Integriertes Seminar

Datenbanken und Informationssysteme SS 05

Thema: Data Streams

Sebastian Bächle

s_baechl@informatik.uni-kl.de

Betreuer: Prof. Dr.-Ing. Dr. h. c. Theo Härder

Plangenerierung, Optimierung und Anfrageverarbeitung in datenstromorientierten Systemen

Technische Universität Kaiserslautern

Lehrgebiet Datenverwaltungssysteme

AG Datenbanken und Informationssysteme

Prof. Dr.-Ing. Dr. h. c. Theo Härder

AG Heterogene Informationssysteme

Prof. Dr.-Ing. Stefan Deßloch

Inhaltsverzeichnis

1 Einleitung **1**

2 Anfrageverarbeitung in datenstromorientierten Systemen **1**
2.1 Phasen der Anfrageverarbeitung . . . 1
2.2 Besonderheiten datenstromorientierter Systeme . . . 1
2.3 Optimierungsziele . . . 2

3 Architekturen von DSMS **3**
3.1 Operatoren . . . 3
3.2 Behandlung der Stromdaten . . . 4

4 Übersetzung von Anfragen **4**

5 Generierung von Anfrageplänen **5**
5.1 Algebraische Optimierung . . . 5
5.2 Anfrageplanbewertung . . . 6
5.3 Kostenmodelle für Join-Algorithmen . . . 9

6 Join-Algorithmen für DSMS **13**
6.1 Der Temporal Progressive Merge Join . . . 13
6.2 Der XJoin . . . 16
6.3 Optimierung mit Metadaten . . . 18

7 Ausführung von Anfragen **20**

8 Fazit **21**

Literatur **22**

1 Einleitung

Bisher wurde die anspruchsvolle Aufgabe des Verwaltens großer Informationsmengen von Datenbankmanagement-Systemen (DBMS) übernommen. Die weltweit zu verarbeitende Datenmenge steigt aber stetig an und gleichzeitig entstehen neue Anforderungen, denen ein Datenmanagement-System gerecht werden muss. In vielen neuen Anwendungsgebieten etwa müssen die Daten nicht mehr vorrangig gespeichert werden. Vielmehr ist es für Applikationen notwendig, Informationen aus ständig neu eintreffenden Daten schnell und flexibel zu extrahieren. Solche Systeme zur Überwachung von Finanzwerten, Netzwerkverkehr etc. sollen möglichst zeitnah und effektiv ein aktives Verhalten des Systems ermöglichen. Klassische DBMS sind für ein solches aktives Verhalten aber nicht konzipiert. Deshalb werden spezielle Datenstrommanagement-Systeme (DSMS) benötigt, die Anfragen auf Datenströmen vielseitig und effizient unterstützen.

In dieser Arbeit sollen die generelle Vorgehensweise der Anfrageverarbeitung und spezielle Algorithmen für DSMS vorgestellt werden. Ein besonderes Augenmerk wird auf die effiziente Realisierung von Verbundanfragen gelegt, da diese eine häufig benötigte und aufwändig durchzuführende Funktion in Anfragen verkörpern. Um die Grundkonzepte der Anfrageverarbeitung in DSMS vorzustellen, werden nur einzelne Anfragen ohne spezielle Beachtung der Dynamik von Datenströmen betrachtet.

2 Anfrageverarbeitung in datenstromorientierten Systemen

2.1 Phasen der Anfrageverarbeitung

Anfragen an das System werden in klassischen DBMS in mehreren Phasen verarbeitet. In der ersten Phase, der so genannten *Übersetzungsphase*, wird die in einer Anfragesprache (z. B. SQL) formulierte Anfrage für die weitere Verarbeitung vorbereitet. Aus dem daraus entstehenden *Anfragegraph* wird in der *Optimierungsphase* ein geeigneter, optimierter Anfrageplan generiert. Dieser wird dann in der dritten Phase, der *Ausführungsphase*, vom System eingesetzt, um die Daten abzuleiten. Danach werden in der vierten und letzten Phase die Ergebnisse der Anfrage dem Anwendungsprogramm zur Verfügung gestellt (*Bereitstellungsphase*).

Anfragen an ein DSMS durchlaufen ebenfalls diese Phasen, wobei es jedoch zum Teil deutliche Unterschiede in der Vorgehensweise sowie in der Dauer bzw. zeitlichen Abgrenzung dieser Phasen gibt. Da in einem datenstrombasierten System die Ankunft neuer Daten zeitlich unbestimmt ist, können und sollten bereits erstellte Ergebnisse ausgeliefert werden, obwohl eine Anfrage noch nicht beendet ist. Es kommt also zu einer Überschneidung der Ausführungsphase und der Bereitstellungsphase. Ebenso können in der Optimierungsphase keine Annahmen über die Laufzeit einer Anfrage gemacht werden, da die Datenströme als potentiell unendlich anzusehen sind.

2.2 Besonderheiten datenstromorientierter Systeme

Aufgrund der neuen Herausforderungen, die aus der Verarbeitung von Datenströmen resultieren, können die in klassischen DBMS eingesetzten Methoden zur Anfrageverarbeitung nicht direkt auf ein DSMS angewendet werden. Auch die Anfragen selbst, die an ein DSMS gestellt werden, sind zwar ähnlich denen an ein DBMS, dennoch gibt es hier verschiedene Typen, zwischen denen unterschieden werden muss.

Zum einen wird in DSMS zwischen einmaligen und kontinuierlichen Anfragen unterschieden [BBD+02]. Einmalige Anfragen haben einen fest definierten Start- und Endzeitpunkt, innerhalb derer sie durchgeführt werden. Kontinuierliche Anfragen können für das stetige Monitoring von Datenströmen eingesetzt werden. Darüber hinaus kann noch zwischen vordefinierten und zur Laufzeit definierten Anfragen unterschieden werden. Ein DSMS sollte also in der Lage sein, jeden dieser möglichen Anfragetypen möglichst effizient umsetzen zu können.

Eine der wichtigsten Eigenschaften von Datenströmen ist ihre potentielle Unendlichkeit. Bei der Verarbeitung der Daten können keine Annahmen über die insgesamt zu verarbeitende Datenmenge gemacht werden. DSMS benötigen also spezielle Mechanismen, um mit begrenzten Ressourcen die unbegrenzten Ströme bearbeiten zu können. In der Regel werden hier Fenstersemantiken auf die Ströme angewandt, um Joins und ähnliche blockierende Operationen anwenden zu können.

Eine weitere Besonderheit, mit der DSMS umgehen können müssen, ist die Ungewissheit, wann Daten zur Verarbeitung verfügbar werden. In DBMS ist die Anfrageverarbeitung *pull-basiert*, das heißt, das System kann gezielt dann auf die Daten zugreifen, wenn sie gerade für die Bearbeitung einer Anfrage benötigt werden. DSMS sind im Gegensatz dazu *push-basiert*. Das System hat keinen Einfluss darauf, wann es welche und wie viele Daten zu verarbeiten hat. Es müssen also Strategien und Algorithmen eingesetzt werden, die auch auf schwankende Datenraten anwendbar sind. Außerdem muss ein DSMS damit umgehen können, nicht immer diejenigen Daten als nächstes zu bekommen, die für eine Aktion benötigt werden.

Anwender verlangen von einem DSMS möglichst schnell Antworten, wenn eine Anfrage an das System gestellt wird. Neben universeller Einsatzbarkeit gehört deshalb die Geschwindigkeit, mit der ein System Ergebnisse liefern kann, zu den wichtigsten Kriterien eines DSMS. Oft werden nicht einmal vollständige Ergebnisse benötigt und der Benutzer gibt sich mit einer kleineren, aber schnell verfügbaren Ergebnismenge zufrieden. Diese und viele weitere Herausforderungen wie Skalierbarkeit, Handhabung von Bursts, dynamische Anpassung an veränderte Stromeigenschaften etc. müssen also beim Entwurf eines DSMS berücksichtigt werden.

2.3 Optimierungsziele

Die hohen Lasten, mit denen ein DSMS zurechtkommen muss, erfordern eine (kontinuierliche) Optimierung der Anfrageverarbeitung. Wie in vielen Bereichen, in denen optimiert werden soll, entstehen auch hier Zielkonflikte. Eine möglichst hohe Durchsatzrate der Operatoren führt in der Regel zu einem größeren Hauptspeicherbedarf. Da dieser ebenfalls begrenzt ist, muss bei zu hohem Datenaufkommen ein Teil der Daten auf den Sekundärspeicher ausgelagert werden, was wiederum zu Lasten der Performance geht. In DSMS müssen bei zu schnellen Strömen mitunter Daten vorzeitig verworfen werden, um die Datenflut beherrschbar zu machen. Damit einher geht aber ein Verlust an Genauigkeit der Ergebnisse.

Aus diesem Grund ist es sinnvoll, ein DSMS so zu entwerfen, dass der Anwender für seine Anfrage spezifische QoS-Kriterien bestimmen kann. Wenn eine Anfrage lediglich einen groben, aber schnellen Überblick über die aktuellen Stromdaten liefern soll, müssen für diese Anfrage nicht zwingend alle Ergebnisse erzeugt werden. Auf diese Weise kann das System Ressourcen sparen, um sie anderen Anfragen zur Verfügung zu stellen. Eine andere Anfrage wiederum kann alle möglichen Ergebnisse benötigen, muss aber nicht zwingend innerhalb einer gewissen Zeitspanne diese Ergebnisse liefern können. Das Ziel muss es also sein, eine flexible Anfrageverarbeitung zu schaffen, die es ermöglicht, sowohl individuell für einzelne Anfragen als auch global eine hohe Leistungsfähigkeit zu erreichen. Dazu werden ein effizientes Bewertungsmodell

sowie eine Auswahl aus verschiedenen Implementierungen logischer Operatoren benötigt, um eine situationsgerechte Anfragedurchführung zu ermöglichen.

3 Architekturen von DSMS

Im Gegensatz zu DBMS sollen in der Regel die Stromdaten von DSMS nicht nachhaltig gespeichert werden. Deshalb benötigen DSMS keine ausgeklügelten Zugriffs- und Speicherungsstrukturen, um die Daten zu verwalten und anzufragen. Vielmehr wird eine Architektur benötigt, die effizient und performant mit den Daten beim Eintreffen umgeht, um daraus abgeleitete Ergebnisse auszugeben. Ein DSMS selektiert, aggregiert und verdichtet also Informationen aus den eingehenden Datenströmen, um selbst wieder einen Strom oder ein einzelnes Ergebnis auszuliefern.

Datenstromorientierte Systeme benötigen spezielle Puffer zum (Zwischen-) Speichern der eintreffenden Daten und Operatoren, die diese gepufferten Daten verarbeiten können. Eine Anfrage in einem DSMS ist deshalb eine Sammlung aus Puffern in Form von Warteschlangen (Queues), Planoperatoren und speziellen Zwischenspeichern (Synopsen), durch die die Stromdaten geleitet werden. Eine Warteschlange verbindet dabei immer die Ausgabe eines Planoperators mit der Eingabe eines anderen. Eine Synopse gehört jeweils zu einem Planoperator. Sie speichert temporäre Informationen für einen statusbehafteten Operator wie z. B. den Join. Im Falle eines Hash-Joins enthält eine Synopse also die Hashtabellen, die der Join führt [ABB+].

