Technische Universität Kaiserslautern
Fachbereich Informatik

AG Datenbanken und Informationssysteme
Prof. Dr.-Ing. Dr. h. c. Theo Härder

Seminar: Data Streams

Multi-Query-Optimierung bei Datenströmen

Ausarbeitung von

Julia Thiele
j_thiele@informatik.uni-kl.de

Betreuer:

Prof. Dr.-Ing. Dr. h. c. Theo Härder

8. Juni 2005

1 Einleitung

Das Thema *Data Streams* befasst sich mit der Erfassung und Verarbeitung von Datenströmen.
Datenströme stellen Daten dar, die in schnellen und unterschiedlichen Zeitphasen auftreten. Des Weiteren können sie eine nicht berechenbare oder unendliche Länge aufweisen. Solche Datentypen treten in vielen Anwendungsbereichen auf, wie z. B. in Finanzwirtschaft, Web- oder Netzwerkanwendungen oder in Bereichen, in denen Sensordaten verarbeitet werden müssen.
Wenn diese Daten mit einem traditionellen *Datenbank-Management-System (DBMS)* verarbeitet werden sollen, führen die Merkmale der Datenströme zu fundamentalen Problemen. Diese treten dadurch auf, dass DBMS keine kontinuierlichen Anfragen unterstützen, welche für Datenstromverarbeitungen notwendig wären. Auch eine Approximation oder Adaption einzelner Datenstromsegmente kann durch traditionelle DBMS nicht erstellt werden, da diese nur über festgelegte Anfragepläne Daten auswerten und somit mit ungenauen Datensätzen nicht umgehen können.
Um diese Probleme lösen zu können, wurden *Datenstrom-Management-Systeme (DSMS)* entwickelt. Ein DSMS besteht aus einem *Datenstrommodell* und aus der Möglichkeit, Anfragen anhand von Datenströmen zu erstellen. Das Datenstrommodell unterscheidet sich folgendermaßen von konventionellen Relationsmodellen:
Die Datenelemente treten als Datenstrom auf und das System hat keinerlei Einwirkung auf die Ankunftsreihenfolge der einzelnen Elemente bzw. der verschiedenen Datenströme. Des Weiteren müssen die einzelnen Elemente nach der Verarbeitung entweder archiviert oder verworfen werden, da durch deren unbestimmte Länge nicht alle Elemente gesichert werden können.
Die Anfrageverarbeitung im DSMS erfolgt einerseits durch einmalige Anfragen oder andererseits durch kontinuierliche Anfragen. Einmalige Anfragen werden mit den, zu einem Zeitpunkt, vorhandenen Daten ausgewertet und das Ergebnis dieses "Schnappschusses" wird an den Anwender weitergeleitet. Mit diesen Anfrageformen können auch die traditionellen Anfragen verarbeitet werden. Dem gegenüber stellen kontinuierliche Anfragen persistente Anfragen dar, die regelmäßig anhand der ankommenden Datenströme ihre Auswertungen bearbeiten, ohne die Anfrage erneut zu stellen. Die Anfragen werden gespeichert und die Ergebnisse gegebenenfalls aktualisiert, sobald neue Daten eintreffen. Alternativ können durch die Anfrage selbst neue Datenströme als Ergebnismenge erzeugt werden. Durch diese Eigenschaften ist diese Form der Anfragen gut für das Internet geeignet [CDTW00].
Eine weitere Unterteilung der Anfragemöglichkeiten ist durch vordefinierte Anfragen und Ad-hoc-Anfragen möglich. Vordefinierte Anfragen sind im DSMS bereits vorhanden, bevor die Daten im System sind. Diese Anfragen sind zum Großteil kontinuierliche Anfragen, obwohl es auch möglich ist, einmalige Anfragen in dieser Art abzubilden. Ad-hoc-Anfragen werden erstellt, sobald die Daten zur Verfügung stehen. Durch diese Form der Anfrageerstellung wird die Konstruktion eines DSMS schwieriger, da für die korrekte Beantwortung der Anfrage Daten benötigt werden können, die zwar schon erhalten, aber aufgrund der Speicherkapazität schon wieder verworfen wurden [BBD+02].
Im Weiteren soll auf die effektive gemeinsame Verarbeitung mehrerer Anfragen in einem DSMS eingegangen werden, was den Bereich der Multi-Query-Optimierung beinhaltet.

2 Multi-Query-Optimierung

2.1 Problematik der Multi-Query-Optimierung

Das Problem der *Multi-Query-Optimierung* entsteht dadurch, dass man eine Menge von Anfragen, die gleiche Unteranfragen beinhalten, so effektiv wie möglich auswerten möchte [RSSB00]. Wenn

man beispielsweise Informationen aus Data-Warehouse-Daten erhalten möchte, kommt es häufig vor, dass Anfragemengen gleiche Unteranfragen enthalten. Dies lässt sich daraus ableiten, dass die Anfragen sehr oft auf die gleichen Sichten zugreifen, sodass diese wiederholt ausgeführt werden müssen. Ein weiterer Grund ist, dass viele Anfragen zur Verknüpfung von Tabellen die gleiche Beziehung ausnutzen, so dass *Joins* über die gleichen Tabellen immer wieder benötigt werden. Dieser Umstand führt zu gleichen Unteranfragen, obwohl die Anfragen an sich verschiedene Ergebnisse erzeugen.
Aus diesem Problem heraus lassen sich folgende Aufgaben der Multi-Query-Optimierung ableiten: Zunächst sollte eine Reduzierung der Auswertungskosten durch Zusammenfassen von Unteranfragen erreicht werden, da diese nicht mehr mehrfach, sondern nur noch einfach bearbeitet werden. Des Weiteren kann durch die Modifikation bzw. das Verbinden der Anfragen ein globaler optimaler Auswertungsplan für alle Anfragen erzeugt werden.

In den folgenden Abschnitten werden verschiedene Algorithmen vorgestellt, die zur Optimierung dieser Anfragemengen dienen können.
Die ersten vier Verfahren *Volcano*, *Volcano-SH*, *Volcano-RU* und die *Greedy-Strategie* basieren auf heuristischen Optimierungsalgorithmen. Hierbei soll entschieden werden, ob eine Unteranfrage bzw. deren Ergebnis materialisiert werden soll und somit nur einmal ausgeführt werden muss. Dabei ist noch zu bemerken, dass die *Greedy-Strategie* die Unteranfragen auswählt, die eine maximale Kostenreduktion für die materialisierten Anfrageergebnisse erreichen. Diese vier Verfahren basieren auf einer *AND-OR-GAG* Repräsentation, sodass alternative Anfragepläne leicht erkennbar werden. Ein *AND-OR-GAG* ist ein gerichteter azyklischer Graph, dessen Knoten in *AND*-Knoten und in *OR*-Knoten unterteilt werden können. Hierbei können *AND*-Knoten nur *OR*-Knoten als Kindknoten und *OR*-Knoten nur *AND*-Knoten als Kindknoten besitzen. *AND*-Knoten stellen algebraische Operationen dar, wie beispielsweise die *Join*- oder die *Select*-Operation. Somit werden diese Knoten als Operationsknoten bezeichnet. Dem gegenüber stellen *OR*-Knoten logische Ausdrücke, die die gleiche Ergebnismenge erzeugen, dar und werden deshalb Gleichheitsknoten genannt [RSSB00].
Die Operationsknoten werden im Graphen durch einen Kreis und die Gleichheitsknoten durch ein Rechteck gekennzeichnet. Bei einem expandiertem GAG handelt es sich um einen Graphen, in dem alle Möglichkeiten zur Erzeugung eines Ausdrucks aufgezeigt werden. Zum besseren Verständnis werden folgende Anfragen

1. A Join B
2. ($\sigma_{A<10}$(A)) Join B
3. ($\sigma_{A<5}$(A)) Join B

definiert und die entsprechenden expandierten GAG in Abbildung 1 dargestellt. Aufgrund der Übersichtlichkeit wurde auf Lösungen verzichtet, die auf der Kommutativitätseigenschaft basieren. Im Weiteren werden diese Anfragen und deren Graphen zur Verdeutlichung der Algorithmen wiederverwendet. Es ist darauf hinzuweisen, dass die Operationsknoten hier anstatt durch Kreise durch abgerundete Rechtecke dargestellt werden.
Darauf folgend wird das Projekt *Niagara* vorgestellt, das mit Hilfe einer Gruppierung der Anfragemenge das Problem der *Multi-Query-Optimerierung* zu lösen versucht.

