Sortierbasierte Joins über Datenströmen

Michael Cammert, Christoph Heinz, Jürgen Krämer, Bernhard Seeger

Philipps-Universität Marburg
{cammert, heinzch, kraemerj, seeger}@informatik.uni-marburg.de

Abstract: Der effektiven Berechnung von Joins kommt auch in der Datenstromverarbeitung essentielle Bedeutung zu. In dieser Arbeit adaptieren wir daher die für eine Vielzahl verschiedener Joinprädikate geeignete sortierbasierte Joinverarbeitung von der passiven auf die aktive Datenverarbeitung. Wir zeigen auf, wie man das Problem der Verarbeitung potentiell unbegrenzter Datenströme mit Hilfe einer exakten Zeitfenstersemantik löst. Zu deren Realisierung stellen wir verschiedene Haupt- und Externspeicheralgorithmen vor. Abschließend erweitern wir die vorgestellten Operatoren zur Berechnung mehrdimensionaler Joins und zeigen experimentell die Vorteile dieses Ansatzes gegenüber binär kaskadierten Joins auf.

1 Einleitung

Die Notwendigkeit der direkten Verarbeitung von Datenströmen [GO03a] wird durch eine Vielzahl von Applikationen motiviert, wie z. B. der kontinuierlichen Auswertung von Sensordaten oder der Überwachung von Netzwerkdatenverkehr. Durch die zu Grunde liegenden autonomen Datenquellen, die i. A. kontinuierlich Daten transferieren, entstehen immense Datenvolumina, die eine persistente Speicherung in einem Datenbankmanagementsystemen (DBMS) mit einer triggerbasierten Auswertung ausschließen [ACG$^+$04]. Vielmehr ist man daran interessiert, die eintreffenden Daten direkt bei deren Ankunft im System zu verarbeiten, und den Externspeicher nur dann zu nutzen, wenn die Hauptspeicher- und Verarbeitungskapazitäten, etwa aufgrund hoher Datenraten, nicht ausreichen.

In Datenstrommanagementsystemen (DSMS) werden im Gegensatz zu den traditionellen DBMS primär Anfragen verarbeitet, die für lange Zeit im System verweilen. Um bei der Anfrageverarbeitung gemeinsame Teilanfragen nutzen zu können, werden in DSMS azyklische gerichtete Operatorgraphen eingesetzt. Die Semantik der Operatoren orientiert sich dabei an der erweiterten relationalen Algebra, die sich für relationale DBMS etabliert hat.

Für den speziellen Fall des Verbundoperators (*Join*) existieren verschiedene physische Realisierungen, d.h. Implementierungen, in einem DBMS. Oftmals werden dafür bestimmte Prämissen gefordert, wie etwa ein auf einer der Relationen angelegter Index. Von fundamentaler Bedeutung ist jedoch das Join-Prädikat, das die zu seiner Auswertung geeigneten Join-Implementierungen signifikant einschränken kann. Viele Konzepte, darunter die wichtigsten hash-basierten Joins, sind auf den Equi-Join (oder auch Natural Join) spezialisiert. Eine einfache Realisierung für beliebige Join-Prädikate liefert der Nested-Loops-

Join, der jedoch für praktische Belange oftmals zu inperformant ist. Zwischen diesen Extremen wurde ein weites Feld an Joinprädikaten identifiziert, die sich besonders effektiv auswerten lassen, wenn die Eingaben einer geeigneten Ordnung folgend sortiert sind. Dies erreicht man durch den Einsatz eines Merge-Joins oder, falls die Eingaben noch nicht sortiert vorliegen, eines Sort-Merge-Joins. Um diese für viele Joinprädikate effizient anwendbare Technik [DS01] auch für die Datenstromverarbeitung zugänglich zu machen, adaptieren wir in dieser Arbeit das in [DSTW02] vorgestellte Rahmenwerk zur sortierbasierten Joinverarbeitung für den Datenstromkontext.