3.1 Operatoren

Die Daten aus den Strömen treffen als relationale Tupel ein, weshalb die Typen logischer Operatoren, die ein DSMS unterstützen sollte, die gleichen sind wie in konventionellen relationalen Systemen. Blockierende Operationen wie Joins oder Aggregatfunktionen benötigen zusätzliche Einschränkungen bezüglich der Tupelmengen, auf welchen die Funktion durchgeführt werden soll. Andernfalls würde für ihre Ausführung unendlich viel Speicher benötigt bzw. sie wären nicht in der Lage, in endlicher Zeit ein Ergebnis zu liefern. Diese Einschränkungen sind entweder zeit- oder mengenbasiert und werden als so genannte *Fenster* (*Windows*) auf den Strömen bezeichnet. Nur die Tupel, die innerhalb des definierten Fensters der Anfrage liegen, werden in die Auswertung einbezogen. Die Fenster für kontinuierliche Anfragen können als so genannte *gleitende Fenster* (*Sliding Windows*) stetig über dem Strom verschoben werden oder unterteilen den Strom in jeweils disjunkte Teilabschnitte.

Die physischen Operatoren (*Planoperatoren*), mit denen ein DSMS diese Funktionalität realisiert, müssen daher auf diese Anforderungen hin implementiert werden. Die Selektion und die Projektion sind als einfache Filteroperationen mit verhältnismäßig geringem Aufwand zu realisieren. Die blockierenden Operationen jedoch müssen effektive Maßnahmen bieten, um in DSMS nicht-blockierend und skalierend ausgeführt werden zu können. Die zwei klassischen Verfahren für einen Join – der Nested Loops Join und der Hash Join – sind für die Verwendung in DSMS entsprechend modifizierbar. Da die konkrete Implementierung aber entscheidenden Einfluss auf die Ausführungsgeschwindigkeit der Anfrage hat, werden spezielle Join-Algorithmen in Abschnitt 6 genauer betrachtet. Eine große Auswahl an Planoperatoren mit unterschiedlichen Vor- und Nachteilen ermöglicht es dem Optimierer, die Anfrageausführung möglichst gut an die momentanen Gegebenheiten im System anzupassen.

3.2 Behandlung der Stromdaten

Für die Leistungsfähigkeit eines DSMS sind neben der Implementierung der Operatoren vor allem die Speicherstrukturen, in denen das System die zu bearbeitenden Daten puffern kann, von großer Bedeutung. Das Aurora-System [ACC+03] beispielsweise organisiert die Warteschlangen als Ringpuffer aus Blöcken von 128 KB. Bei Bedarf kann die Größe der Ringpuffer stets durch Hinzunahme bzw. Wegnahme weiterer Blöcke angepasst werden. Sollte der Hauptspeicher knapp werden, muss das System Teile eines Ringpuffers auf den Sekundärspeicher auslagern. Dazu wird versucht, die Blöcke eines Puffers auszuwählen, die zur Zeit nicht gebraucht werden. Für diese Entscheidung kann das System z. B. die Fenster, die von den Anfragen auf die Puffer angewendet werden, zu Hilfe nehmen. Ein solches Fenster benötigt lediglich einen *Start*- und einen *End*-Zeiger, um die Grenzen eines Fenster zu markieren. Auf diese Weise brauchen die Daten für verschiedene Anfragen auch nicht dupliziert zu werden. Es reicht aus, für jeden Operator die Zeiger auf einer Warteschlange anzuwenden. Aus diesen Informationen kann das System bestimmen, welche Daten nicht in den Fenstern von Anfragen mit hoher Prioriät liegen und deshalb ausgelagert werden können.

4 Übersetzung von Anfragen

Die Übersetzungsphase hat das Ziel, die Anfrage in eine geeignete Repräsentation zu transformieren, mit Hilfe derer die Anfrage restrukturiert, optimiert und mit Planoperatoren besetzt werden kann. Diese Phase ist identisch mit der Übersetzungsphase in DBMS. Die meist textuell formulierte Anfrage wird dazu in einen Anfragegraphen übersetzt. Als erstes wird die Anfrage auf ihre syntaktische und lexikalische Korrektheit hin überprüft. Danach folgt die semantische Analyse. Dort wird überprüft, ob die angefragten Ströme mit den entsprechenden Attributen im System registriert sind, die (Vergleichs-) Operatoren typverträglich sind etc. Als dritter Schritt der Übersetzungsphase wird der bisher aufgebaute Anfragegraph normalisiert, um ihn besser bearbeiten zu können. Die mit `UND` und `ODER` verknüpften Prädikate der Anfrage werden dabei in eine konjunktive bzw. disjunktive Normalform umgeformt. Die KNF tendiert dazu, Vereinigungsoperationen zuerst durchzuführen, während die DNF Verbundoperationen vorzieht. Anschließend wird der Anfragegraph noch vereinfacht. Es werden Redundanzen entfernt und Hüllen über Qualifikationsprädikate gebildet (z. B. zur Konstantenpropagierung), um zusätzliche Auswertungsmöglichkeiten zu bieten. Mit Hilfe der semantischen Integritätsbedingungen, die für die Ströme definiert sind, können weitere Vereinfachungen erreicht werden.

Q_1 zeigt eine mögliche Anfrage an ein DSMS, in dem mehrere miteinander kommunizierende Geräte überwacht werden. Q_1 erfragt den Fall, in dem ein Gerät des Typs DEV-4711, das über den Strom *a* angebunden ist, bei einer Anfrage an das Gerät mit Strom *b* warten muss, da dieses bereits eine Anfrage aus *c* bearbeitet. Sie soll im Folgenden als Beispiel für den Ablauf der Anfrageverarbeitung dienen.

Anfrage Q_1:

```
SELECT Sa.ID, Sc.ID, Sb.timestamp
FROM a Sa[500], b Sb[500], c Sc[100]
WHERE Sa.ID = Sb.targetID
AND Sb.requestID = Sc.requestID
AND Sa.type = 'DEV-4711'
AND Sb.state = 'BUSY'
```

Da die Übersetzungsphase analog zu der in konventionellen DBMS abläuft und die Anfrage keine komplexen Teile enthält, wird hier auf die einzelnen Transformationsschritte nicht weiter eingegangen. Die graphische Repräsentation der Anfrage entspricht dem Anfragegraphen AG_1 in Abbildung 1.

5 Generierung von Anfrageplänen

Ein *Anfrageplan* legt für eine Anfrage konkret fest, in welcher Reihenfolge die (Zwischen-) Ergebnisse erzeugt werden sollen und welche Planoperatoren hierfür verwendet werden. Für eine Anfrage werden mehrere solcher Anfragepläne generiert und dann wird mit Hilfe eines Bewertungsmodells entschieden, welcher am schnellsten bzw. effizientesten ist. Beim Erstellen der Anfragepläne wird auf die Optimierung der Ausführung besonders Wert gelegt. Die Reihenfolge, in der die einzelnen Operationen durchgeführt werden, bietet beispielsweise viele Möglichkeiten, die Ausführungsgeschwindigkeit zu verbessern. Dazu kann der Anfragegraph wie in DBMS restrukturiert werden (*algebraische Optimierung*). Die *nicht-algebraische Optimierung* in einem DBMS befasst sich mit der Auswahl der günstigsten Planoperatoren und Zugriffspfade auf die benötigten Relationen. In DSMS fällt Letzteres weg. Die Auswahl der Planoperatoren wird direkt bei der Bewertung der einzelnen Anfragepläne durchgeführt (siehe Abschnitt 5.2).

5.1 Algebraische Optimierung

Die algebraische Optimierung beruht auf Transformationsregeln, die festlegen, welche Operatoren man vertauschen kann, ohne die Semantik der Anfrage zu ändern. Zwei Anfragegraphen auf Datenströmen sind hierbei äquivalent, wenn bei der Anwendung des *Snapshot-Operators* auf die Ströme zu jedem Zeitpunkt t die gleichen Multimengen resultieren [CHKS04]. Die von DBMS bekannten Regeln der algebraischen Optimierung sind in DSMS häufig, aber nicht immer analog anwendbar. Aufgrund der Einfachheit und der Bekanntheit der Regeln von DBMS werden an dieser Stelle nur einige wenige exemplarisch vorgestellt.

Um die Größe der einzelnen Tupel schnell zu reduzieren und den Speicherverbrauch in den Warteschlangen der nachfolgenden Operatoren so gering wie möglich zu halten, sollten Projektionen (ohne Reduzierung von Duplikaten) so früh wie möglich vorgenommen werden. Betrachtet man einen Projektionsoperator als eine Art Zugriffsfilter kann er direkt zwischen der Ausgangswarteschlange des Vorgängers und dem nachfolgenden Operator installiert werden, wodurch eine zusätzliche Warteschlange im Operatorgraphen und somit Speicherplatz im System gespart wird.

Eine weitere Regel zur algebraischen Optimierung ist die frühe Selektion von Tupeln, die für spätere, zum Teil aufwändige Operationen wie z. B. Joins überhaupt erst in Frage kommen. Damit kann die Belastung des Systems effektiv gesenkt werden. Mehrere Selektionen und Projektionen kann man – falls es im Zuge einer *Multi-Query-Optimierung* nicht zu einem Nachteil werden sollte – auch zu einer einzigen zusammenfassen. Sollte von mehreren, aufeinanderfolgenden unären Operatoren, welche nicht zusammengefasst werden sollen, mindestens einer zeitaufwändig zu berechnen sein, spielt allerdings die Reihenfolge der Operatoren eine entscheidende Rolle (Bsp. in Abschnitt 5.2).

Die dritte Regel, die an dieser Stelle betrachtet werden soll, ist die Umordnung der Verbundoperationen, um die Zwischenergebnisse so gering wie möglich zu halten. Zur Schätzung der Größe der Zwischenergebnisse werden in DBMS die Kardinalitäten der beteiligten Relationen

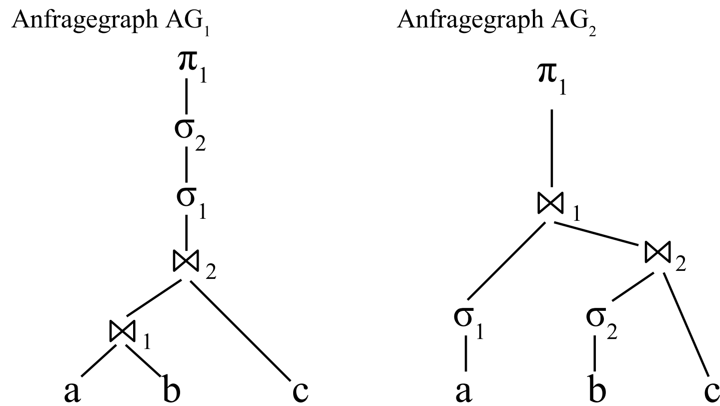

Abbildung 1: Alternative Anfragegraphen AG_1 und AG_2 für die Anfrage Q_1

sowie der *Join-Selektivitätsfaktor* benötigt. Da in den potentiell unendlichen Datenströmen aber keine Kardinalitäten angegeben werden können, sind Regeln, die sich auf die Größe von Relationen beziehen, nicht ohne weiteres auf DSMS übertragbar. Weitere Regeln zu algebraischen Restrukturierung finden sich in [Mit95, S.53].

Abbildung 1 zeigt zwei mögliche Anfragegraphen für die Anfrage Q_1. AG_1 ist hierbei die direkte Transformation der textuellen Anfrage, während bei AG_2 die Selektionen auf den einzelnen Strömen möglichst früh vorgenommen werden, um die Datenrate möglichst früh zu senken.