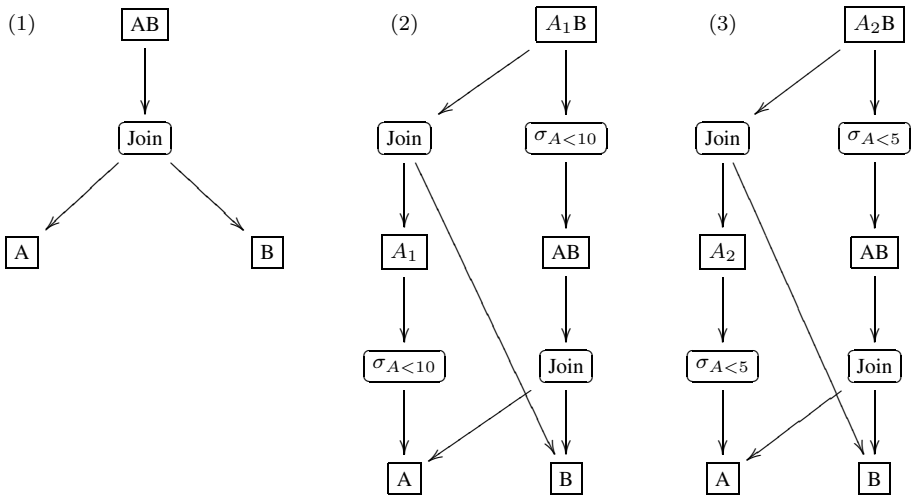

Abbildung 1: Expandierte GAG der Anfragen 1, 2 und 3

2.2 Der Volcano-Ansatz

Der *Volcano-Algorithmus* erarbeitet anhand gerichteter azyklischer Graphen eine Mengenauswahl der Anfragen, die aus Kostengründen materialisiert werden sollen. Hierzu wird der Graph mittels Tiefensuche durchlaufen. Dabei wird der lokal beste Plan rekursiv bestimmt, wobei allerdings auf gemeinsam nutzbare Teilausdrücke nicht geachtet wird. Um Kosten eines Knotens berechnen zu können, werden folgende Werte für Operations- und Gleichheitsknoten festgelegt:

$$cost(o) = Ausführungskosten(o) + \sum_{e_i \in children(o)} cost(e_i)$$

Diese Formel entspricht den Kosten eines Operationsknoten, wobei die lokalen Kosten und die Summe der eingehenden Kosten von Bedeutung sind. Die Kosten eines Gleichheitsknotens betragen:

$$cost(o) = min\{cost(o_i | o_i \in children(e))\}$$

Falls ein Knoten keine Kinder besitzt, so gilt $cost(o) = 0$. Dies ist beispielsweise der Fall, wenn ein Knoten eine Relation repräsentiert. Der beste Plan eines Gleichheitsknotens wird somit durch die Pläne für die Eingabeknoten und die geringsten Kosten der Kindknoten bestimmt [Sud00].
Damit keine Knoten doppelt besucht werden, speichert der Volcano-Algorithmus diese ab. Um weitere Kosten zu sparen, verwendet dieser Algorithmus das Verfahren *branch and bound pruning* [Sud00]. Bei diesem Verfahren werden Teile des Graphen nicht betrachtet, obwohl sie nicht untersucht wurden. Dies geschieht aufgrund einiger festgelegter Regeln, wie beispielswiese bei der Alpha-Beta-Suche.
Diese initialen Bewertungen des Graphen werden durch die Wiederverwendungskosten *reusecost* erweitert, die den Materialisierungsaufwand von Sichten angeben.
Um die endgültigen Kosten eines Knotens zu berechnen, wird dann letztendlich folgende Formel zur Berechnung gewählt:

$$cost(o) = Ausführungskosten(o) + \sum_{e_i \in children(o)} C(e_i)$$

mit $C(e_i) = cost(e_i)$, wenn $e_i \in M$, oder $min\{cost(o), reusecost(e_i)\}$, wenn $e_i \in M$, wobei M die Menge aller materialisierten Knoten darstellt.
Zur Verdeutlichung dieser Vorgehensweise wird dieses Verfahren anhand der Anfragenbeispiele vorgestellt. Dabei ändert sich der Graph der Anfrage 1 nicht, da dieser nur einen Erzeugungsweg enthält. Die Änderungen der Anfragen 2 und 3 werden in Abbildung 2 verdeutlicht. Hierbei entsprechen die Graphen 2' und 3' den besten Plänen des Volcano-Algorithmus, die aufgrund der berechneten Kosten (K) erzeugt wurden. Die jeweiligen Kosten der Operationsknoten wurden an die darauf weisenden Pfeile geschrieben. Hierbei handelt es sich um fiktive Zahlen, so dass ein möglichst ideales Beispiel erzeugt werden konnte.

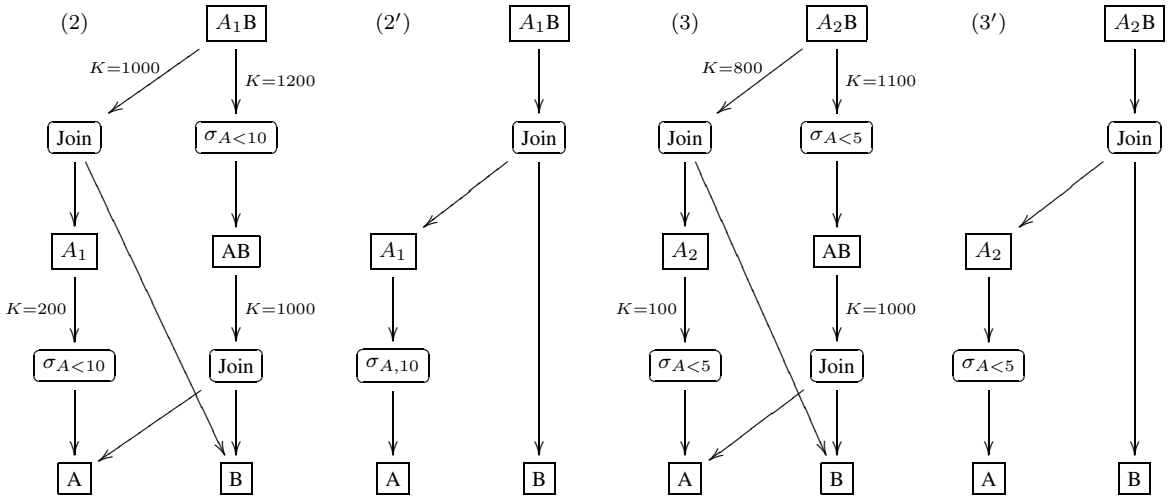

Abbildung 2: Expandierte GAG mit Kostenangaben und beste Pläne der Anfragen 2 und 3

2.3 Optimierung Volcano-SH

Der *Volcano-SH-Algorithmus* ist eine Erweiterung des Volcano-Algorithmus. Grundlage für die Vorgehensweise bildet der berechnete beste Plan des Volcano-Algorithmus. Da dieser Plan noch gleiche Unterausdrücke enthalten kann, entscheidet der Algorithmus, ob Teilausdrücke zur Wiederverwendung materialisiert werden sollen oder nicht. Durch die Sichtenmaterialisierung können diese Ergebnisse mehrfach verwendet werden und müssen nicht erneut ausgeführt werden, so dass die Ausführungskosten sinken. Hierzu muss zunächst überprüft werden, ob eine solche Materialisierung überhaupt sinnvoll ist. Um dieses Problem zu lösen, kommen einige Möglichkeiten in Frage. Die erste ist eine naive Herangehensweise, indem man folgende Formel zur Materialisierungsentscheidung heranzieht:

$$cost(e) + matcost(e) + reusecost(e) * (numuses(e) - 1) < numuses(e) * cost(e)$$

bzw.

$$matcost(e)/(numuses(e) - 1) + reusecost(e) < cost(e)$$

Die Funktion $matcost$ steht die für die Materialisierung anfallenden Kosten und $reusecost$ entsprechen den Kosten, die zur Wiederverwendung dieser Ergebnisse anfallen. $numuses$ repräsentiert die Anzahl der Wiederverwendungen, d. h. die Anzahl gleicher Teilausdrücke. Problematisch bei diesem Ansatz ist, dass $numuses$ und $cost$ jeweils von den zuvor materialisierten Knoten abhängig sind und somit nicht ohne weiteres berechnet werden können [RSSB00].
Ein weiterer sehr naiver Ansatz wäre, anhand des Graphen die Kosten zu berechnen und davon auszugehen, dass alle Teilausdrücke bei ihrem ersten Auftreten materialisiert werden. Allerdings ist diese Entscheidungsmöglichkeit in der Praxis nicht sinnvoll umzusetzen, da sie aufgrund der exponentiellen Knotenanzahl im besten Plan von Volcano sehr teuer ist.