Die in DBMS etablierten Konzepte der passiven Datenverarbeitung lassen sich dabei aufgrund der Datenstromcharakteristika nicht ohne weiteres übertragen. Zu diesen zählt insbesondere die Anforderung der direkten Verarbeitung eintreffender Elemente. Ebenso muss eine nicht blockierende Verarbeitung gewährleistet werden, d.h. Ergebnisse müssen bereits produziert werden, bevor eine Eingabe komplett vorliegt. Das ist eine fundamentale Einschränkung, da die meisten sortierbasierten Operatoren ihre Eingaben zunächst vollständig sortieren. Mit dem Progressive-Merge-Join (PMJ) [DSTW02, DSTW03] wurde eine Lösung vorgestellt, bei der Teile des Joinergebnisses bereits während der Merge-Schritte beim Sortieren berechnet werden. Eine geeignete Adaption dieses Verfahrens auf Datenströme muss jedoch berücksichtigen, dass das Eintreffen und Verarbeiten neuer Elemente mit den ressourcenaufwändigen Mergeschritten koordiniert werden muss.

Als ein weiteres Problem stellt sich die potentielle Unbeschränktheit von Datenströmen heraus. Für eine exakte Berechnung aller potentiellen Join-Ergebnisse müsste man alle Elemente der beteiligten Ströme speichern. Üblicherweise löst man dieses Problem durch die Einführung so genannter gleitender Fenster (*sliding windows*) über den Datenströmen. In diesen wird kontinuierlich eine endliche Teilmenge der Datenströme gespeichert. Für die exakte Definition eines Zeitfensters existieren verschiedene Ansätze, wobei wir uns auf die in [KS05] vorgestellte, deterministische Zeitfenstersemantik konzentrieren. Deren wesentliche Idee ist die Verwendung logischer und insbesondere endlicher Gültigkeitsintervalle für die einzelnen Elemente. Eine darauf definierte Ordnung gestattet es uns, das Join-Ergebnis bezüglich der Zeitfenstersemantik unter Einsatz endlicher Ressourcen zu berechnen.

Unser Ansatz zur Realisierung von sortierbasierten Joins über Datenströmen ist in PIPES [KS04] integriert, einem bibliotheksorientierten Ansatz zur Konstruktion applikationsspezifischer DSMS. Zusammenfassend stellen wir das erste sortierbasierte Join-Verfahren für Datenströme und zwei sortierbasierte Algorithmen vor, die eine Join-Semantik unter Verwendung von Zeitfenstern realisieren.

Der Rest dieser Arbeit ist wie folgt aufgebaut: In Abschnitt 2 stellen wir verwandte Arbeiten vor, bevor wir uns in Abschnitt 3 ausführlich der sortierbasierten Joinverarbeitung in DBMS und insbesondere dem PMJ widmen, den wir in Abschnitt 4 für Datenströme adaptieren. Abschnitt 5 stellt unsere Semantik zur Berechnung von Joins über Zeitfenstern auf Datenströmen und zwei Algorithmen zu deren Implementierung vor. Abschnitt 6 verallgemeinert die Algorithmen für mehrdimensionale Joins. Bevor in Abschnitt 8 ein Fazit der Arbeit gezogen wird, zeigen in Abschnitt 7 Experimente die Performanz und Tauglichkeit der vorgestellten Verfahren.

2 Verwandte Arbeiten

Verschiedene Arbeiten befassen sich mit Joins auf Datenströmen. Die wesentlichen Arbeiten zur Optimierung von Join-Algorithmen beruhen dabei auf Hashing. Die grundlegende Idee liefert dabei der Symmetric-Hash-Join [WA93]. Dieser baut über jeder Eingabe eine Hashtabelle auf und verarbeitet die Eingaben symmetrisch, indem eintreffende Elemente in die zur eigenen Eingabe gehörige Hashtabelle eingefügt werden und die jeweils andere nach Joinpartnern durchsucht wird. Mit dem XJoin [UF00] wurde dieses Prinzip für den Fall verzögert eintreffender Elemente erweitert. Man lagert Teile der Hashtabelle auf den Externspeicher aus und nutzt Zeiten geringer Systemlast für deren Verarbeitung. Dieses Konzept wurde mit dem MJoin [VNB03] verfeinert und auf mehrdimensionale Joins adaptiert. [GO03b] behandelt mehrdimensionale Joins über gleitenden Fenstern und vergleicht Hash- und Nested-Loops-Implementierungen. Für diese wird in [KNV03] ein Kostenmodell für den binären Fall angegeben.