5.2 Anfrageplanbewertung

Bewertungsmodelle für Anfragepläne in klassischen DBMS sind kostenbasiert und benötigen Metadaten der Relationen wie vorhandene Indizes, Anzahl der Tupel pro Speicherseite, Kardinalitäten der Relationen etc. Solche Informationen existieren in einem stromorientierten System allerdings nicht bzw. sind nicht definiert (z. B. die Kardinalität eines unendlichen Stromes). Deshalb haben Viglas und Naughton die durchsatzbasierte Optimierung (*Rate-Based Optimization*) für DSMS vorgeschlagen [VN02]. Als einführendes Beispiel zur durchsatzbasierten Bewertung von Anfrageplänen betrachten wir die zwei Selektionen σ_1 und σ_2 im Anfragegraph AG_1. Sowohl σ_1 als auch σ_2 habe den Selektivitätsfaktor $f_i = 0,1$ und die Ausgaberate des Joins sei $r_{out} = 500/sek$. Die Selektion σ_1 kann 50 Tupel pro Sekunde verarbeiten und σ_2 sei so einfach, dass sie die Daten so schnell verarbeiten kann, wie sie eintreffen. Für ein einfaches Kostenmodell bei der Abarbeitung der Ablaufkombinationen $\sigma_1 \rightarrow \sigma_2$ und $\sigma_2 \rightarrow \sigma_1$ kann der Optimierer nicht entscheiden, welcher Plan günstiger ist, da als Kardinalität eines Stromes ∞ angenommen werden muss und somit auch die Kosten beider Pläne unendlich sind.

Das durchsatzbasierte Modell betrachtet hier die Ausgaberate, die beide Kombinationen erreichen können:

$$\sigma_1 \rightarrow \sigma_2 : r = (f_1 \cdot max(50/sek, 500/sek)) \cdot f_2 = 0,5/sek$$

$$\sigma_2 \rightarrow \sigma_1 : r = max((f_2 \cdot 500/sek), 50/sek) \cdot f_1 = 5/sek$$

Die zweite Anordnung produziert schneller Ergebnisse und ermöglicht es dem System, unter Umständen schon früher ausgewertete Tupel aus der Eingangswarteschlangen zu entfernen. Diese einfache Überlegung zeigt, dass in strombasierten Systemen die Ankunfts- und Verarbeitungsraten von Operatoren entscheidende Kriterien sind, um Anfragepläne zu bewerten.

Für die Selektion und Projektion als einfache unäre Operationen ist es von Bedeutung, ob ihre Verarbeitungskosten c_{op} pro Tupel kleiner oder größer als die durchschnittliche Ankunftsrate ihres Quellstromes sind. Im ersten Fall entspricht die Ausgaberate r_{op} des Operators *op* der Eingaberate r_{Quelle}, da die Tupel schneller verarbeitet werden können, als sie ankommen. Im zweiten Fall stellt die Verarbeitungszeit den begrenzenden Faktor dar und es gilt $r_{op} = 1/c_{op}$. In der Regel kann man jedoch davon ausgehen, dass die Auswertung eines Selektionsprädikates sehr schnell durchführbar ist und die Datenraten nicht beeinflusst werden.

Komplizierter ist es, die Ausgaberate für einen Join zu ermitteln. Die Ausgaberate selbst ist definiert als das Verhältnis aus der Anzahl an generierten Ergebnissen und der dafür benötigten Verarbeitungszeit. Viglas und Naughton beginnen hier mit der Ermittlung der Anzahl der Join-Ergebnisse zu einem Zeitpunkt t. Seien nun r_a und r_b die Ankunftsraten der beiden Ströme a und b und f_1 die Join-Selektivität des Joins $\bowtie_1$ der Beispielanfrage. Nach einer Zeiteinheit können $f_1 r_a r_b$ Tupel als Ergebnis des Joins generiert werden. Nach $2t$ werden weitere $2 f_1 2 r_a r_b - f_1 r_a r_b$ (jeweils $f_1 2 r_a r_b$ pro Eingabestrom minus der bereits im ersten Intervall erzeugten Ergebnistupel) generiert. Zum Zeitpunkt $t2$ sind es also $f_1 r_a r_b + 3 \cdot f_1 r_a r_b$ Ausgabetupel. Verfährt man auf diese Weise weiter, erhält man induktiv die Formel für kontinuierliche Zeit:

$$r = f_1 r_a r_b \cdot \int (2t - 1)\, dt$$

Löst man dieses Integral, so erhält man die Anzahl n der Join-Ergebnisse, die zu einem beliebigen Zeitpunkt t erzeugt werden:

$$n = f_1 r_a r_b t \cdot (t - 1)$$

Zur Berechnung der Ausgaberate r_{out} eines Joins benötigt man noch die Zeitkosten der Join-Operation. Die Kosten eines Joins hängen sehr stark von der Implementierung der Speicherverwaltung und dem verwendeten Algorithmus ab. Aus diesem Grund werden hier nur die zugrunde liegenden Überlegungen des durchsatzbasierten Bewertungsmodells und keine konkreten Werte für einen Join aufgeführt. Die Kostenmodelle aus Abschnitt 5.3 können leicht in dem hier vorgestellten Bewertungsmodell verwendet werden.

Die Kosten eines Joins lassen sich differenziert betrachten als die Summe der Teilkosten c_i, die durch das Eintreffen eines neuen Tupels im linken bzw. rechten Strom eines Joins entstehen. Im Beispiel bezeichne c_a die Kosten für das Auftreten eines neuen Tupels im Strom a. Die Kosten für den Strom a betragen somit $C_a = r_a \cdot c_a$. Analog seien auch c_b und C_b definiert.

Im Intervall t erreichen $t \cdot r_a$ Tupel über den Strom a und $t \cdot r_b$ Tupel über den Strom b den Join $\bowtie_1$. Als anfallende Verarbeitungskosten für die in diesem Zeitfenster angekommenen Daten erhält man somit:

$$\begin{aligned} C_{\Delta t} &= t \cdot r_a \cdot c_a + t \cdot r_b \cdot c_b \\ &= t \cdot C_a + t \cdot C_b \\ &= t \cdot (C_a + C_b) \end{aligned}$$

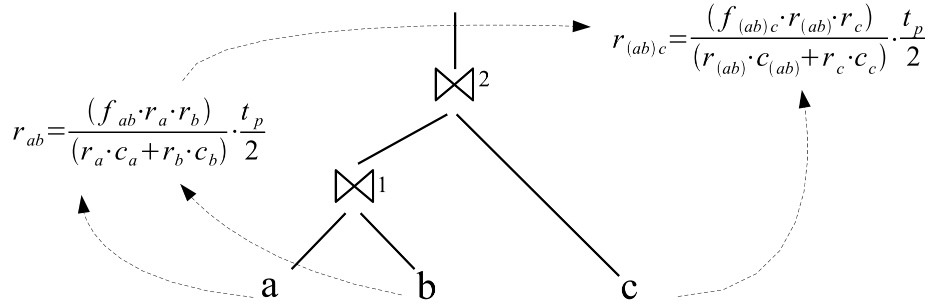

Abbildung 2: Local Rate Maximization für das Intervall t_p

Jetzt kann die Ausgaberate r_{out} einer Join-Operation einfach über ihre Definition berechnet werden:

$$\begin{aligned} r_{out} &= \frac{f_1 r_a r_b t \cdot (t-1)}{t \cdot (C_a + C_b)} \\ &= \frac{f_1 r_a r_b \cdot (t-1)}{(C_a + C_b)} \\ &\approx \frac{f_1 r_a r_b t}{(C_a + C_b)} \end{aligned}$$

Möchte man nun für einen Anfrageplan feststellen, wie viele Ergebnisse er zu einem beliebigen Zeitpunkt t_p produziert, reicht es, die Funktion der Ausgaberate des Planes über die Zeit t zu integrieren:

$$\#Outputs = \int_0^{t_p} f(t)\, dt$$

Mit Hilfe der Herleitung der Ausgaberate als eine Funktion der Zeit lassen sich Anfragepläne anhand zweier Optimalitätskriterien auswählen: Entweder wählt man den Plan P_i, der bis zu einem Zeitpunkt t_0 die meisten Ergebnisse produzieren wird oder man wählt den Plan P_j, der am schnellsten eine vorgegebene Anzahl an Ergebnissen n liefern wird. Die letztgenannte Variante ist beispielsweise von Vorteil, wenn man eine Mindestzahl an Ergebnissen benötigt, um eine bestimmte Genauigkeit der Anfrage garantieren zu können.

Das Problem dieser Methode ist jedoch, dass für jeden zu untersuchenden Anfrageplan das Integral gelöst werden muss, was für die praktische Anwendung bei einer großen Anzahl an Plänen zu aufwändig ist. Deshalb schlagen die Autoren den Einsatz von Heuristiken vor, um in konkreten Systemen eine schnelle Berechenbarkeit zu ermöglichen. Ihre beiden Vorschläge der *Local Rate Maximization* (LRM) und der *Local Time Minimization* (LTM) beruhen auf der Annahme, dass eine lokale Optimierung der Ausführungsgeschwindigkeit zu einem globalen Optimum führt.

Die LRM versucht die Anfragepläne mit der besten Ausgaberate über einem Intervall t_p zu finden. Dazu schätzt der Optimierer die Ausgaberate eines Operators ab und propagiert dieses Ergebnis gemäß Abbildung 2 nach oben. Der Wert $t_p/2$ ist ein Schätzwert, um die mittlere Gesamtrate der Operation zu ermitteln. Wie Abbildung 3 zeigt, beruht dieser Wert darauf, dass man mit $t_p/2$ für jeden beliebigen Zeitpunkt x die Ausgaberate um denselben Wert überschätzt, wie man ihn für den Zeitpunkt $t_p - x$ unterschätzt.

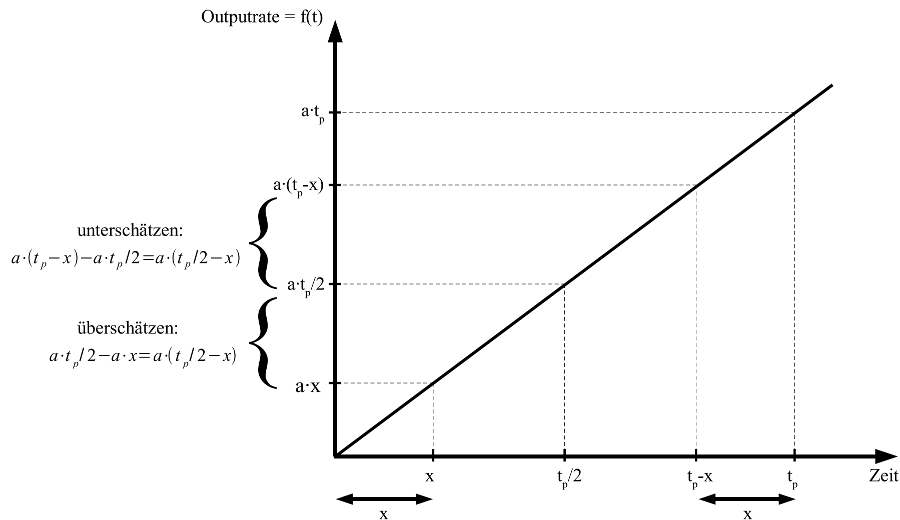

Abbildung 3: Abschätzung eines mittleren Ausgaberate im Intervall t_p

Wie bereits erwähnt, kann es auch von Vorteil sein abzuschätzen, welcher Anfrageplan am schnellsten eine benötigte Anzahl von Ergebnissen liefern wird. Die LTM baut dazu auf der Heuristik der LRM auf. Die Anzahl n der von einem Join-Operator erzeugten Ergebnistupel hängt mit $n = r \cdot t$ funktional von der dafür bereitgestellten Zeit ab. Folglich reicht es, für alle Anfragepläne P_i die benötigte Zeit $t_i = n/r$ zu berechnen, um den günstigsten Plan auszuwählen. Die Ausgaberaten r_{op} der Operatoren eines Anfrageplanes werden analog zur LRM rekursiv berechnet.