Procedure VOLCANO-SH(P)

input: consolidated Volcano best plan P for virtual root of DAG
output: Set of nodes to materialize M, and the corresponding best plan P
global variable: M, the set of nodes chosen to be materialized

$M = \{\}$
Perform prepass on P to introduce subsumption derivations
Let $C_{root} = COMPUTEMATSET(root)$
Set $C_{root} = C_{root} + \sum_{d \in M}(cost(d) + matcost(d))$
Undo all subsumption derivations on P where the subsumption node is not chosen to be materialized.
return(M,P)

Procedure COMPUTEMATSET(e)

if cost(e) is already memorized, return cost(e)
Let operator o_e be the child of $e \in P$
For each input equivalence node e_i of o_e

// returns computation cost of e_i
Let $C_i = COMPUTEMATSET(e_i)$
if e_i is materialized, let $C_i = reusecost(e_i)$

Compute $cost(e)$ = cost of operation $o_e + \sum_i C_i$
if $(matcost(e)/(numuses^-(e) - 1) + reusecost(e) < cost(e))$

if (e is not introduced by a subsumption derivation)

add e to M //Decide to materialize e

else if $cost(e) + matcost(e) + reusecost(e) * (numuses^-(e) - 1)$ is less than savings to parents of e due to introducing materialized e

add e to M //Decide to materialize e

Memorize and return cost(e)

Abbildung 3: Volcano-SH-Algorithmus

Im Folgenden wird eine Variante beschrieben, welche anhand der *Anzahl der Vaterknoten im Graphen* eine Materialisierungsentscheidung trifft.
Der im Algorithmus (siehe Abbildung 3) realisierte Ansatz durchläuft den Baum von den Blattknoten hinauf zur Wurzel. Bei jedem Knoten wird dann entschieden, ob dieser materialisiert werden soll oder nicht. Falls eine Entscheidung zur Materialisierung getroffen wurde, so ist gleichzeitig auch für alle folgenden Knoten die Entscheidung getroffen. Allerdings ist für die Materialisierungsentscheidung

eines Knotens e immer noch $numuses(e)$ relevant, was weiterhin von seinen bisher materialisierten Vaterknoten abhängig ist. Um dieses Problem zu lösen, verwendet der Algorithmus den Wert $numuses^-(e)$, der die Anzahl der Vorfahrenknotens e im besten Plan von Volcano angibt. Dadurch ergibt sich folgende Formel zur Materialisierungsentscheidung:

$$matcost(e)/(numuses^-(e) - 1) + reusecost(e) < cost(e)$$

Da hier statt $numuses$ aus der naiven Ansatzweise $numuses^-$ verwendet wird, ist bei der Entscheidung zur Materialisierung eines Knotens gesichert, dass damit geringere Kosten auftreten.
Damit dieses Verhalten verständlicher ist, wird der Algorithmus anhand des Beispiels verdeutlicht. Zunächst erhält der Algorithmus einen zusammengesetzten Plan aller Anfragen, wobei es sich jeweils um den besten Plan des Volcano-Algorithmus handelt (siehe Abbildung 4). In diesem Graphen entspricht X der virtuellen Wurzel.

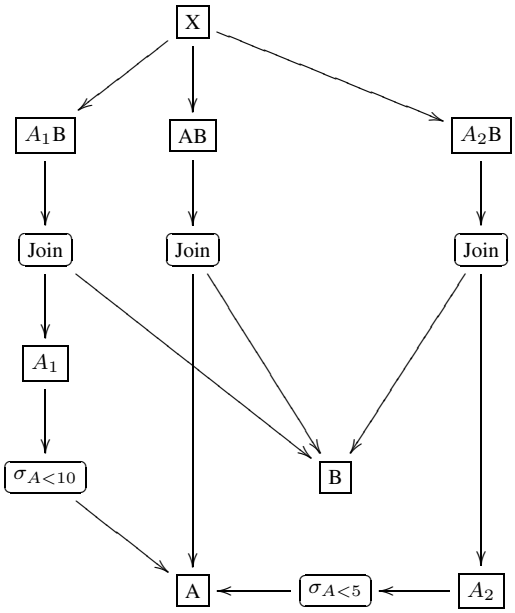

Abbildung 4: Zusammengesetzter GAG mit den jeweils besten Plänen der Anfragen

Im ersten Schritt wird der Graph auf gleiche Unterausdrücke untersucht. Dies ist in diesem Beispiel bei $\sigma_{A<5}(A)$ und $\sigma_{A<10}(A)$ der Fall. Hier wird dann $\sigma_{A<5}(A)$ durch $\sigma_{A<5}(\sigma_{A<10}(A))$ ersetzt. Wird am Ende des Algorithmus festgestellt, dass $\sigma_{A<10}(A)$ nicht materialisiert wurde, so wird die Umordnung des Graphen wieder rückgängig gemacht. In Abbildung 5 wird dieser neue Graph gezeigt. Außerdem wurden die auf die Operationsknoten zeigenden Pfeile mit den jeweiligen Kosten beschriftet. Hierbei ist zu beachten, dass der Knoten A_1 aufgrund gewählter Zahlen materialisert wird (doppelter Rahmen) und dadurch die Kosten der Vorgängerknoten geringer sind. In diesem Beispiel wird lediglich die Sicht $\sigma_{A<10}(A)$ materialisiert und der beste Ausführungsplan ist der gezeigte Graph aus Abbildung 5.
Somit lässt sich abschließend sagen, dass diese Erweiterung die beste Teilmenge der Knoten ausgibt, die vorzugsweise materialisiert werden sollen. Der Nachteil des Volcano-Algorithmus, dass frühere

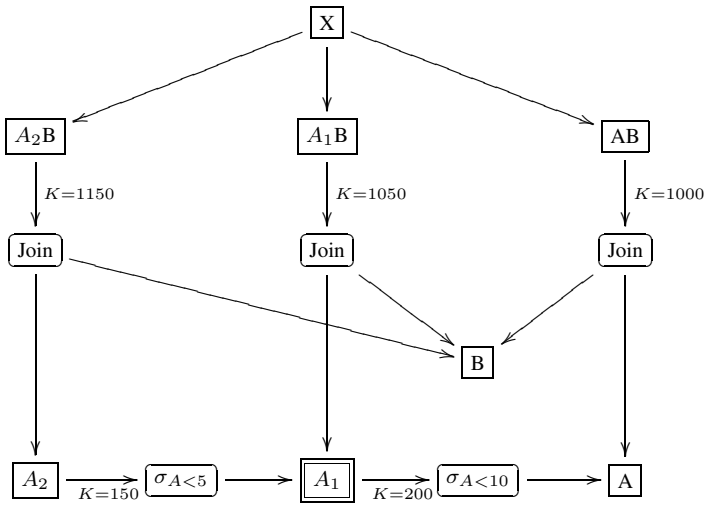

Abbildung 5: Auswertungsplan des Algorithmus-SH mit Kostenangaben

Materialisierungsentscheidungen in die Ausführungskosten mit einbezogen werden, wurde in dieser Optimierung nicht behoben [RSSB00].

2.4 Optimierung Volcano-RU

Da es mit dem *Volcano-SH-Algorithmus* immer noch Teilausdrücke gibt, die nicht zusammengefasst wurden, befasst sich der *Volcano-RU-Algorithmus* hauptsächlich mit diesem Problem. Hierbei werden solche Teilausdrücke untersucht, die scheinbar nicht zusammengefasst werden können; nach einer Umsortierung der Tabellen ist dies doch möglich. Ein Beispiel hierfür wäre: *(R Join S) Join P* und *(R Join T) Join S*. Diese beiden Ausdrücke könnten sich die Sicht *R Join S* teilen. Somit ist die Hauptaufgabe des Volcano-RU-Algorithmus anhand einer Folge von Anfragen gemeinsam nutzbare Teilausdrücke zu erkennen. Im folgenden Abschnitt wird die Vorgehensweise genauer beschrieben:

Der in Abbildung 6 dargestellte Algorithmus erhält eine Sequenz von Anfragen $Q_1 \dots Q_k$, die zusammen bearbeitet werden sollen. Nachdem diese Anfragen optimiert wurden, werden die zugehörigen Gleichheitsknoten im GAG für den besten Ausführungsplan P als Kandidaten zur eventuellen Wiederverwendung markiert. Eine weitere Überprüfung, ob der Knoten e auch dann noch sinnvoll materialisiert ist, wenn er ein weiteres Mal verwendet wird, entscheidet darüber, ob dieser Knoten e in die Menge N der Materialisierungskandidaten aufgenommen wird. Wurde der Knoten in N aufgenommen, so wird bei der nächsten Anfrage die Annahme getroffen, dass der Knoten e bereits materialisiert ist. Im Falle der Materialisierung eines Knotens ist zu beachten, dass die Materialisierungskosten nicht zu den weiteren Wiederverwendungskosten addiert werden.
Nachdem alle Anfragen optimiert wurden, beginnt die zweite Phase des Algorithmus. Hier wird unter diesen neuen Voraussetzungen der *Volcano-SH-Algorithmus* (siehe Abbildung 3) ausgeführt. Dieser Aufruf ist notwendig, da die erste Phase des Volcano-RU-Algorithmus nicht auf gemeinsame Teilausdrücke innerhalb einer Anfrage überprüft. Dies ergibt sich aus der Tatsache heraus, dass die Knoten, die in der Menge N enthalten sind, nicht automatisch materialisiert werden, sondern lediglich Kan-

```
Procedure VOLCANO-RU

    Input: Expanded DAG on queries Q_1...Q_k(including subsumptions derivations)
    Output: Set of nodes to materialize M, and the corresponding best plan P

        N = {} //Set of potentially materialized nodes
        for each equivalence node e, set count[e]=0
        for i = 1 to k

            Compute P_i, the best plan for Q_i, using the Volcano, assuming nodes in N are materialized
            for every equivalence node in P_i

                set count[e] = count[e] + 1
                if (cost(e) + matcost(e) + count[e] * reusecost(e) < (count[e] + 1) * cost(e))
                //Worth materializing if used once more

                    add e to set N

        Combine P_1...P_k to get a single DAG-structured plan P
        (M,P) = VOLCANO-SH //Volcano-SH makes finel materialization decision
```

Abbildung 6: Volcano-RU-Algorithmus

didaten darstellen. Ob es wirklich sinnvoll ist, diese Knoten zu materialisieren, ergibt sich erst nach der Anwendung des *Volcano-SH-Algorithmus*. Somit ist die Hauptaufgabe dieses Algorithmus, die Anfragen innerhalb der Sequenz so zu optimieren, dass bei der Sequenzausführung Teilausdrücke gemeinsam verwendet werden können.
Diese Vorgehensweise kann ebenfalls am bisherigen Beispiel erklärt werden. Voraussetzung zur Ausführung des Algorithmus ist der zusammengefügte expandierte GAG aller Anfragen (siehe Abbildung 7). Die Kostenangaben werden wie in den letzten Abbildungen verwendet. Unter der Voraussetzung, dass hier die Werte so gewählt sind, wird die Sicht *A Join B* als Kandidat zur Materialisierung ausgewählt und dadurch ergibt sich ein anderer optimaler Plan als bei der alleinigen Ausführung des Volcano-SH-Algorithmus. Dieser ist ebenfalls in der Abbildung 7 unter GAG-RU dargestellt.
Dieser optimale Ausführungsplan wird im zweiten Schritt mittels des Volcano-SH-Algorthmus bearbeitet, da Teilausdrücke im ersten Schritt nicht beachtet werden. Dies wären im Beispiel die Selektionen, die wie auch in Abbildung 5 behandelt werden. Werden die Kosten entsprechend gewählt, so werden die Sichten *A Join B* und $\sigma_{A<10}(A)$ materialisiert.
Bei diesem Algorithmus ist zu beachten, dass die Reihenfolge der einzelnen Anfragen stark das Optimierungsergebnis beeinflusst. Würde man die Reihenfolge der Anfragen ändern, so kann sich daraus ein ganz anderes Ergebnis herleiten. In dieser Implementierung des Algorithmus wird die übergebene Reihenfolge und deren Umkehrung betrachtet, wobei dann die Variante mit den geringeren Kosten verwendet wird. Dabei wird der GAG aber nur ein einziges Mal erzeugt, so dass die Kosten zur Auswertung der beiden Varianten sehr gering sind. Es ist zwar auch eine zufällige Reihenfolge der Anfragen zur Optimierungsübergabe möglich, diese führt allerdings zu höheren Kosten [RSSB00].

2.5 Der Greedy-Ansatz

Der *Greedy-Algorithmus* stellt eine Alternative zum *Volcano-Algorithmus* mit seinen Optimierungen dar. Der Greedy-Ansatz beschreibt eine *gierige* Herangehensweise an die Problemstellung. Der Ansatz zeichnet sich darin aus, dass er in jedem Teilschritt immer den Folgeschritt auswählt, der den höchsten Gewinn verspricht. Um einen solchen Folgezustand bestimmen zu können, werden Bewertungsstrategien verwendet. Der Greedy-Algorithmus findet immer dann eine optimale Lösung, wenn

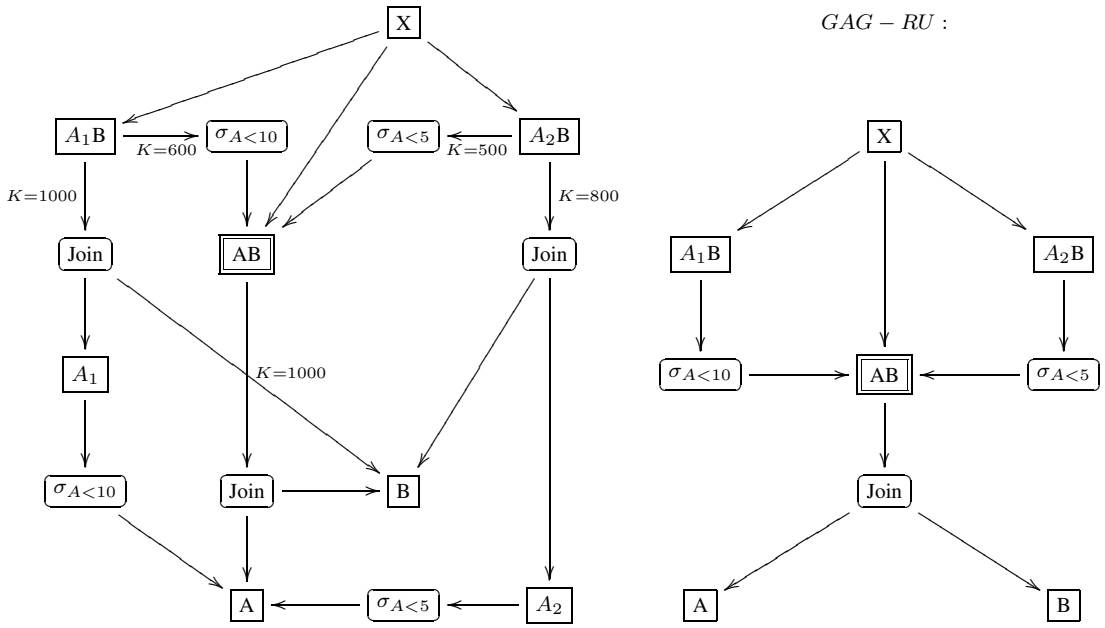

Abbildung 7: Expandierter GAG mit Kostenangaben und resultierender GAG (GAG-RU)

die Problemstellung die Struktur eines Matroiden besitzt.
In diesem Abschnitt soll vor allem darauf eingegangen werden, in welcher Weise man diesen Algorithmus effizient implementieren kann. Hierzu wird zunächst eine naive Variante der Greedy-Methode beschrieben. Diese Variante basiert darauf, dass der Algorithmus eine Knotenanzahl S zur Materialisierung auswählt und daraufhin einen optimalen Ausführungsplan mit diesen materialisierten Knoten berechnet. Dies wird auf einer gewissen Anzahl an Knotenmengen wiederholt, so dass die beste Menge der materialisierte Knoten gefunden werden kann. Voraussetzung zur Algorithmusausführung ist, dass es im GAG eine virtuelle Wurzel gibt. Diese erhält als Eingabe die Anfragefolge, die mittels des *no-op*-logischen Operators bearbeitet werden kann. Außerdem charakterisiert eine sogenannte Funktion $bestcost(Q, S)$ die Kosten, die für den optimalen Plan von Wurzel Q mit den in S materialisierten Knoten anfallen. Diese Kosten lassen sich in die Materialisierungs- und in die Ausführungskosten aufteilen. Zur Berechnung dieser Kosten kann wie bei der Berechnung der Kosten im Volcano-SH-Algorithmus verfahren werden. Durch das Iterieren der verschiedenen Materialisierungsmöglichkeiten der einzelnen Anfragen lässt sich leicht erkennen, dass in dieser Form der Aufwand exponentiell ist und somit nicht praxisrelevant.
Aus diesen Gründen wurde folgende Variante zur Implementierung verwendet (siehe Abbildung 8). Hierbei wird die optimale Menge S approximativ durch Konstruktion eines Knotens pro Zeiteinheit bestimmt. Der Algorithmus aus Abbildung 8 wählt iterativ Knoten zur Materialisierung aus. In jeder Iteration wird genau der Knoten ausgewählt, der den höchsten Kostengewinn verspricht; er wird dann in die Menge X aufgenommen. Leider hat auch diese Variante ihre Nachteile. Wenn die Menge aller Gleichheitsknoten Y im GAG sehr groß ist und deshalb die Funktion $bestcost$ oft aufgerufen werden muss, stellt dies einen erheblichen Aufwand dar. Aufgrund dieses Nachteils sollen im folgenden drei wichtige und neue Optimierungsvorschläge vorgestellt werden, die den Algorithmus aus Abbildung 8