Der in [DSTW02, DSTW03] vorgestellte Progressive Merge Join (PMJ) ist der erste nicht blockierende sortierbasierte Joinoperator und wird aufgrund seiner Relevanz für diese Arbeit in Abschnitt 3.2 ausführlicher diskutiert. Der Hash-Merge-Join [MLA04] ist ein hybrider Algorithmus, der den verfügbaren Hauptspeicher analog zum XJoin nutzt, aber für ausgelagerte Elemente den PMJ verwendet.

3 Sortierbasierte Joinverarbeitung in DBMS

Im Folgenden stellen wir zuerst ein Rahmenwerk für sortierbasierte Joinverarbeitung vor, um dann genauer auf den PMJ einzugehen.

3.1 Rahmenwerk für sortierbasierte Joinverarbeitung

Die wesentliche Idee des in [DSTW02] vorgestellten Rahmenwerks zur sortierbasierten Joinverarbeitung beruht auf einer geeigneten globalen Ordnung auf den Definitionsbereichen der Eingaben. Diese muss die Eigenschaft haben, dass zwei Elemente, die sich bezüglich des Joinprädikats als Ergebnis qualifizieren, mit hoher Wahrscheinlichkeit nahe beieinander liegen.

Anhand eines einfachen binären Sort-Merge-Joins (SMJ) veranschaulichen wir das Prinzip des Rahmenwerks. Zunächst werden die Eingaben getrennt nach der vorgegebenen globalen Ordnung aufsteigend sortiert und dann dieser Ordnung folgend zusammengeführt. Dazu wird zunächst pro Eingabe eine noch näher zu beschreibende Datenstruktur namens *SweepArea* angelegt. Anschließend erfolgt eine synchronisierte Verarbeitung der sortierten Eingaben, wobei jeweils das nächste zu verarbeitende Element nach der globalen Ordnung gewählt wird. Für dieses werden drei Schritte durchgeführt:

- Einfügen: Das Element wird in die seiner Eingabe zugehörige SweepArea eingefügt.

- Reorganisation: Das Element wird benutzt, um diejenigen Elemente aus den SweepAreas zu entfernen, die aufgrund seines Auftretens beim sortierten Einlesen nicht mehr zu weiteren Ergebnissen beitragen können.
- Anfragen: Man sucht mit dem aktuellen Element in der anderen SweepArea nach Elementen, die mit ihm das Join-Prädikat erfüllen und gibt diese zurück.

Diese abstrakte Funktionalität wird mit einer SweepArea modelliert. Für ein gegebenes Joinprädikat P_{join} muss man die Operationen Anfragen und Reorganisieren geeignet konkretisieren. Für die Semantik des Anfragens folgt dabei: Wenn mit einem Element x der linken Eingabe die SweepArea der rechten angefragt wird, muss diese genau die Elemente y zurückliefern, mit denen das Joinprädikat P_{join} erfüllt wird, d.h. $P_{join}(x, y) = true$. Anfragen mit Elementen der rechten Eingabe werden symmetrisch behandelt.

Für den Reorganisationsschritt definieren wir ein zusätzliches Prädikat P_{rm}^{E} für jede Eingabe $E \in \{li, re\}$. Wenn das Element x der linken Eingabe verarbeitet wird, entfernt man alle Elemente y aus der rechten SweepArea E^{re}, die mit x das Prädikat der rechten Eingabe erfüllen, d.h. $P_{rm}^{re}(x, y) = true$. Der Fall eines Elementes der rechten Eingabe wird wieder symmetrisch behandelt. Die Prädikate müssen so gewählt werden, dass nur die Elemente entfernt werden, die später nicht mehr zu Ergebnissen beitragen können. Hierbei nutzt man wieder die globale Sortierung aus. Formal erhält man folgende Bedingungen:

$$\forall\, w \geq v : P_{rm}^{li}(v, u) \Rightarrow \neg P_{join}(u, w) \tag{1}$$

$$\forall\, w \geq v : P_{rm}^{re}(v, u) \Rightarrow \neg P_{join}(w, u) \tag{2}$$

Werden diese Bedingungen sichergestellt, so erhält man mit obigem Algorithmus ein vollständiges und korrektes Ergebnis [Cam02, DSTW02]. Dabei gelten die Bedingungen immer für $P_{rm}^{E}(x, y) := false$. Das korrespondiert mit der Tatsache, dass in keinem Fall Ergebnisse verloren gehen, wenn man nie Elemente aus der SweepArea entfernt.