In einem Beispiel sollen die Anfragepläne P_1 und P_2, deren Operatorfolgen denen in den Anfragegraphen AG_1 und AG_2 entsprechen, bewertet werden. Aus Einfachheitsgründen wird angenommen, dass für beide die gleichen Planoperatoren eingesetzt werden und die Kosten für jedes neue Tupel, das auf einen Verbund geprüft werden soll, 2 ms betragen. Die Eingaberate des Stromes a sei $100/sek$, die des Stromes b sei $600/sek$ und die von c sei $300/sek$. Die Selektivitäten der Joins seien $f_1 = 0,01$ und $f_2 = 0,002$ und die der Selektionen $\sigma_1 = 0,5$ und $\sigma_2 = 0,3$.

Im ersten Plan erreicht der Join über die drei Ströme eine Ausgaberate von ca. $62/sek$. Die Join-Anordnung in Anfrageplan P_2 erreicht für den Join mit etwa $125/sek$ fast den doppelten Wert. Betrachtet man beide Pläne insgesamt, so erreicht Plan P_2 mit einer Gesamtausgaberate von $66/sek$ im Gegensatz zu $56/sek$ bei P_1 auch die bessere Gesamt-Performance.

5.3 Kostenmodelle für Join-Algorithmen

Das betrachtete durchsatzbasierte Kostenmodell wird entscheidend von der Wahl der Operatorkosten C_{op} beeinflusst. Diese Operatorkosten selbst werden wiederum von dem Algorithmus zur Realisierung des Operators, dem Speichermanagement und den Eingangsraten der Operatoren beeinflusst. Auch maschinenabhängige Parameter wie die Bandbreite des E/A-Subsystems, die Hauptspeichergröße und die Leistungsfähigkeit der CPU können mit Hilfe der Operatorkosten im durchsatzbasierten Modell berücksichtigt werden. Da der dynamische Aspekt in DSMS die herausragende Rolle spielt, können vom Optimierer darüber hinaus auch Laufzeitinformationen wie die Auslagerungsquote der Warteschlangen, die Freispeichersituation etc. in die Berechnungen mit einbezogen werden. In diesem Abschnitt soll das Kostenmodell für Varianten des Nested-

Loops- sowie des Hash-Join-Algorithmus von Kang und Viglas vorgestellt werden [KNV03].

Trifft ein neues Tupel in einem der Eingangsströme eines Joins ein, so wird (abstrahiert von Maßnahmen wie vorherige Pufferung etc.) jeweils ein Durchlauf der drei Stufen *Prüfen*, *Einfügen* und *Invalidieren* angestoßen. Beim *Prüfen* wird getestet, ob das Tupel mit den bereits eingetroffenen Tupeln des Join-Partners einen Verbund eingeht. Beim *Einfügen* wird das neue Tupel in das Join-Fenster seines Eingangsstromes eingefügt und beim *Invalidieren* werden alle ungültig gewordenen Tupel aus dem Fenster des Eingangsstromes gelöscht. Zum Einfügen ist es in der Regel nur nötig, den *Start*-Zeiger des aktuellen Fensters auf das neue Tupel zu verschieben. Beim Invalidieren können mehrere Tupel gleichzeitig aus dem Fenster gelöscht werden. Der Aufwand hierzu hängt von der Fenstersemantik und dem Join-Algorithmus ab, der unter Umständen schon entscheiden kann, ob manche Tupel noch einen Verbund eingehen können oder nicht. Auch hier reicht es in der Regel aus, alte Tupel durch ein Verschieben des *End*-Zeigers aus dem Fenster zu löschen.

Die Kosten $C_{\bowtie_1}$ für den Join $\bowtie_1$ betragen also:

$$\begin{aligned} C_{\bowtie_1} &= r_a \cdot (Prüfen(b) + Einfügen(a) + Invalidieren(a)) \\ &\quad + r_b \cdot (Prüfen(a) + Einfügen(b) + Invalidieren(b)) \end{aligned}$$

Um unterschiedlich schnelle Eingangsströme besser verarbeiten zu können, werden die Kosten des Joins auf die einzelnen Eingangsströme aufgeteilt. Für das Beispiel bedeutet das, dass der Join in die zwei Joins $a \rtimes b$ und $a \ltimes b$ zerlegt wird. Die Kosten des gesamten Joins ergeben sich durch einfache Addition der beiden Kosten:

$$C_{\bowtie_1} = C_{a \rtimes b} + C_{a \ltimes b}$$

wobei gilt:

$$C_{a \rtimes b} = r_a \cdot (Prüfen(b)) + r_b \cdot (Einfügen(b) + Invalidieren(b))$$

und

$$C_{a \ltimes b} = r_b \cdot (Prüfen(a)) + r_a \cdot (Einfügen(a) + Invalidieren(a))$$

Der *Nested Loops Join* (NJ) läuft in einer äußeren Schleife über die Tupel eines Eingangsstromes und sucht für diese in einer inneren Schleife Verbundpartner in den Tupeln des zweiten Eingangsstromes. Während das Eintreffen neuer Tupeln im äußeren Eingangsstrom einfach durch eine erneute Iteration aufgefangen werden kann, ist das Eintreffen neuer Tupel im inneren Strom problematisch. Um den Join nicht-blockierend durchführen zu können, könnte man alle während eines Durchlaufes der inneren Schleife eingetroffenen Tupel des inneren Stromes auffangen und in einer zweiten inneren Schleife die Verbundprüfung durchführen.

Die Kosten für den Verbund von Tupeln aus a mit den Tupeln aus b mit Hilfe des einfachen NJ-Algorithmus errechnet sich wie folgt: Es werden $r_a \cdot B$ Vergleiche in der inneren Schleife durchgeführt. B bezeichnet die Fenstergröße von b, also die Anzahl der Tupel aus b, die momentan als mögliche Verbundpartner betrachtet werden. Außerdem werden noch r_b Einfügungen und ebenso viele Invalidierungen im Fenster B ausgeführt. Jede dieser Aktionen führt zu einem Zugriff auf ein bestimmtes Tupel, weshalb für jede Aktion zugriffsspezifische Kosten anfallen. Der Wert $c_{p,n}$ steht hier für die Zugriffs- und Vergleichskosten eines Tupels und $c_{i,n}$ für Einfügekosten in der konkreten NJ-Implementierung. Der Einwege-NJ $a \rtimes b$ hat somit die Kosten:

$$C_{a \rtimes b}(NJ) = r_a \cdot B \cdot c_{p,n} + 2 \cdot r_b \cdot c_{i,n}$$

Der einfache *Hash Join* (HJ) vermeidet es in vielen Iterationen, die Tupel des jeweils anderen Stromes nach Verbundpartnern abzusuchen, sondern versucht die Suche über eine Hashfunktion

möglichst schnell auf wahrscheinliche Verbundpartner einzuschränken. Dazu führt er für die Tupelfenster eigene Hashtabellen, auf die alle Tupel eines Eingabestromes abgebildet werden. Trifft nun aus Strom a ein neues Tupel ein, wird dafür ein Hashwert berechnet und in der entsprechenden Hashklasse des Stromes b nach möglichen Verbundpartnern gesucht. Die Anzahl der verschiedenen Hashpartitionen in einer Hashtabelle beträgt $|B|$. Es müssen somit $r_a \cdot \frac{B}{|B|}$ Vergleiche für den Einwege-Join vorgenommen werden. Analog zum Nested Loops Join fallen für Einfügungen und Invalidierungen $2 \cdot r_b$ mal Kosten an:

$$C_{a \rtimes b}(HJ) = r_a \cdot \frac{B}{|B|} \cdot c_{p,h} + 2 \cdot r_b \cdot c_{i,h}$$

Joins mit Hashfunktionen eignen sich nur für einen Gleichheitsverbund, während der NJ sehr aufwändig bei der Suche vorgeht. Dieser Performance-Nachteil des NJ kommt besonders zum Tragen, wenn bei einem Einwege-Join von a zu b der Strom a sehr viel schneller ist als Strom b. Abhilfe schafft hier ein Index über das Fenster des Stromes b, der es erlaubt, schneller einen möglichen Verbundpartner zu finden. Wählt man für den Index einen B^+-Baum, so verbessert sich die Kostenfunktion zu:

$$\begin{aligned} C_{a \rtimes b}(BJ) &= r_a \cdot (\left\lceil log_{N+1} \left\lceil \frac{B}{N} \right\rceil \right\rceil + 1) \cdot \lceil log_2 N \rceil \cdot c_{p,b} \\ &\quad + r_b \cdot 2 \cdot (\left\lceil log_{N+1} \left\lceil \frac{B}{N} \right\rceil \right\rceil + 1) \cdot \lceil log_2 N \rceil \cdot c_{i,b} \end{aligned}$$

N steht hier für die Anzahl der Tupel in einem Baumknoten. Eine Alternative zur B^+-Baumstruktur ist der T-Baum-Index [LC86]:

$$\begin{aligned} C_{a \rtimes b}(TJ) &= r_a \cdot (1,5 \cdot \left\lceil log_2 \left\lceil \frac{B}{N} \right\rceil \right\rceil - 1) + \lceil log_2 N \rceil \cdot c_{p,t} \\ &\quad + r_b \cdot (2 \cdot 1,5 \cdot \left\lceil log_2 \left\lceil \frac{B}{N} \right\rceil \right\rceil - 1) + \lceil log_2 N \rceil \cdot c_{i,t} \end{aligned}$$

Die Kosten eines vollständigen Joins erhält man durch einfache Addition. Eine symmetrische Anwendung des Hash Joins (HHJ) für die Operation $a \bowtie b$ führt folglich zu den Kosten $C_{a \rtimes b} + C_{a \ltimes b}$. Der Ansatz der getrennten Betrachtung der Verbundkosten macht es möglich, asymmetrische Varianten wie NT als einen Join aus NJ und TJ zu untersuchen. Wie später noch gezeigt wird, sind asymmetrische Varianten effektiv für Joins bei unterschiedlich schnellen Strömen einsetzbar.

Immer noch unbekannt sind die jeweiligen Kostengewichte c_p und c_i der Kostenformeln. Sie werden für die konkrete Implementierung durch Messungen ermittelt. Man misst die benötigte CPU-Zeit für die Batch-Bearbeitung einer Tupelmenge, welche der Anzahl an Tupeln entspricht, die in einer Minute bei einer bestimmten Eingangsrate zu bearbeiten sind. Für eine Eingangsrate von 100 Tupel/sek besteht die Testmenge also aus 6000 Tupeln. Das hat den Vorteil, dass man damit auch die Rate bestimmen kann, bei der ein bestimmter Algorithmus das System über seine Grenzen belastet. In diesem Falle ist die Bearbeitungszeit nämlich größer als 60 Sekunden. Um die beiden Kostenarten ermitteln zu können, wird für jeden Algorithmus ein Durchgang vorgesehen, in dem entweder nur Verbundpartner gesucht oder nur Einfügungen und Invalidierungen vorgenommen werden.

In Tabelle 1 sind die Gewichte, die von Kang et al. für ihre Implementierungen ermittelt wurden, aufgeführt. Diese Werte können auf jedem System unterschiedlich sein.