Procedure GREEDY

Input: Expanded DAG for the consolidated input query Q
Output: Set of nodes to materialize and the corresponding best plan
$X = \{\}$
Y = set of equivalence nodes in the DAG
while ($Y! = \{\}$)

L1: Pick the node $x \in Y$ with the smallest value for $bestcost(Q, x \cup X)$
if ($bestcost(Q, x \cup X) < bestcost(Q, X)$)

$Y = Y - x$; $X = X \cup x$

else $Y = \{\}$ //benefit<0, so break out of loop

return X

Abbildung 8: Greedy-Algorithmus

noch effizienter machen [RSSB00]:

- Da die materialisierten Knoten des optimalen Ausführungsplanes die Knoten sind, die durch mehrere Anfragen gemeinsam genutzt werden, wäre es sinnvoll, die Menge Y direkt mit diesen Knoten zu initialisieren. Diese Knoten werden dann als *teilbare Knoten* bezeichnet.
  Diese teilbaren Knoten werden anhand eines Teilbarkeitgrades festgelegt. Hierzu gibt es folgende Definitionen, wobei z den zu untersuchenden Knoten und x die Wurzel des Untergraphen entsprechen:
  Für den Operationsknoten gilt:

$$E[x][z] = \sum_{y \in children(x)} E[y][z]$$

  Für den Gleichheitsknoten gilt:

$$E[x][z] = Max\{E[y][z] | y \in children(x)\}$$

  Wenn ein Knoten den Teilbarkeitsgrad 1 besitzt, so kann dieser Knoten nicht im GAG gemeinsam genutzt werden. Ist der Teilbarkeitsgrad > 1, so kann der Knoten gemeinsam genutzt werden und wird dann auch als teilbarer Knoten bezeichnet. Die Zeitkomplexität zur Berechnung einer Zeile $E[x]$ ist proportional zur Anzahl der Einträge ungleich Null in der Menge E. Im schlechtesten Fall entspricht dies der Anzahl der Knoten im GAG im Quadrat. Trotzdem ist die Berechnung der Teilbarkeit eines Knotens in der Praxis effizient.

- Eine weitere Optimierung ist im Bereich des Funktionsaufrufes $bestcost$ bei der Marke $L1$ in Abbildung 8 möglich. Da dieser Funktionsaufruf häufig mit gleichen Parametern abläuft und dabei lediglich der zweiten Parameter minimal verändert wird, kann dieser Aufruf durch einen Algorithmus, der inkrementell die Kosten aktualisiert, optimiert werden. Dieser Algorithmus verwertet die bisherigen Kostenberechnungen, um einen neuen Plan zu berechnen. Hierzu wurde der Algorithmus *UpdateCost* (siehe Abbildung 9) zur erneuten Berechnung der Kosten entwickelt.

  Dabei repräsentiert S die Menge aller Knoten, die zu einem Zeitpunkt gemeinsame Teilausdrücke aufweisen. Da diese Knotenmenge sich während der Kostenberechnung ändert, werden

Function UpdateCost(S, S')

//PropHeap is a priority (initially empty), containing equivalence nodes
//ordered by their topological sort number
while (PropHeap is not empty)

N = equivalence node with minimum topological sort number in PropHeap
Remove N from PropHeap
oldCost = old value of cost(N)
// children(N) are operation nodes
$cost(N) = Min\{cost(p)|p \in children(N)\}$
if ($cost(N)$! = oldCost or $N \in (S - S')$ or $N \in (S' - S)$)

for every parent operation node p of N
$cost(p)$ =cost of executing operation $p + \sum_{c \in children(p)} C(c)$
where $C(c) = cost(c)$ if $c \notin S'$ and $C(c) = min\{reusecost(c), cost(c)\}ifc \in S'$
add p's parent equivalence node to PropHeap if not already present

$TotalCost = compcost(root) + \sum_{s \in S'}(cost(s) = matcost(s))$

Abbildung 9: UpdateCost-Algorithmus

diese Änderungen in S' gespeichert, so dass hier die neuen gemeinsam genutzten Knoten enthalten sind. Der Algorithmus beginnt mit den Knoten, die von der Menge S zur Menge S' gewechselt haben. Anhand dieser Knoten werden die Kosten an die Vaterknoten weitergeleitet, die ihrerseits ebenfalls geänderten Kosten weiterreichen. Letztlich werden die Gesamtkosten dadurch berechnet, dass die Berechnungs- und die Materialisierungskosten von allen Knoten in S' addiert werden. Um einen Mehrfachbesuch eines Knotens zu verhindern, werden die Knoten nach einer topologischen Sortierreihenfolge besucht.

- Die dritte Optimierungsvariante wird *Monotonie-Heuristik* genannt. Hierbei wird verhindert, dass $bestcost$ für jedes $x \in Y$ an der Marke $L1$ aus Abbildung 8 ausgeführt wird. Es wird der kleinste Wert der $bestcost$ auf effiziente Art und Weise berechnet.
  Die Monotonie-Heuristik bedient sich dafür einer Hilfsfunktion

$$benefit(x, X) = bestcost(Q, X) - bestcost(Q, x \cup X).$$

  Diese maximiert genau dann ihren Nutzen, wenn $bestcost$ minimal ist. Des Weiteren enthält sie die Eigenschaft der Monotonie, was in diesem Zusammenhang bedeutet, dass der Nutzen nicht mehr ansteigt als die Anzahl der materialisierten Knoten. Somit gilt:
  $benefit$ ist monoton, wenn $\forall X \supseteq Y, benefit(x, X) \leq benefit(x, Y)$.
  Nachdem für alle Knoten der Nutzen berechnet wurde, werden alle Knoten nach diesem Kriterium sortiert und in einem Heap C abgelegt. Im nächsten Schritt wird der erste Knoten n aus dem Heap C markiert, der Nutzen erneut berechnet und die Liste im Heap wird dementsprechend neu sortiert. Ist nach diesem Durchlauf der erste Knoten wieder der markierte, so wird dieser aus dem Heap C entfernt, materialisiert und in der Menge X aufgenommen, da er den maximalen Grad an Teilbarkeit aufweist. Ansonsten wird der erste Knoten wieder markiert und es wird wiederum der Nutzen berechnet.
  Bleibt die Monotonieeigenschaft durch dieses Verfahren erhalten, so ist dieser Knoten n letztendlich der Knoten mit dem maximalen Nutzen. In diesem Fall kann die optimale Laufzeit bis zu 90% reduziert werden [Sud00].

2.6 Das Projekt Niagara

In diesem Abschnitt wird das Projekt *Niagara* vorgestellt. Hier wurde ein System entwickelt, das verteile Anfragen anhand von XML-Datensätzen mittels einer eigenen Anfragesprache *XML-QL* in einem verteilten DBMS verarbeiten kann. Mit diesem System soll das Ziel verfolgt werden, dass eine große Benutzeranzahl ihre kontinuierlichen Anfragen mittels einer *High-Level-Anfragesprache* erstellen und vom System ausführen lassen kann.
Unter der Annahme, dass viele dieser Anfragen sehr ähnlich zueinander sein werden, wurde die Idee der Anfragengruppierung in Betracht gezogen. Diese Gruppenbildung führt zu Berechnungen, die gemeinsam ausgeführt und genutzt werden können. Des Weiteren können die gemeinsamen Ausführungspläne so gespeichert werden, dass hierdurch eine I/O-Kostenreduktion im Vergleich zu den Ausführungskosten der einzelnen Anfragen erreicht werden kann. Ein weiterer Vorteil dieser Anfragegruppierung liegt bei der Überprüfung der Anfragebedingungen. Hier muss nicht mehr für jede Anfrage einzeln die Bedingung überprüft werden, sondern dies geschieht für die ganze Gruppe, so dass auch hier proportional weniger Aufwand anfällt. Ein Nachteil an diesem Ansatz besteht darin, dass das Finden des optimalen Plans nur für eine kleine Anzahl an gleichartigen Anfragen geeignet ist. Diese Bedingung trifft nicht auf die Bearbeitung von kontinuierlichen Anfragen zu, so dass dieser Ansatz in dieser Form nicht angewendet werden kann. Außerdem ist die Gruppierungsmöglichkeit nicht geeignet für die Anwendung im Internet, da hier Anfragen dynamisch hinzugefügt oder gelöscht werden können.
Aus diesem Grund wurde eine andere Herangehensweise an diesen Ansatz gewählt, der die Anfragen anhand ihrer Signatur gruppiert. Die Signaturvergabe erfolgt mit Hilfe von Ausdruckssignaturen. Wenn eine neue Anfrage in das System kommt, wird diese anhand ihrer Signatur einer existierenden Gruppe zugeordnet. Wird keine Gruppe mit gleicher Signatur gefunden, so wird eine neue Gruppe erzeugt. Durch diese Vorgehensweise muss keine Umgruppierung aller Anfragen erfolgen. Zur Anfragebearbeitung wurde ein *inkrementelles Gruppenoptimierungsschema* entwickelt, das auf einem Anfragezerlegungsschema aufbaut. Im weiteren Algorithmusverlauf wird der Anfragebaum analysiert. Durch das Gruppenoptimierungsschema werden die Anfragen in viele kleinere Anfragen umgewandelt, so dass diese mit der gleichen Technik, wie auch benutzerdefinierte Anfragen, bearbeitet werden können. Einen großen Vorteil stellt bei diesem Ansatz dar, dass er, bis auf ein paar kleinere Änderungen, genauso implementiert werden kann wie allgemeine Anfragemaschinen.
Da kontinuierliche Anfragen in zwei weitere Kategorien unterteilt werden können, war es auch wichtig, dass das System diese unterstützt. Bei diesen beiden Kategorien handelt es ich um *änderungsbasierte* und um *zeitbasierte Anfragen*, die aufgrund ihrer Ausführungszeitpunkte unterschieden wurden. Änderungsbasierte Anfragen werden genau dann ausgeführt, wenn neue relevante Daten im System ankommen und haben somit eine bessere Antwortzeit. Wenn man allerdings diese direkten Antworten nicht benötigt, verbraucht dieser Anfragetyp viel Systemressourcen, die anderweitig besser verwendet werden könnten. Dem gegenüber stehen die zeitbasierten Anfragen, die in einem spezifizierten Zeitintervall ausgeführt werden und damit effizienter in ihrer Verwendung sind. Dadurch werden die Anfragen besser skalierbar, aber auch schwieriger zu gruppieren [CDTW00].