Als Beispiel betrachten wir den Intervall-Join, bei dem zwei halboffene Intervalle $[a_1, b_1)$ und $[a_2, b_2)$ das Joinprädikat erfüllen, wenn sie sich schneiden, also $[a_1, b_1) \cap [a_2, b_2) \neq \emptyset$, gilt. Die Ordnung auf halboffenen Intervallen definiert man aufsteigend primär auf dem Intervallbeginn und sekundär auf dem Intervallende. Für ein neu eintreffendes Intervall $[a, b)$ stellt sich die Frage, welche Elemente aus den SweepAreas entfernt werden können. Alle nachfolgend eintreffenden Elemente müssen aufgrund der globalen Ordnung von der Form $[a', b')$ mit $a' \geq a$ sein und werden daher nur Intervalle schneiden, deren Ende $\tilde{b}$ die Bedingung $\tilde{b} > a$ erfüllt. Daher können alle Intervalle $[\tilde{a}, \tilde{b})$ mit $\tilde{b} \leq a$ aus den SweepAreas entfernt werden.

3.2 Progressive Merge Join (PMJ)

Der Progressive-Merge-Join (PMJ) [DSTW02, DSTW03] für die passive Datenverarbeitung behebt das Problem, dass der Sort-Merge-Join, weil er in seiner ersten Phase die Eingaben nacheinander komplett sortiert, blockierend ist. Hierzu wird als externer Sortieral-

gorithmus i. A. der externe Merge-Sort eingesetzt. Zur Sortierung füllt dieser den Hauptspeicher mit Elementen einer Eingabe, die er sortiert und als sortierte Teilfolgen (auch *Runs* genannt) auf den Externspeicher zurückschreibt (*Run-Generierung*). Diese werden dann zu längeren Runs zusammengeführt, indem jeweils möglichst viele von ihnen synchronisiert bezüglich der Ordnung gelesen und direkt wieder herausgeschrieben werden. Den begrenzenden Faktor bildet dabei die Zahl der Externspeicherseiten, auch *Fan-In* genannt, die gleichzeitig gelesen werden können.

Die wesentliche Idee des PMJ ist die Verschmelzung der ähnlich ablaufenden Merge-Schritte des externen Sortierens und des Merge-Joins. In beiden Merge-Schritten werden die zugrundeliegenden Daten synchronisiert verarbeitet, indem das nächste zu verarbeitende Element anhand der globalen Ordnung ausgewählt wird. Für das Sortieren wählt man das kleinste Element aus verschiedenen Runs *einer* Eingabe. Beim Merge-Join wählt man das kleinste Element aus je einem Run *mehrerer* Eingaben. Der Joinschritt kann allerdings auch durchgeführt werden, wenn man mehrere Runs pro Eingabe erlaubt. Und da eine nach der globalen Ordnung ablaufende Abarbeitung mehrerer Eingaben insbesondere auch jede Eingabe lokal sortiert (nach derselben Ordnung) liest, kann dabei ebenfalls wieder ein längerer sortierter Run pro Eingabe erzeugt werden. Insgesamt erhält man einen Merge-Schritt (*Multiple-Merge*), der mehrere Runs pro Eingabe verarbeitet und daraus den kompletten Join und einen sortierten Run pro Eingabe erzeugt.

Die Rungenerierung erfolgt beim PMJ nun nicht mehr getrennt nach Eingaben. Der Hauptspeicher wird vielmehr zu je einem Teil mit Elementen *aller* Eingaben gefüllt, die dann zunächst jeweils sortiert werden. Bevor sie jedoch als Runs nach Eingabe getrennt auf den Externspeicher geschrieben werden, wird noch der Join dieser sortierten Teile der Eingaben berechnet und als Ergebnis weitergeleitet. In der Merge-Phase des PMJ kommt dann der Multiple-Merge zum Einsatz. Dabei wird der Fan-In unter den Eingaben aufgeteilt, und es werden jeweils der Join aller beteiligten Eingaberuns und längere Ausgaberuns produziert. Die sich dabei ergebende Struktur des Verschmelzens von Runs bezeichnen wir als *Merge-Baum*. Die Aufteilung des Hauptspeichers zwischen den Eingaben wird ausführlich für beide Schritte in [Cam02] diskutiert.