Tabelle 2 enthält die damit errechneten Kosten der verschiedenen Join-Varianten für die Ströme a und b aus den vorangegangenen Betrachtungen. Für die Berechnungen wurde eine Hashfunktion mit 250 Hashklassen zugrunde gelegt. Die Anzahl N der Tupel pro Baumknoten wurde auf

	c_p	c_i
NJ	$3 \cdot 10^{-4}$	10^{-4}
BJ	$5,5 \cdot 10^{-4}$	$7,8 \cdot 10^{-4}$
BJ	$2,6 \cdot 10^{-4}$	$2,6 \cdot 10^{-4}$
TJ	$2,6 \cdot 10^{-4}$	$2,7 \cdot 10^{-4}$

Tabelle 1: Ermittelte Kostengewichte der verschiedenen Join-Algorithmen

	NJ	HJ	BJ	TJ
$a \rtimes b$	15,12	1,05	4,73	3,37
$a \ltimes b$	90,02	0,82	2,91	1,91

Tabelle 2: Join-Kosten der verschiedenen Algorithmenvarianten

100 festgelegt. Der NJ ist aufgrund des großen Fensters deutlich langsamer als die drei anderen vorgestellten Varianten. In diesem Beispiel wäre ein HHJ mit Modellkosten von $1,05+0,82 = 1,87$ am günstigsten.

Für die Auswahl eines geeigneten Algorithmus zur Durchführung der Anfrage sind die Punkte interessant, an denen sich die Kostengeraden schneiden. An diesen Schnittpunkten ändert sich die Vorteilhaftigkeit der Algorithmenkombinationen. Die Schnittpunkte lassen sich durch das Gleichsetzen zweier benachbarter Algorithmenkombinationen berechnen. Man erhält dann ein Eingangsratenverhältnis, bei dem beide Algorithmen gleich teuer sind. Der Schnittpunkt von TNJ und THJ errechnet sich beispielsweise wie folgt:

$$\begin{array}{rrcl}
& C_{a \bowtie b}(TNJ) & = & C_{a \bowtie b}(THJ) \\
\Leftrightarrow & C_{a \ltimes b}(TJ) + C_{a \rtimes b}(NJ) & = & C_{a \ltimes b}(TJ) + C_{a \rtimes b}(HJ) \\
\Leftrightarrow & C_{a \ltimes b}(NJ) & = & C_{a \ltimes b}(HJ) \\
\Leftrightarrow & r_a \cdot B \cdot 3 \cdot 10^{-4} + 2 \cdot r_b \cdot 10^{-4} & = & r_a \cdot \frac{B}{|B|} \cdot 5,5 \cdot 10^{-4} + 2 \cdot r_b \cdot 7,8 \cdot 10^{-4} \\
\Leftrightarrow & r_a \cdot (B \cdot 3 \cdot 10^{-4} - \frac{B}{|B|} \cdot 5,5 \cdot 10^{-4}) & = & r_b \cdot (2 \cdot 7,8 \cdot 10^{-4} - 2 \cdot 10^{-4}) \\
\Leftrightarrow & \frac{r_a}{r_b} & = & \frac{(2 \cdot 7,8 \cdot 10^{-4} - 2 \cdot 10^{-4})}{(B \cdot 3 \cdot 10^{-4} - \frac{B}{|B|} \cdot 5,5 \cdot 10^{-4})} \\
\Leftrightarrow & \frac{r_a}{r_b} & = & \frac{13,6}{3 \cdot B - 55}
\end{array}$$

Der Wert $B/|B|$ gibt lediglich die Größe der Hashpartitionen der Implementierung wieder und wurde in der Umformung aus Einfachheitsgründen auf 10 gesetzt. Bemerkenswert an diesem Term ist die Tatsache, dass nur die Fenstergröße entscheidend ist. Je größer man des Fenster wählt, desto mehr verschiebt sich der Schnittpunkt nach links. Wenn also der Eingangsstrom b bei einer Fenstergröße von 500 Tupeln mehr als 106 mal schneller ist als a, ist der TNJ-Algorithmus besser als THJ.

Die Güte dieser Vergleiche zeigt sich deutlich in Abbildung 4. Die Schnittpunkte der errechneten Modellkosten beschreiben recht genau die Situation in einem realen System. Wie in der Grafik ersichtlich, ermöglicht dieses Verfahren jedoch nicht nur die Bestimmung desjenigen Algorithmus, der für die aktuellen Stromraten der günstigste ist, sondern erlaubt es alternativ denjenigen Algorithmus auszuwählen, der innerhalb der erwarteten Schwankungen in den Stromraten die beste Gesamt-Performance liefert.

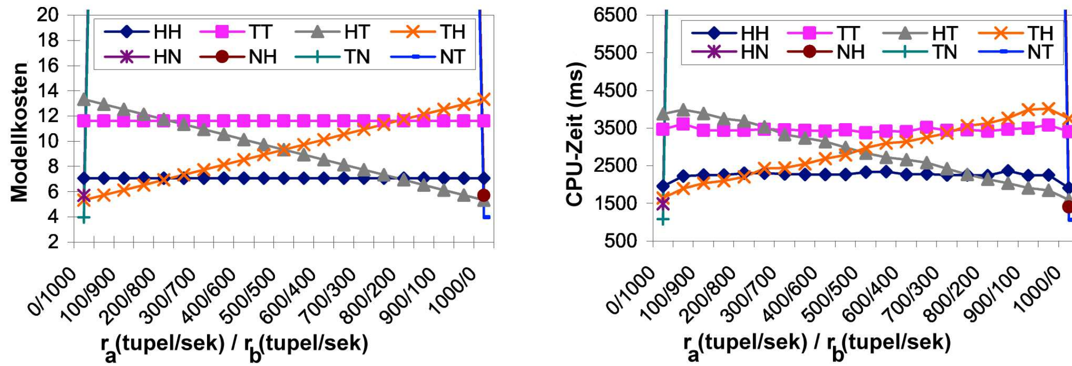

Abbildung 4: Vergleich der errechneten Modellkosten und real gemessener Ausführungszeiten verschiedener Join-Varianten

6 Join-Algorithmen für DSMS

Da der Join als kostenintensive und häufig verwendete Operation einen besonderen Einfluss auf die Verarbeitungsgeschwindigkeit von Anfragen hat, werden in den nachfolgenden Abschnitten mehrere Join-Algorithmen für DSMS eingehend besprochen. Die Abschnitte 6.1 und 6.2 stellen zwei effiziente Join-Algorithmen für große Fenster vor. Im letzten Abschnitt werden noch einige Ansätze vorgestellt, wie mit Hilfe von Meta-Informationen weitere Verbesserungen hinsichtlich der Durchsatzgeschwindigkeit und des Speicherverbrauchs erzielt werden können.

6.1 Der Temporal Progressive Merge Join

Die bisher betrachteten Join-Algorithmen und -Bewertungen gingen davon aus, dass der Join immer komplett im Hauptspeicher durchgeführt werden kann. In einem realen Umfeld mit vielen gleichzeitig durchzuführenden Anfragen und sehr hohen Lasten während so genannter Bursts ist dies jedoch nicht garantierbar. Gerade bei komplexen Anfragen mit großen Datenmengen müssen Teile der Warteschlangen auf den Sekundärspeicher ausgelagert werden. Der *Temporal Progressive Merge Join* (TPMJ) ist eine mögliche Variante, um Joins in DSMS mit der Auslagerung auf den Sekundärspeicher effizient durchführen zu können [CHKS05]. Darüber hinaus ist der TPMJ im Gegensatz zum HJ in der Lage, verschiedenste Join-Prädikate zu unterstützen.

Der TPMJ ist eine um eine Fenstersemantik erweiterte Adaption des *Progressive Merge Joins* (PMJ) [DSTW02] für strombasierte Systeme. Der PMJ basiert auf der Idee des einfachen *Sort Merge Joins* (SMJ). Zuerst sortiert der SMJ die Tupel der einzelnen Relationen anhand einer globalen Ordnung. In der nächsten Phase beginnt er anhand dieser globalen Ordnung den Verbund durchzuführen. Jede Eingabe erhält dabei eine so genannte *SweepArea*, in die das zu verarbeitende Tupel eingefügt wird. Dann werden in einer Reorgansiationsphase anhand dieses Elementes alle Tupel aus den SweepAreas der Eingaben entfernt, die aufgrund der geordneten Abarbeitung nicht mehr für einen Verbund in Frage kommen. Anschließend sucht der SMJ in den SweepAreas der anderen Eingabeströme nach Verbundpartnern.

Der SMJ ist aufgrund seiner Zweiphasigkeit eine blockierende Operation, da er zuerst alle Eingaben erhalten und sortieren muss, bevor er mit dem Verbund beginnen kann. Der PMJ ist eine nicht-blockierende Variante, die bereits beim Sortieren Ergebnisse liefert. Der PMJ arbeitet ähnlich wie das externe Sortieren, das zum Sortieren großer Datenmengen über den Sekundärspeicher

abgewickelt wird. Beim externen Merge Sort wird ein Teil der Eingabe in den Hauptspeicher geladen, dort mit konventionellen Algorithmen sortiert und als sortierte Teilfolge (*Run*) wieder ausgeschrieben. Solche Runs werden dann zusammengeführt, indem zwei Runs anhand ihrer Ordnung synchronisiert gelesen werden und sofort wieder als gemeinsamer Teil-Run ausgeschrieben werden. Der PMJ nutzt diese Methode, indem er gleichzeitig zwei Teilmengen der Eingaben zur Run-Generierung liest. Vor dem Ausschreiben führt er allerdings auf diesen beiden Runs schon die Verbundoperation durch. Dadurch liefert der PMJ schon Ergebnisse, bevor die Eingaben komplett sortiert sind. In der Merge-Phase werden wiederum Joins durchgeführt, indem wieder gleichzeitig mehrere Runs jeder Eingabe vermischt werden. Um eine Duplikaterzeugung zu vermeiden, wird geprüft, ob zwei Verbundpartner aus derselben Run-Generierungsphase stammen. Ist dies der Fall, dann wurde der Verbund bereits in der Run-Generierung durchgeführt und das Ergebnis muss nicht erneut ausgeliefert werden.

Das Verfahren kann mit kleinen Erweiterungen für die Verwendung auf Datenströmen angepasst werden. Nimmt man den vom System gestatteten maximalen Speicherverbrauch für die Eingangswarteschlange als gegeben hin, so werden die Warteschlangen erst einmal bis zu einem gewissen Belegungsgrad gefüllt. Ist dieser Belegungsgrad erreicht, wird z. B. über einen Zeiger markiert, dass die folgenden Tupel nicht mehr zu dieser Run-Generierung gehören. In einem eigenen Thread mit hoher Priorität werden die Tupel des aktuellen Runs im Speicher z. B. mit Quicksort sortiert, der Join berechnet und der Run ausgeschrieben, um im Hauptspeicher Platz freizugeben. Auf diese Weise werden für jeden Strom auch gleich viele Runs erzeugt, was vorteilhaft für den späteren Merge-Schritt ist. Dieser wird nach Möglichkeit über einen Thread mit geringerer Priorität realisiert, der dann in Phasen geringerer Systemlast aktiv wird.