2.7 Das System NiagaraCQ

NiagaraCQ ist ein Netzdaten-Management-System, das für das *Niagara-Projekt* entwickelt wurde. Dieses System unterstützt skalierbare kontinuierliche Anfrageprozesse über verteilten XML-Dateien, die durch das inkrementelle Gruppenoptimierungsschema erzeugt wurden.

Diese Verwirklichung der Grundidee des Projektes wurde anhand folgender Techniken skalierbar und effizient entwickelt: Indem nur die geänderten Passagen der aktualisierten XML-Dateien betrachtet und inkrementell ausgewertet werden, lassen sich die Ausführungskosten senken. Außerdem wird verhindert, dass Anfragen erneut ausgeführt werden, obwohl sich keine Änderungen ergeben haben. Dadurch wird erreicht, dass dem Benutzer nur neue Ergebnisse vom System geliefert werden, anstatt wiederholt alte. Da die Daten nicht alle auf der Festplatte gespeichert werden können und die System-Performanz sinken würde, wurden Caching-Verfahren mit begrenzten Speicherkapazitäten eingesetzt, um diesem Problem entgegenzuwirken. Des Weiteren kann das System Datenänderungen mittels *Push*- und *Pull*-Modellen in heterogenen Datenquellen erkennen.
Im Weiterem soll speziell auf die NiagaraCQ-Sprache und auf die grundlegenden Schritte der Gruppenoptimerungsstrategie eingegangen werden.

2.7.1 NiagaraCQ-Sprache

Die NiagaraCQ-Sprache ist eine einfache Kommandosprache, anhand derer man kontinuierliche Anfragen erstellen oder löschen kann.

```
CREATE  CQ_name
XML-QL  query
DO  action
{START start_time} {EVERY time_interval} {EXPIRE expiration_time}

DELETE CQ_name
```

Abbildung 10: Create- und Delete-Befehl in NiagaraCQ

In Abbildung 10 werden die Befehle zum Erstellen und Löschen der Anfragen spezifiziert. Bei dem Create-Befehl sieht man, dass man diese Anfragesprache mit der Anfragesprache XML-QL kombinieren und zusätzlich noch Zeitangaben einbringen kann. Die Anfrage wird ab dem Startzeitpunkt ausgeführt. Wird ein Zeitintervall ungleich Null angegeben, so werden diese Anfragen als zeitbasierte Anfragen, andernfalls als änderungsbasierte Anfragen verarbeitet. Nachdem ihre Gültigkeitsdauer (expiration-time) abgelaufen ist, werden diese Anfragen automatisch vom System wieder gelöscht [CDTW00].

2.7.2 Gruppenoptimierungsstrategie von NiagaraCQ

Die in diesem System angewendete Strategie der inkrementellen Gruppenoptimierung kann eine große Anzahl an Anfragen sehr gut verarbeiten. Die Gruppenerzeugung geschieht aufgrund der Anfragesignaturen, so dass die Anfragen innerhalb einer Gruppe eine ähnliche Struktur aufweisen. Innerhalb dieser Gruppen kann somit ein gemeinsamer Anfrageanteil ausgewählt und bearbeitet werden. Diese Möglichkeit der gemeinsamen Bearbeitung der Teilanfragen erlaubt es, den einzelnen Anfragen das Teilanfragenergebnis aus der Ausführung des Gruppenplans zu verwenden, ohne ein weiteres Ausführen dieser Teilanfrage notwendig zu machen. Wenn eine neue Anfrage im System hinzukommt, so wird sie zunächst einer Gruppe zugewiesen. Dieses dynamische Hinzufügen und Löschen einzelner Anfragen hat allerdings auch zur Folge, dass das Gruppenergebnis sich verschlechtert, so dass hier ein dynamisches Umgruppieren durchaus notwendig wäre und stellt zukünftige Arbeit dar. Dies ist im System noch nicht implementiert. Somit stellt diese Strategie das Resultat in suboptimalen Gruppen dar, reduziert aber trotz allem die Kosten der Gruppenoptimierung entscheidend. Des

Weiteren ist diese Strategie in einer dynamischen Umgebung sehr skalierbar [CDTW00].

Ausdruckssignatur
Um überhaupt Gruppen erzeugen und dann die Anfragen gruppieren zu können, müssen die Anfragen, wie schon erwähnt, eine Signatur zugewiesen bekommen. Dies geschieht mit Hilfe von *Ausdruckssignaturen*, wie sie in Abbildung 12 dargestellt ist.

```
Where <Quote><Quote><Symbol>INTC</>
</></>element_as $g
in "http://www.cs.wisc.edu/db/quotes.xml"
construct $g

Where <Quote><Quote><Symbol>MSFT</>
</></>element_as $g
in "http://www.cs.wisc.edu/db/quotes.xml"
construct $g
```

Abbildung 11: XML-QL-Anfragebeispiele

```
Quotes.Quote.Symbol in quotes.xml = constant
```

Abbildung 12: Ausdruckssignatur der Anfragen aus Abbildung 11

Beispielsweise erfolgt die Signatur zu einem Selektionsprädikat, indem die Konstanten dieses Prädikates durch Variablen ersetzt werden. Zunächst wird ein Anfrageplan mittels des Niagara-Anfrage-Parsers generiert. Der untere Teil des Plans entspricht dabei der Ausdruckssignatur der Anfrage. Nachdem die Anfrage geparst wurde, wird ein neuer Operator *Trigger-Action* am Kopf der Signatur hinzugefügt. Diese Signatur erlaubt es, Anfragen gleicher syntaktischer Struktur zusammenzufassen. Dies wird in Abbildung 13 dargestellt.

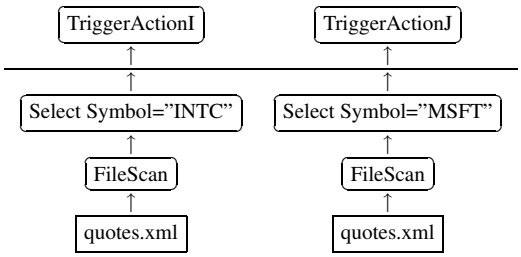

Abbildung 13: Anfrageplan der Anfragen aus Abbildung 11

Gruppeneigenschaften
Nachdem beschrieben wurde, wie die Anfragen in die Gruppen sortiert werden können, wird nun auf die Struktur der Gruppen an sich eingegangen. Eine Gruppe besteht aus drei Teilen: der Gruppensignatur, der konstanten Gruppentabelle und dem Gruppenplan. Die Gruppensignatur entspricht dem gemeinsamen Teil der Ausdruckssignatur aller Anfragen in dieser Gruppe. In der konstanten Gruppentabelle werden die Signaturen aller Anfragen in einer XML-Datei gespeichert. Der Gruppenplan leitet sich aus dem Teilanfrageplan ab, den alle gruppenzugehörigen Anfragen gemeinsam

haben. Durch die Ausdruckssignatur ist es möglich, dass die Anfragen einer Gruppe verschiedene Konstanten beinhalten. Da das Ergebnis der gemeinsamen Teilanfrage aber alle möglichen Werte der Anfragekonstanten enthält, muss dieses Ergebnis anhand dieser Werte gefiltert und mit Hilfe von Operatoren den entsprechenden Anfragen zugewiesen werden. Diese Operatoren entsprechen einem speziellen Splittoperator kombiniert mit einem Joinoperator. Der Joinoperator arbeitet dabei auf den konstanten Werten der Gruppentabelle. Jedes Ergebnis des Joinoperators wird dann durch den Splittoperator an die Anfragen verteilt, in dem die Zielanfrage basierend auf dem Zielpuffernamen des Ergebnistupels ausgewertet werden kann. Diese Situation wird in Abbildung 14 veranschaulicht.