Prinzipiell berechnet der letzte Merge-Schritt, bei dem alle Elemente der Eingaben in einem Run enthalten sind, das komplette Joinergebnis. Das folgt aus der Korrektheit des Join-Rahmenwerks. Es ergibt sich aber das Problem auftretender Duplikate, da manche Joinergebnisse schon in vorherigen Merge-Schritten als Ergebnis identifiziert wurden. Um das zu vermeiden, könnte man alle Joinergebnisse erst im letzten Schritt herausgeben, was jedoch keinen Vorteil mehr gegenüber dem Sort-Merge-Join bringen würde. Man möchte Ergebnisse möglichst früh produzieren, also sofort bei deren erstem Auftreten. Dieses erfolgt, wenn die beteiligten Eingabeelemente beim Sortieren gemeinsam im Hauptspeicher vorlagen oder wenn sie zum ersten Mal gemeinsam an einem Merge-Schritt beteiligt sind. In beiden Fällen werden sie in Runs gespeichert, die im selben Schritt erzeugt werden. Man stellt nun sicher, dass derart zusammengehörige Runs in den folgenden Merge-Schritten nur gemeinsam verarbeitet werden. Um ein Ergebnis als Duplikat zu identifizieren, muss man dann nur feststellen, ob die beteiligten Elemente aus zusammengehörigen Runs kommen. Dieses Vorgehen gestattet eine effektive Duplikateliminierung und garantiert korrekte und vollständige Ergebnisse des PMJ [DSTW02, Cam02].

4 PMJ für Datenströme

In diesem Abschnitt werden zunächst die Anforderungen der Datenstromverarbeitung skizziert. Davon ausgehend stellen wir eine Adaption des PMJ von der passiven auf die aktive Datenverarbeitung von DSMS vor.

4.1 Anforderungen aktiver Datenverarbeitung

Ein Datenstrom entsteht durch den kontinuierlichen Versand von Daten durch eine autonome Datenquelle. Um Datenströme und insbesondere Anfragen darauf zu verarbeiten, muss man die Charakteristika dieser autonomen Datenquellen berücksichtigen. Das führt zu den folgenden Annahmen:

- Die Verarbeitung der Anfragen erfolgt direkt beim Empfang der Daten, ohne zuvor den gesamten Datenstrom explizit zu speichern.
- Die Ordnung auf den Datenströmen sowie deren Übertragungsgeschwindigkeit wird nicht durch die verarbeitenden Algorithmen, sondern primär durch die autonomen Datenquellen selbst bestimmt.
- Die Datenströme sind in ihrer Größe potentiell unbeschränkt.
- Jedes Element eines Datenstromes wird nur einmal vom Strom geliefert.
- Änderungsoperationen auf Elementen eines Datenstromes sind nicht möglich. Neue Elemente können nur am Ende eines Datenstromes angehängt werden.

Daraus ergeben sich direkte Konsequenzen für die Operatoren zur Verarbeitung von Datenströmen. Dabei verstehen wir unter aktiver oder datengetriebener Verarbeitung die Eigenschaft von Operatoren, Elemente von Datenströmen direkt bei deren Ankunft zu verarbeiten. Dadurch ist es möglich, dass Elemente aus unterschiedlichen Datenströmen einen mehrdimensionalen Operator, wie z. B. einen Join, zeitgleich erreichen und dann sofort verarbeitet werden müssen. Dieses Verarbeitungsprinzip unterscheidet sich signifikant von der bedarfsgesteuerten Verarbeitung in traditionellen DBMS, die üblicherweise auf dem Iteratorkonzept beruht.

Darüber hinaus dürfen Operatoren nicht blockierend implementiert sein, d.h. sie müssen insbesondere in der Lage sein, zur Laufzeit kontinuierlich Ergebnisse zu produzieren. Außerdem besitzt man keinen wahlfreien Zugriff auf die Elemente, sondern man muss sie in der vom Datenstrom vorgegebenen Reihenfolge verarbeiten.