Das Problem des unbegrenzten Speicherverbrauchs bleibt mit diesem Verfahren aber immer noch bestehen. Dem PMJ fehlt ein Entscheidungskriterium, ab wann er welche Tupel verwerfen kann, um Speicherplatz freizugeben. Aus diesem Grunde wurde der PMJ von Cammert et al. um eine temporale Fenstersemantik zum TPMJ ergänzt. Dazu muss garantiert werden, dass jedes Eingabetupel explizit oder implizit mit einem Startzeitstempel t_s versehen wird und, im Falle von impliziten Zeitstempeln, die Tupel auch in aufsteigender Reihenfolge ihrer Zeitstempel ankommen. Jedes Tupel erhält zusätzlich einen Endzeitstempel t_e, der bei gleitenden Zeitfenstern der Breite r gleich $t_s + r$ bzw. bei festen Fenstern der Breite m gleich $m \cdot n$ mit $n = min\{x \mid m \cdot x > t_s\}$ ist. Auf diese Weise wird für jedes Tupel ein *Gültigkeitsintervall* $[t_s, t_e)$ definiert. Ein Verbund zwischen zwei Tupeln findet bei einer Fenstersemantik damit nur dann statt, wenn sich ihre Gültigkeitsintervalle überlappen. Das Ergebnistupel erhält als Gültigkeitsintervall den Schnitt der Ursprungstupel. Bei einem binären Join reicht es somit aus, ein Tupel mit dem Intervall $[t_s, t_e)$ nur so lange zu behalten, bis im anderen Strom ein Tupel $[t'_s, t'_e)$ mit $t'_s \geq t_e$ eintrifft.

Aufgrund des geordneten Eintreffens der Tupel treffen nach einem Tupel x im Strom a nur noch Tupel x' mit $x' \geq x$ in a ein. Die Relation $\geq$ bezieht sich dabei auf die Gültigkeitsintervalle. Damit in der Reorgansiationsphase das Tupel x nicht entfernt wird, obwohl in b noch potentielle Join-Partner für x eintreffen können, dürfen die Löschungen nur noch in den SweepAreas der anderen Ströme durchgeführt werden. Es können allerdings Ergebnisse entstehen, die in der globalen Ordnung der Ergebnisse vor bereits produzierten Ergebnissen stehen. Aus diesem Grund muss ein Ergebnis für eine ordnungserhaltende Ausgabe zuerst so lange gepuffert werden, bis in allen Eingaben ein Element mit einem zum Ausgabetupel schnittfreien Gültigkeitsintervall eingetroffen ist.

Beim TPMJ ist die Strategie, mit der Duplikate verhindert werden, von besonderer Bedeutung. Beim normalen PMJ gibt man die Ergebnisse so früh wie möglich aus. Doch nun müssen die Ergebnisse aufgrund der Ordnung der Ergebnisintervalle sortiert ausgegeben werden. Der

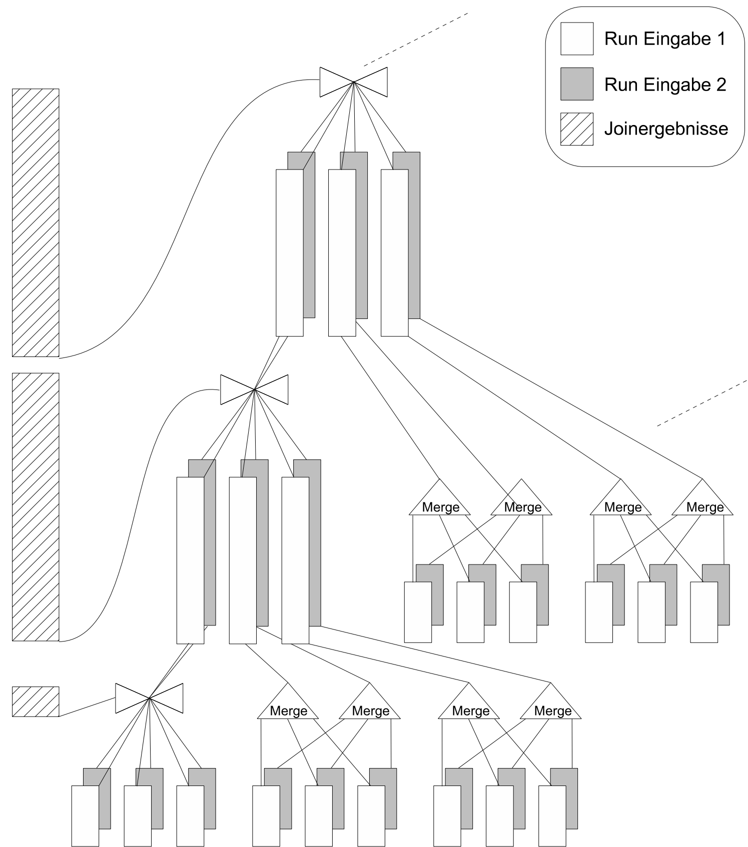

Abbildung 5: Der beim TPMJ entstehende schiefe Merge-Baum

TPMJ unterteilt dazu die Zeitachse in disjunkte Intervalle $[a_i, e_i)$ und ordnet diese den Merge-Schritten zu. In jedem Merge-Schritt entstehen somit nur Ergebnisse, deren Gültigkeitsintervalle im Intervall $[a_i, e_i)$ liegen. Durch diese disjunkte Unterteilung ist sichergestellt, dass keine Duplikate entstehen. Ein Ergebnis mit dem Startzeitstempel t wird also genau dann erzeugt, wenn alle anderen Eingabetupel dieses Merge-Vorganges einen Startzeitstempel kleiner oder gleich t haben.

Anhand des linken, unteren Joins in Abbildung 5 lässt sich die Festlegung der den Merge-Schritten zugeordneten Intervalle erläutern. Das Intervall dieses kombinierten Join- und Merge-Schrittes hat als untere Grenze den Startzeitstempel des ältesten Tupels im am weitesten links stehenden Run. Das offene Intervallende wird durch den kleinsten Startwert der maximalen Gültigkeitsintervalle bestimmt. Das garantiert, dass alle in diesem Zeitintervall liegenden Verbünde erstellt werden können. Die dort entstehenden Ergebnisse müssen nur noch nach ihren Gültigkeitsintervallen sortiert und ausgegeben werden. Somit genügt es, die anderen Runs in dieser Ebene zu mischen, ohne den Verbund zu bilden. Die dort entstehenden Ergebnisse würden nämlich in der nächsten Phase erneut gebildet und eine Duplikateleminierung wäre notwendig.

Wie Abbildung 5 zeigt, werden nur an der linken Flanke des Merge-Baumes Ergebnisse erzeugt. Ein Eingabetupel mit dem Intervall $[t_s, t_e)$ muss nur bis zu dem Merge-Schritt i mit dem Ergebnisintervall $[a_i, e_i)$ gespeichert werden, bei dem gilt: $t_e \leq e_i$. Im darauf folgenden Merge-Schritt j kann ein solches Tupel nämlich keinen Verbund mehr eingehen, da in diesem Schritt nur noch Ergebnisse entstehen, deren Startwert $t'_s \geq e_i$ sind. Auf diese Weise stellt der TPMJ sicher,

dass bei unendlichen Strömen nicht auch unendlich Speicher benötigt wird.

Der hier vorgestellte Join-Algorithmus ist dann geeignet, wenn die Datenmenge für einen Join zu groß ist, um den Algorithmus im Hauptspeicher durchführen zu können. Allerdings erfordert der Algorithmus, dass zumindest die für einen Join-Merge-Schritt benötigten SweepAreas sowie die noch zu sortierenden Ergebnisse in den Hauptspeicher geladen werden können. Ist die Join-Selektivität hierfür zu groß, müssen die Ergebnisse selbst wieder mit Hilfe des externen Sortierens nach Gültigkeitsintervallen geordnet werden.

6.2 Der XJoin

Der *XJoin* [UF00] ist eine Weiterentwicklung des symmetrischen Hash Joins, der durch das Einbeziehen des Sekundärspeichers einen geringeren Hauptspeicherverbrauch ausweist. Der XJoin ist darauf ausgelegt, auch bei schwankenden Strömen kontinuierlich Ergebnisse zu erzeugen. Die Idee des Verfahrens ist es, den zugeordneten Hauptspeicher so zum Puffern von Tupeln zu nutzen, dass beim Eintreffen neuer Datentupel direkt einige Ergebnisse erzeugt werden können. In Leerlaufphasen, in denen gerade keine neuen Tupel mehr ankommen, werden diejenigen Ergebnistupel nachgeliefert, deren Verbund erst durch ein erneutes Laden vom Sekundärspeicher möglich ist.

Im Gegensatz zu dem schon vorgestellten TPMJ ist der XJoin nicht für Sliding-Window-Anfragen konzipiert worden. Der XJoin eignet sich vor allem für Einmalanfragen oder Anfragen, bei denen jeweils ein großes Fenster auf die Ströme angewendet wird. Darüber hinaus ist der XJoin eine mögliche Variante, wenn die Tupel in unregelmäßig schnellen Blöcken im System eintreffen und sich die Erzeugung der Verbundtupel nach der aktuellen Systemlast richten kann. Wie bereits erwähnt, ist es in vielen Anwendungen oft schon ausreichend, schnell Ergebnisse zu liefern, ohne dass diese komplett sein müssen. Neben seiner Eignung für solche Anwendungen ist der XJoin jedoch auch in der Lage, alle Ergebnisse zu liefern.

Der XJoin bildet den Verbund in drei Phasen. In der ersten Phase wird ein *Memory-to-Memory* Join durchgeführt, um dem Benutzer auch bei geringem Hauptspeicherverbrauch schnell Ergebnisse liefern zu können. Abwechselnd dazu wird in der zweiten Phase ein *Disc-to-Memory* Join durchgeführt, in dem weitere Verbundergebnisse mit ausgelagerten Tupeln erzeugt werden. In der dritten Phase werden schließlich alle noch fehlenden Ergebnistupel erzeugt.

Als eine Variante des HHJ benutzt der XJoin ebenfalls Hashtabellen, um Verbundpartner zu finden. Für jeden Eingabestrom s_i wird eine Hashtabelle geführt, die die Tupel in einzelne Partitionen P_{k,s_i} aufteilt. Im Gegensatz zum HHJ befindet sich aber nur ein Teil MP_{k,s_i} jeder Partition im Hauptspeicher. DP_{k,s_i} bezeichnet den Teil der entsprechenden Partitionen, der aus Platzgründen in den Sekundärspeicher ausgelagert wurde. In den Partitionsteilen im Hauptspeicher sind jeweils die aktuellsten Tupel der Eingabeströme enthalten. Diese werden wie beim gewöhnlichen HHJ beim Eintreffen mit Hilfe einer Hashfunktion in die entsprechende Partition ihres Eingabestromes eingefügt und dann mit den Tupeln der entsprechenden Hash-Partition des zweiten Stromes auf einen Verbund geprüft. Die so direkt erzeugbaren Ergebnisse können sofort ausgegeben werden. Sobald der Hauptspeicherverbrauch der Partitionen zu groß wird, wird die Partition, die den meisten Speicherplatz belegt, in den Sekundärspeicher ausgeschrieben. Die Tupel werden dabei einfach an die bereits ausgelagerten Tupel dieser Partition angehängt (siehe Abbildung 6).

