Systems: A Control-Based Approach.
In Proceedings of the 30th VLDB Conference, Toronto, Canada, 2004
www.cs.purdue.edu/homes/tuyc/pub/draft/QoS_Control.ps