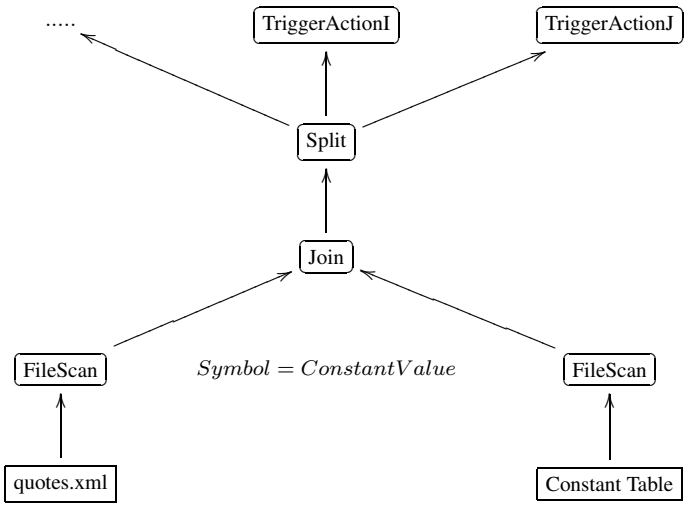

Abbildung 14: Gruppenplan der Anfragen aus Abbildung 11

Da die Anzahl der aktiven Gruppen im System sehr hoch ist (so etwa 1000-10000), werden die Gruppenpläne in einer speicherresidenten Hash-Tabelle, der sogenannten Gruppentabelle, gespeichert. Den Schlüssel dieser Tabelle bildet die Gruppensignatur.

Inkrementeller Gruppierungsalgorithmus
In diesem Teilabschnitt soll beschrieben werden, wie der Ablauf einer Anfrage im System genau vor sich geht. Sobald eine neue Anfrage im System ankommt, durchläuft der Gruppenoptimierer den Anfrageplan von den Blattknoten hinauf zur Wurzel. Mit der aufgebauten Ausdruckssignatur versucht der Gruppenoptimierer nun, die Anfrage einer existierenden Gruppe zuzuweisen. Wurde eine Gruppe gefunden, so bricht der Optimierer den Anfrageplan in zwei Teile. Der untere Teil wird gelöscht und der obere Teil wird dem Gruppenplan hinzugefügt. Wird allerdings keine Gruppe gefunden, so wird eine neue Gruppe zu dieser Signatur erzeugt und in die Gruppentabelle mit aufgenommen. Im Allgemeinen hat eine Anfrage verschiedene Signaturen, so dass sie in mehrere Gruppen einsortiert werden kann. Aus diesem Grund wird der Signaturvergleichsprozess solange über den Rest des Anfrageplans ausgeführt, bis wieder der Anfang des Plans erreicht wird. Zusammenfassend lässt sich sagen, dass dieser Algorithmus effizient ist, da er nur einen Durchlauf über den Anfrageplan erfordert.

Zeitbasierte Anfragen
Das bisher vorgestellte Verfahren ist hauptsächlich geeignet, änderungsbasierte Anfragen zu verar-

beiten. Da das System aber auch zeitbasierte Anfragen unterstützt und diese etwas anders behandelt werden müssen, wird diese Vorgehensweise im Folgenden beschrieben. Prinzipiell können zeitbasierte Anfragen genauso wie änderungsbasierte Anfragen gruppiert werden. Es müssen lediglich die Zeitangaben mit einbezogen werden. Hierbei sind folgende zwei Herausforderungen zu bewältigen: Zum einen müssen die Zeitaktivitäten erfasst werden und zum anderen ist eine gemeinsame Ausführung und Nutzung in verschiedenen Zeitintervallen nicht ohne weiteres korrekt. Bei diesem Punkt besteht die Möglichkeit, nur solche Ergebnisse zu teilen, die bereits vor dem angegebenen Zeitpunkt produziert wurden.
Innerhalb dieser zeitbasierten Anfragen kann man zwei Kategorien von Ereignissen festlegen, die dann mit Hilfe von Ereignisermittlung bearbeitet werden. Einerseits Datenquellenänderungsereignisse und zum anderen Zeitereignisse. Datenquellenänderungsereignisse können weiter unterteilt werden in *push-based-* und in *pull-based*-Ereignisse. Push-based-Ereignisse informieren das System, sobald interessante Daten geändert wurden und pull-based-Ereignisse kontrollieren periodisch, ob sich Daten ändern. Zeitereignisse hingegen werden nur zu spezifizierten Zeiten ausgeführt und werden in chronologischer Reihenfolge in Ereignislisten gespeichert. Somit kann bei der Anfrageausführung überprüft werden, ob davon abhängige Anfragen ausgeführt werden müssen.
Alternativ zur Ereignisermittlung gibt es die inkrementelle Auswertung. Diese Auswertung erlaubt es, Anfragen nur in geänderten Daten auszuführen, welches die Anzahl der Ausführungen entscheidend reduziert. Hierzu werden sogenannte Deltadateien gespeichert, in denen die Änderungen erfasst werden. Die Anfragen können direkt über diesen Deltadateien ausgeführt werden, anstatt über den Original-Dateien. Allerdings ist diese Vorgehensweise nicht bei allen Operatoren möglich, wie beispielsweise beim Joinoperator. Hier berechnet *Niagara* die Änderungen in einer XML-Datei und mischt diese dann in die Deltadatei.

Memory caching
Da allerdings nicht alle Informationen der kontinuierlichen Anfragen gespeichert werden können, werden Caching-Verfahren mit begrenztem Speicherplatzbedarf angewendet. Diese Verfahren werden hauptsächlich für Anfragepläne, Systemdatenstrukturen und für Datendateien verwendet. Dabei wird folgendermaßen verfahren:
Nicht-gruppierte, änderungbasierte Anfragen werden zwischengespeichert und mittels der *Least Recently Used*-Strategie periodisch ausgeführt. Zeitbasierte Anfragen mit kleinen Intervallen haben vor denen mit großen Intervallen Vorrang. NiagaraCQ hält jüngst referenzierte Dateien im Cache. Kleine Deltadateien, die durch den Splittoperator erzeugt werden, werden bearbeitet und dann verworfen. Durch das Zwischenspeichern kleinerer Dateien werden viele I/O-Zugriffe verhindert. Des Weiteren kann die Ereignisliste der zeitbasierten Anfragen sehr groß sein, so dass diese lediglich in einem Zeitfenster dieser Liste gespeichert wird.

3 Auswertung der Optimierungsvarianten

In diesem Kapitel werden die vorgestellten Optimierungsstrategien in einen Vergleich zu bisherigen Strategien gestellt. Hierdurch sollen die Vorteile anhand konkreter Zahlen oder Beurteilungen gezeigt werden.

3.1 Vergleich Volcano und Greedy

In diesem Abschnitt wird der Algorithmus *Greedy* im Vergleich zum ursprünglichen *Volcano-Algorithmus* dargestellt, um hinsichtlich ihrer Multi-Query-Optimierung den Nutzen darzustellen.
Allgemein konnte festgestellt werden, dass die Kosten zur Planberechnung sinken, wenn der verfügbare Speicherplatz ansteigt. Da die Kosten mit höheren Datenmengen auch ansteigen, wurden die Testverfahren mit Hilfe von TPCD-Anfragen vom Typ BQ5 auf einer TPCD-Datenbank durchgeführt. Dabei konnte festgestellt werden, dass der Greedy-Ansatz bei einer Benchmarkausführung etwa 33000 Sekunden schneller als der Volcano-Algorithmus einen Ausführungsplan entwickelt [RSSB00].