Die zweite Phase des XJoin wird gestartet, wenn beide Eingabeströme blockieren. Dabei wird der ausgelagerte Teil DP_{k,s_i} einer Partition ausgewählt und mit der korrespondierenden Hauptspeicherpartition des zweiten Eingabestromes auf einen Verbund hin überprüft. Nachdem alle möglichen Ergebnisse mit diesem ausgelagerten Teil einer Partition erzeugt wurden, wird

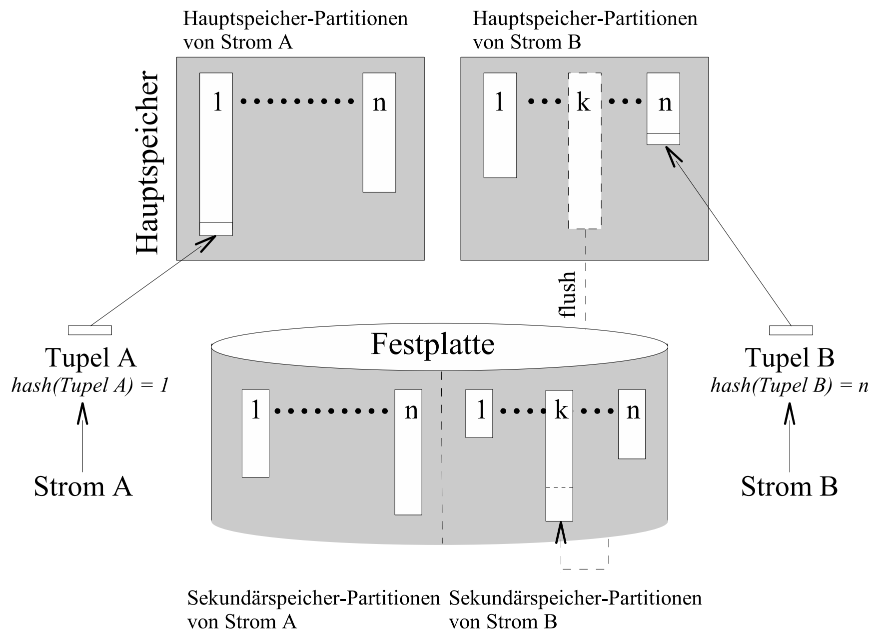

Abbildung 6: Partitionierung der Tupel beim XJoin

geprüft, ob in der Zwischenzeit neue Tupel eingetroffen sind. Sollte dies der Fall sein, wird wieder Phase 1 gestartet; falls nicht, fährt Phase 2 mit einer neuen Sekundärspeicherpartition fort. Natürlich kann im Laufe der Verarbeitung eine Partition mehrfach für Phase 2 ausgewählt werden, da sich diese ständig vergrößern kann und sich die Hauptspeichertupel des anderen Stromes ebenfalls ändern. Durch dieses Verfahren entsteht allerdings ein Overhead durch das mitunter mehrmalige Überprüfen der gleichen Tupel. Dies birgt die Gefahr, dass die zweite Phase das System für andere Anfragen zu sehr belastet oder beim Deblockieren der Eingaben die sofortige Generierung aktueller Tupel verzögert wird.

Phase 3 des XJoins ist eine reine Aufräumphase, in der alle Ergebnistupel ausgegeben werden, die in den vorherigen Phasen noch nicht erzeugt wurden. Der XJoin verlangt dazu, dass von keiner der Eingaben mehr neue Tupel hinzukommen. Es wird der Strom i bestimmt, der die wenigsten Eingabetupel geliefert hat. Für diesen wird dann die Partition DP_{1,s_i} in den Hauptspeicher geladen und bei Bedarf die anderen Partitionen auf den Sekundärspeicher ausgelagert. Dann wird die Hashtabelle dieser Partition mit den Tupeln aus MP_{k,s_j} und DP_{k,s_j} angefragt, um Ergebnisse zu erzeugen. Dieses Verfahren wird für alle Partitionen durchgeführt.

In den Phasen 2 und 3 würde der XJoin in der bisherigen Variante immer wieder bereits ausgelieferte Ergebnisse erzeugen. Deshalb wird noch ein Mechanismus benötigt, um die Duplikate zu erkennen und deren Ausgabe zu verhindern. Dazu bekommt jedes neu eingetroffene Tupel einen Ankunftszeitstempel ATS und einen Zeitstempel DTS, wenn das Tupel vom Hauptspeicher auf die Platte ausgeschrieben wird. Diese Zeitstempel beschreiben das Zeitintervall, in welchem das Tupel im Hauptspeicher vorgehalten wurde. Werden nun in Phase 2 zwei Tupel getestet, deren Hauptspeicherintervalle sich überlappen, ist der eventuelle Verbund schon in Phase 1 erstellt worden und kann übersprungen werden. Da eine Partition auf dem Sekundärspeicher aber mehrmals in der zweiten Phase ausgewählt werden darf, können immer noch Duplikate erzeugt werden. Dies wird verhindert, in dem für jede Partition Zusatzinformationen gespeichert wer-

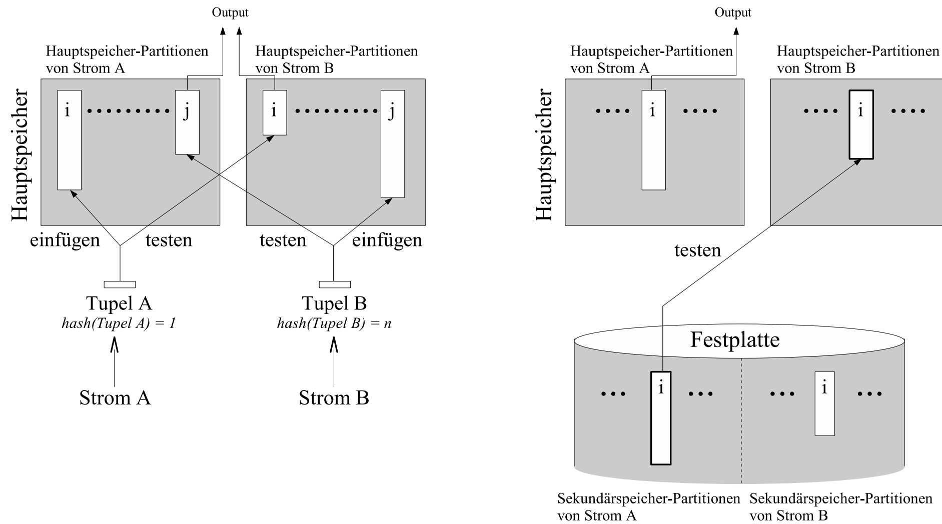

Abbildung 7: Phase 1 und 2 des XJoins

den, wann Phase 2 auf ihnen durchgeführt wurde. Für jeden Durchlauf wird ein Tupel der Form ($DTS_{last}, ProbeTS$) angelegt. DTS_{last} ist der *DTS*-Zeitstempel des letzten Tupels der Teilpartition in dieser Phase und *ProbeTS* der Zeitstempel, zu dem die Phase 2 durchgeführt wird. In einem Durchlauf über die Partition $DP_{k,a}$ kann so für zwei Tupel t_a und t_b, die das Verbundprädikat erfüllen, geprüft werden, wann das Tupel t_a bereits mit Haupspeichertupeln von b auf Verbund getestet wurde und ob sich das Tupel t_b zu einem dieser Zeitpunkte im Hauptspeicher befand. Ist beides der Fall, so kann man daraus schließen, dass der Verbund bereits erzeugt wurde und das Ergebnis überspringen. In Phase 3 kann diese Methode symmetrisch angewandt werden, um dort ebenfalls die Erzeugung von Duplikaten zu verhindern.

In [UF99] schlagen Urhan und Franklin noch die Einführung eines Caches für einen Teil der Sekundärspeichertupel in Phase 2 vor. In einem zweiten Lauf von Phase 2 wählt man dann $DP_{i,b}$, wenn im vorherigen $DP_{i,a}$ als Sekundärspeicherpartition genommen wurde. So können noch schneller Ergebnisse erzeugt werden, ohne zusätzliche E/A-Operationen durchführen zu müssen.

6.3 Optimierung mit Metadaten

Die bisher vorgestellten Verfahren sind universell auf Stromdaten anwendbar und haben wenige bis gar keine dynamischen oder inhaltlichen Informationen für eventuelle Anpassungen beachtet. In einem DSMS ist es in der Regel jedoch der Fall, dass Meta-Informationen über die einzelnen Stromquellen und deren Tupel verfügbar sind. Bevor ein Strom in einem DSMS verfügbar ist, muss der Aufbau seiner Datentupel dem System bekannt gemacht werden. Solche Informationen können vom einem System ausgenutzt werden, um z. B. bei einem 1:N-Verbund über Primär- und Fremdschlüsselbeziehungen die Ausführung zu beschleunigen.

Eine Möglichkeit, um mit Hilfe von Meta-Informationen den Speicherverbrauch und die Laufzeit zu verbessern, ist das Konzept der *Punctuations* [TMSF03]. Eine Punctuation ist ein spezielles Tupel, das zusätzlich in einen Strom eingefügt wird, um den unendlichen Strom in eine Menge von endlichen Subströmen zu unterteilen. Jede Punctuation beschreibt dabei ein bestimmtes Attributmuster. Für den jeweiligen Strom wird nach Eintreffen eines solchen Markertupels ga-

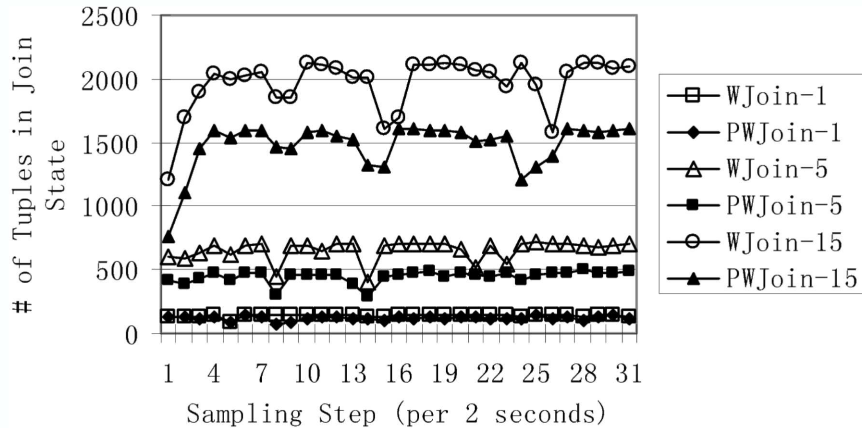

Abbildung 8: Speichervergleich zwischen PWJoin und normalem Window-Join, Länge der Subströme 100 Tupel; Join-Selektivität: 0,4

rantiert, dass keine weiteren Tupel, deren Ausprägung auf das Muster der Punctuation passen, folgen werden. Sinnvoll erzeugbar sind solche Marker z. B. bei Online-Auktionen oder Telefonaten. Hier kann nach Ende der Auktion bzw. nach Ende des Gespräches sicher davon ausgegangen werden, dass keine weiteren Daten mit derselben Auktions- oder Telefonat-ID folgen können. Für Join-Operationen bedeutet dies, dass Datentupel schon verworfen werden können, wenn in einem der beiden Ströme eine solche Punctuation auftritt und alle damit erzeugbaren Ergebnisse bereits ausgegeben wurden.

Ding und Rundensteiner entwickelten auf Basis dieser Überlegungen den *PJoin*, der als Erweiterung des XJoins in der Lage ist, die gleiche Funktionalität wie dieser mit wesentlich geringerem Speicherverbrauch durchzuführen [DMRH04]. Diesen verbesserten sie weiter zum *PWJoin* [DR04], der darüber hinaus noch in der Lage ist, die Joins mit einer Fenstersemantik durchzuführen. Die Invalidierung der Tupel, die aus dem Zeitfenster herausfallen, verläuft analog zu dem bereits vorgestellten Vorgehen beim HHJ. Ist ein neu angekommenes Tupel eine Punctuation, durchsucht der Algorithmus das Fenster des jeweils anderen Stromes, um dort die Tupel zu löschen, die nicht mehr für einen Verbund in Frage kommen. Damit diese Aktionen performant durchführbar sind, unterhält der PWJoin eine spezielle Datenstruktur, um effizient sowohl in chronologischer als auch in wertabhängiger Reihenfolge den Eingangspuffer durchlaufen zu können.