3.2 Vergleich Gruppenoptimierung (Niagara) mit einer Nichtgruppenoptimierungsstrategie

Bei diesem Vergleich eines gruppierenden mit einem nicht-gruppierenden Optimierungsansatzes soll der Nutzen des gemeinsamen Berechnens und die Verhinderung unnötiger Anfrageausführungen dargestellt werden. Hierzu wurden einige grundlegende Festlegungen getroffen, wie beispielsweise die Systemumgebung. Weitere Angaben hierzu sind im Aufsatz [CDTW00] im Abschnitt 5 enthalten. Unter der Annahme, dass die benutzerdefinierten Anfragen sehr ähnlich sein werden, war es notwendig, einige Anfragetypen zu klassifizieren. Dabei wurden folgende Typen erzeugt:

1. Notify me when Intel stocks change.
2. Notify me of all stocks whose prices rise more than 5 percent.
3. Notify me when Intel stocks trades below 100 dollars.
4. Notify me all of changes to stocks in the computer service industry and related company information.

Des Weiteren wurden folgende Parameter zur Experimentauswertung festgelegt:

- N repräsentiert die Anzahl der installierten Anfragen, welche in Bezug auf Systemskalierung eine Aussage treffen kann.
- F stellt die Anzahl ausgeführter Anfragen innerhalb einer Gruppe dar.
- C steht für die Anzahl der geänderten Tupel.
- T ist die Ausführungszeit. Im gruppierendem Ansatz besteht eine benutzerdefinierte Anfrage aus einem Gruppenteil und aus einem nicht zur Gruppe gehörenden Teil. Hieraus ergibt sich, dass die Ausführungszeit sich aus den entsprechenden Ausführungszeiten T_g und T_{ng} für jede in F ausgeführte Anfrage zusammensetzt und somit folgt

$$T = T_g + \sum_{F} T_{ng}$$

.

Im ersten Versuch wurden nur Anfragen des Typs 1 betrachtet. Außerdem wird angenommen, dass $C = 1000$ und $F = N$ gilt, so dass in beiden Ansätzen alle Anfragen ausgeführt werden. Bei der nicht-gruppierenden Optimierungsstrategie steigt die Ausführungszeit T dramatisch an, wenn N ansteigt. Durch dieses Ergebnis zeigt sich, dass dieser Ansatz nicht für Hochlast-Systeme geeignet ist.

Dem gegenüber benötigt der gruppenorientierte Ansatz bedeutend weniger Ausführungszeit. Bei einer weiteren Anwendung wurden $F = N = 2000\ Anfragen$ erzeugt und auch weiterhin wurden nur Anfragen vom Typ 1 betrachtet. Hier soll festgestellt werden, inwiefern sich C auf T auswirkt. Beim nicht-gruppierenden Ansatz kann in diesem Fall festgestellt werden, dass T ansteigt, sobald auch C ansteigt. Beim gruppierenden Ansatz verhält sich T proportional zu C, da für jede Anfrage in N der Auswahloperator ausgeführt werden muss.
Beim nächsten Experiment wurden die Anfragentypen 2 und 3 betrachtet und Werte wie folgend belegt: $C = 1000\ Tupel$ und $F = N$. Auch in diesem Fall arbeitet der gruppierende Ansatz besser als der nicht-gruppierende. Dabei ist zudem noch auffällig, dass bei den Anfragen vom Typ 4 die gruppierende Strategie noch sehr viel besser als die nicht-gruppierende ist, da der Joinoperator teurer als der Auswahloperator ist.
In den ersten Experimenten wurden die Anfragetypen jeweils getrennt betrachtet, um die Effektivität der verschiedenen Ausdruckssignaturen zu verdeutlichen. Es können aber auch Optimierungsergebnisse erreicht werden, wenn Anfragetypen kombiniert werden. Bei diesem Ansatz ist es allerdings notwendig, dass die Anfragetypen gleiche Ausdruckssignaturen aufweisen [CDTW00].

4 Globale QoS-Optimierung

Das Thema des Quality-of-Service (QoS) befasst sich mit der Dienstgüte in Kommunikationsnetzen. Je nach Standard werden unterschiedliche Parameter zum Festlegen und Messen von QoS verwendet [wik05].
Bei der Bearbeitung von Anfragen auf Datenströmen sind allerdings neue Techniken notwendig, die beispielsweise nicht-blockierende Ausführungen oder auch Beschränkungen beim Speicherplatz erlauben [Tum05].
Um in einem System QoS einsetzen zu können, bedarf es folgender Eigenschaften: Man benötigt eine QoS-Spezifikation, welche die QoS-Anforderungen an eine Anwendung festlegt. Die QoS-Spezifikation setzt Richtlinien in Bezug auf Performanz, Synchronisation und Kosten. Außerdem wird eine Abbildung zwischen den QoS-Anforderungen und den Systemressourcen verwendet. Diese Abbildung beruht auf Parametern, welche in der Spezifikation festgelegt wurden. Des Weiteren sollten QoS-Mechanismen zur Verfügung stehen, die das entsprechende QoS-Verhalten realisieren [qos05].
Bei der globalen Quality-of-Service-Optimierung besteht die Grundidee darin, die Kosten der Anfragebearbeitung und die Antwortzeit zu minimieren, wobei die Ergebnisqualität erhalten bleiben soll. Hierbei wird in traditionellen DBMS nach dem Best-Effort-Prinzip verfahren.

5 Zusammenfassung und Ausblick

Zusammenfassend lässt sich zum Thema *Multi-Query-Optimierung* sagen, dass schon einige sehr gute Ansätze hierzu realisiert wurden. Einige grundlegende Verfahren, beispielsweise der *Volcano-Algorithmus* mit zwei seiner Optimierungsmöglichkeiten, der *Greedy-Ansatz* und das *Niagara-Projekt* mit seinem Subsysten *NiagaraCQ*, wurden hier vorgestellt.
Der *Volcano-Algorithmus* behandelt das Problem, wie man bei mehreren Anfragen gemeinsame Teilausdrücke erkennen und somit weniger Ausführungskosten erhalten kann. Dies wurde aufbauend auf der Datenstruktur des gerichteten azyklischen Graphen realisiert. Diese Realisierungen der Grundidee bilden eine Basis für zukünftige Herausforderungen, wie beispielsweise die Anordnung der Anfra-

gen, so dass temporär gespeichterte Daten während der Ausführung wiederverwendet werden können [RSSB00]. Eine Erweiterung wäre die Einbeziehung der Bearbeitung sehr großer Anfragemengen (große Workloads), die anhand abstrakter Anfragen reduziert werden könnten.
Das Projekt *Niagara* ist ein skalierbares Anfragesystem für kontinuierliche Anfragen im Internet. Aufbauend auf der Idee, das viele gestellte Anfragen sich sehr ähnlich sind, wurde der Ansatz der inkrementellen Gruppenoptimierung gewählt. Dies hat sich bewährt, was im Vergleich zu nicht-gruppierten Anfragen leicht erkennbar ist. Bei dem System *NiagaraCQ* wird darauf hingewiesen, dass zukünftige Arbeit die Realisierung der dynamischen Umgruppierung der einzelnen Gruppen beinhalten sollte [CDTW00].

Allgemein kann man zu diesem Thema beispielsweise folgende Metafragen stellen: Besteht im Datenbankbereich die Notwendigkeit fundamentale und allgemeingültige Modelle, Algorithmen und Systeme für Datenströme zu entwerfen? Vielleicht reicht es aus, diese Gebiete mittels Ad-hoc-Lösungen zu bewältigen. Sind Datenstromsysteme effektiver als konventionelle Datenbanktechnologien, die Stromoperatoren wie beispielsweise Trigger unterstützten? Oder besteht wirklich die Notwendigkeit einer eigenen *Datenstrom-Algebra* [BBD+02]?
Ob diese Fragen eine Antwort erhalten, wird sich erst mit der Zeit und der weiteren Entwicklung auf diesem Gebiet ergeben.

Literatur

[BBD+02] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. *Models and Issues in Data Stream Systems*, 2002. PODS 2002: 1-16.

[CDTW00] Jianjun Chen, David J. DeWitt, Feng Tian, and Yuan Wing. *NiagaraCQ: A Scalable Continuous Query System for Internet Databases*, 2000. SIGMOD Conference 2000: 379-390.

[qos05] *Quality of Service*, 2005. http://www.objs.com/survey/QoS.htm.

[RSSB00] Prasan Roy, S. Seshadri, S. Sudarshan, and Siddhesh Bhobe. *Efficient and Extensible Algorithms for Multi Query Optimization*, 2000. SIGMOD Conference 2000: 249-260.

[Sud00] S. Sudarshan. *Multi-Query-Optimization*, 2000. http://www.cse.iitb.ac.in/dbms/Data/Courses/CS632/1999/mqo-talk/mqo-talk.html.

[Tum05] *Datenbanksysteme*, 2005. http://www-db.in.tum.de/research/overview.shtml.

[wik05] *Quality of Service*, 2005. http://de.wikipedia.org/wiki/Qos.