Der *MJoin* [DRH03] ist ebenfalls eine Erweiterung des XJoins, die sich Punctuations zunutze macht. Er eignet sich besonders für 1:N-Joins, die er mit Hilfe der Schemadaten des DSMS erkennt. Ein Verbund zwischen einem eindeutigen Schlüssel *Key* eines Tupels und einem Fremdschlüssel *ForeignKey* liefert einen solchen 1:N-Join. Die Tupel des Stromes a können sofort nach dem Verbund verworfen werden, da sicher ist, dass kein weiterer Verbund möglich ist. Eine aufwändige Duplikateleminierung ist ebenfalls nicht notwendig. Abbildung 9 zeigt beispielhaft die Größenordnung der Verbesserung im Speicherplatzverbrauch, die mit dem MJoin in Versuchen erreicht wurde.

In praktischen Anwendungen treffen die Datentupel häufig in schnellen Bursts ein, in denen Tupel mit gleichen Eigenschaften als Cluster im System eintreffen. Der MJoin kann dies ausnutzen, indem er die Scheduling-Strategie zur Abarbeitung der Eingabewarteschlange anpasst. Wenn im Fenster des Stromes a ein Tupel t_a vorhanden ist, zu dem er in b ein Verbundtupel findet, lohnt es

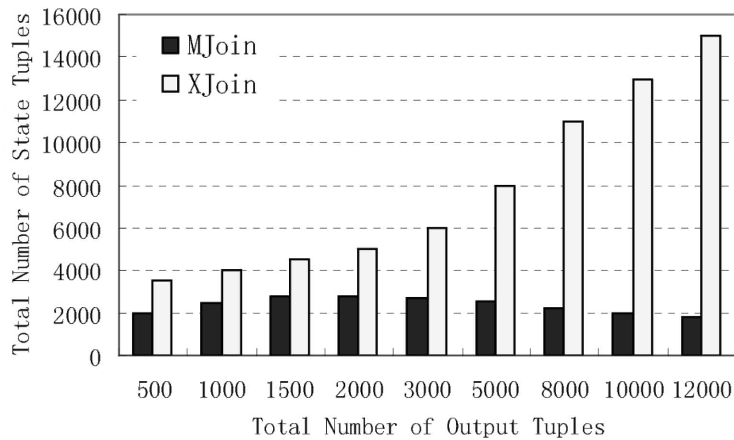

Abbildung 9: Speicherverbrauch, MJoin vs. XJoin

sich, solange die Tupel aus b abzuarbeiten, bis ein Tupel t_b kommt, das keinen Verbund mehr mit t_a eingeht. Auf diese Weise kann die Belegung der Eingabewarteschlange sehr schnell verkleinert werden. Darüber hinaus bietet der MJoin eine weitere Möglichkeit zur Optimierung bezüglich Ausgaberate oder Speicherverbrauch. Um die Durchsatzrate zu erhöhen, wird das Löschen von Tupeln nach einer Punctuation im Batch-Betrieb durchgeführt. Um Speicherplatz zu sparen, werden nicht mehr benötigte Tupel nach jeder Punctuation gelöscht.

7 Ausführung von Anfragen

Nachdem ein geeigneter Anfrageplan ausgewählt wurde, muss dieser im System installiert werden. Bei DSMS können wie in konventionellen DBMS die Anfragepläne bereits vor ihrer Nutzung im System gespeichert werden, um dann möglichst schnell ausgeführt zu werden. Sinnvoll ist die Festlegung eines Anfrageplanes im Voraus aber nur, wenn schon konkrete und einigermaßen verlässliche Annahmen über die Eingangsraten der Ströme gemacht werden können. Da in DSMS aber keine speziellen Zugriffsoperationen über Indizes und ähnliches benötigt werden, werden dabei auch keine kompilierten Anfrageprogramme gespeichert. Die Dynamik von Datenströmen würden ohnehin im laufenden Betrieb ein ständiges Neukompilieren von adaptiv angepassten Anfrageplänen erfordern. Das Aurora-System beispielsweise speichert seine Anfragepläne in einfachen XML-Dateien ab. Wird nun einer dieser erstellten Anfragepläne abgerufen, so installiert das System die darin beschriebenen Operatorkonstellationen. Installieren bedeutet in diesem Zusammenhang, dass die Planoperatoren instanziiert, entsprechend initialisiert und mit ihrer Ein- und Ausgabe an die jeweiligen Ein- und Ausgabewarteschlangen angebunden werden. Schließlich werden auch diejenigen Eingaben angeschlossen, an denen die Daten in diese neuen Teile des Systems fließen.

Dann übernimmt der Scheduler des Systems die eigentliche Ausführung. Der Scheduler hat die Aufgabe, jeweils einen der Operatoren im System auszuführen. Für die Leistungsfähigkeit eines DSMS ist das Scheduling eine wichtige Komponente. Je nach Fließgeschwindigkeit der einzelnen Ströme im System muss die verfügbare Rechenzeit auf die einzelnen Operatoren verteilt werden, um Staus und zu hohen Speicherverbauch zu vermeiden.

Da Datenströme dynamische Objekte sind und DSMS auf starke Schwankungen in den Datenraten reagieren müssen, ist besonders bei kontinuierlichen Anfragen eine stetige Optimierung zur Laufzeit nötig. Beispielsweise können neue oder entfernte Anfragen an das System die Bedingungen für eine optimale Anfrageverarbeitung stark verändern. Für diese Fälle existieren einige Ansätze wie z. B. Eddies, die sich um die dynamische Anpassung der Anfragen kümmern.

Die klassische Bereitstellungsphase der Anfrageverarbeitung verschwimmt, wie bereits erwähnt, mit der Ausführungsphase. Entweder werden die jeweiligen Ergebnisse direkt in einen Ausgabestrom geleitet oder sie werden nochmals in einer Warteschlange gepuffert, um beispielsweise sortiert ausgegeben zu werden.

8 Fazit

Die Grundkonzepte der klassischen Anfrageverarbeitung können für DSMS in einigen Bereichen effektiv eingesetzt werden. Trotzdem erzwingen es gerade die dynamischen Aspekte, die für das Management von Datenströmen essentiell sind, dass neue Algorithmen entwickelt und in der Praxis erprobt werden müssen. Mit dem vorgestellten Bewertungsmodell existiert bereits ein guter Ansatz, um die Anfrageverarbeitung in DSMS effektiv und kontrollierbar verbessern zu können.

In den gemachten Betrachtungen wurde aber stets eine mögliche Multi-Query-Optimierung ausgeklammert. Ebenso kann in DSMS nicht davon ausgegangen werden, dass ein zu Beginn möglichst optimaler Anfrageplan im Laufe einer Anfrage auch vorteilhaft bleibt. Deshalb müssen Optimierungsbetrachtungen auch auf die anderen Anfragen im System und die Laufzeit der Anfragen ausgeweitet werden, um dynamische Anpassbarkeit zu ermöglichen. Ebenso sind die Reaktion des Systems bei Überbelastung sowie die Realisierung einer garantierbaren Ergebnisqualität weitere wichtige Themen in der Entwicklung stabiler, praktisch einsetzbarer Systeme. Für viele dieser Probleme existieren auch schon unterschiedliche Ansätze. Die Hauptaufgabe für die Konstruktion von DSMS wird somit in Zukunft die Integration und Abstimmung der einzelnen, sich als praxistauglich erwiesenen Konzepte sein, um eine einheitlich agierende, dynamische Anfrageverarbeitung zu ermöglichen.

Literatur

[ABB+] Arasu, Arvind ; Babcock, Brian ; Babu, Shivnath ; Datar, Mayur ; Ito, Keith ; Nishizawa, Itaru ; Rosenstein, Justin ; Widom, Jennifer: *STREAM: The Stanford Stream Data Manager*. http://dbpubs.stanford.edu:8090/pub/2004-20. – Online–Ressource, Abruf: 22. Mai 2004

[ACC+03] Abadi, Daniel J. ; Carney, Don ; Cetintemel, Ugur ; Cherniack, Mitch ; Convey, Christian ; Lee, Sangdon ; Stonebraker, Michael ; Tatbul, Nesime ; Zdonik, Stan: Aurora: A New Model and Architecture for Data Stream Management. In: *The VLDB Journal* 12 (2003), Nr. 2, S. 120–139

[BBD+02] Babcock, Brian ; Babu, Shivnath ; Datar, Mayur ; Motwani, Rajeev ; Widom, Jennifer: Models and Issues in Data Stream Systems. In: *Proceedings of the 21th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS)*, 2002, S. 1–16

[CHKS04] Cammert, Michael ; Heinz, Christoph ; Krämer, Jürgen ; Seeger, Bernhard: Anfrageverarbeitung auf Datenströmen. In: *Datenbank-Spektrum* 4 (2004), Nr. 11, S. 5–13

[CHKS05] Cammert, Michael ; Heinz, Christoph ; Krämer, Jürgen ; Seeger, Bernhard: Sortierbasierte Joins über Datenströmen. In: *Tagungsband der GI-Fachtagung „Datenbanksysteme in Business, Technologie und Web (BTW)“*, 2005, S. 365–384

[DMRH04] Ding, Luping ; Mehta, Nishant ; Rundensteiner, Elke A. ; Heineman, George T.: Joining Punctuated Streams. In: *Proceedings of the 9th International Conference on Extending Database Technology (EDBT)* Bd. 2292, 2004, S. 587–604

[DR04] Ding, Luping ; Rundensteiner, Elke A.: Evaluating window joins over punctuated streams. In: *Proceedings of the 13th ACM Conference on Information and Knowledge Management (CIKM)*, 2004, S. 98–107

[DRH03] Ding, Luping ; Rundensteiner, Elke A. ; Heineman, George T.: MJoin: A Metadata-Aware Stream Join Operator. In: *Proceedings of the 2nd International Workshop on Distributed Event-Based Systems (DEBS)*, 2003, S. 1–8

[DSTW02] Dittrich, Jens-Peter ; Seeger, Bernhard ; Taylor, David S. ; Widmayer, Peter: Progressive Merge Join: A Generic and Non-blocking Sort-Based Join Algorithm. In: *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB)*, 2002, S. 299–310

[KNV03] Kang, Jaewoo ; Naughton, Jeffrey F. ; Viglas, Stratis D.: Evaluating Window Joins over Unbounded Streams. In: *Proceedings of the 19th International Conference on Data Engineering (ICDE)*, 2003, S. 341–352

[LC86] Lehman, Tobin J. ; Carey, Michael J.: A Study of Index Structures for Main Memory Database Management Systems. In: *Proceedings of the 12th International Conference on Very Large Data Bases (VLDB)*, 1986, S. 294–303

[Mit95] Mitschang, Bernhard: *Anfrageverarbeitung in Datenbanksystemen: Entwurs- und Implementierungskonzepte*. Braunschweig, Wiesbaden : Vieweg, 1995

[TMSF03] Tucker, Peter A. ; Maier, David ; Sheard, Tim ; Fegaras, Leonidas: Exploiting Punctuation Semantics in Continuous Data Streams. In: *IEEE Transactions on Knowledge and Data Engineering* 15 (2003), Nr. 3, S. 555–568

[UF99] Urhan, Tolga ; Franklin, Michael J.: XJoin: Getting Fast Answers From Slow and Bursty Networks / Computer Science Department, University of Maryland. 1999 (Technical Report CS-TR-3994, UMIACS-TR-99-13)

[UF00] Urhan, Tolga ; Franklin, Michael J.: XJoin: A Reactively-Scheduled Pipelined Join Operator. In: *IEEE Data Engineering Bulletin* 23 (2000), Nr. 2, S. 27–33

[VN02] Viglas, Stratis D. ; Naughton, Jeffrey F.: Rate-Based Query Optimization for Streaming Information Sources. In: *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, 2002, S. 37–48