Für die konkrete Implementierung eines Operators muss man zuerst dessen Semantik definieren. Diese lässt sich wieder über die vorgegebene Reihenfolge eines Datenstroms ausdrücken [KS05]. Ausgangspunkt dafür ist die erweiterte relationale Algebra. Da diese auf endlichen Multimengen (Relationen) beruht, Datenströme aber potentiell unbegrenzt sind, betrachtet man alle endlichen Teilfolgen eines Datenstroms. Über diesen definiert man die

Semantik der Operatoren gemäß der erweiterten relationalen Algebra. Auf diese Weise ergibt sich eine wohldefinierte Semantik für die fundamentalen Operationen eines DSMS, die wir im Folgenden verwenden.

4.2 PMJ für Datenströme

Der PMJ als sortierbasierter Joinoperator kann bei der Verarbeitung des Joinprädikates enorm von einer Sortierung nach einer geeigneten Ordnung profitieren. Dabei ist er in der Lage, verschiedenste Joinprädikate effizient zu unterstützen, darunter insbesondere solche, die nicht oder nur mit erheblichem Mehraufwand durch hash-basierte Verfahren unterstützt werden [DS01]. Die Adaption des PMJ auf Datenströme erlaubt daher den effizienten Einsatz einer Vielzahl von Joinprädikaten, die zuvor in DSMS nur schlecht unterstützt wurden.

Joinoperatoren auf Datenströmen müssen sicherstellen, dass neu eintreffende Elemente direkt verarbeitet werden. Die Elemente müssen jedoch auch in geeigneter Form gespeichert werden, da später eintreffende Elemente eventuell als Joinpartner in Frage kommen. So sucht der Symmetric-Hash-Join [WA93] als ein möglicher Joinoperator beim Eintreffen eines neuen Elements direkt dessen Joinpartner unter den bereits gespeicherten Elementen. Zur sortierbasierten Joinberechnung müssen wir aber zunächst sortierte Teilfolgen generieren. Dies gestattet uns, unter Verwendung des in Abschnitt 3.1 vorgestellten Rahmenwerks deren Join zu berechnen und sie als Runs für die externspeicherbasierte Weiterverarbeitung zu speichern. Ein Parameter bei diesem Vorgehen ist die Wahl eines geeigneten Sortierverfahrens. Berücksichtigt man das sukzessive Eintreffen der Elemente, so wäre Insertion Sort eine Möglichkeit. Dieses Verfahren scheidet jedoch durch sein ungünstiges Verhalten im schlechtesten Fall aus.

Als weiteres inkrementelles Sortierverfahren bietet sich, insbesondere im Umfeld des externen Sortierens, Replacement-Selection an. Das Verfahren liefert im Mittel Runs, die doppelt so groß wie der benutzte Teil des Hauptspeichers sind. Eine Adaption für den PMJ gestaltet sich jedoch schwierig. Zunächst muss man pro Eingabe einen Heap verwalten. Beim Eintreffen eines neuen Elements verarbeitet man das kleinste Element aller Heaps, wobei man es als potentiellen Joinpartner einsetzt und dann herausschreibt. Das kleinste Element gehört aber nichts zwangläufig zur selben Eingabe wie das eingetroffene Element. Folglich müsste man zur Laufzeit Speicher zwischen den Heaps umverteilen. Hinzu kommt noch die Problematik, dass Replacement-Selection empfindlich auf Elemente verschiedener Größe reagiert [LG98].

In unserer Umsetzung haben wir uns für ein einfaches und effektives Vorgehen entschieden, das einen dem Joinoperator vorgegebenen maximalen Speicherverbrauch berücksichtigt. Trifft ein neues Element ein, so wird es zunächst an eine seiner Eingabe zugeordneten Liste angehängt. Stößt die Summe der Längen aller Listen an eine vorher festgelegte Speichergrenze, so sortiert man die Listen (mit Quicksort), berechnet ihren Join und schreibt sie auf den Externspeicher. Während des Sortiervorgangs können jedoch weitere zu verarbeitende Elemente eintreffen. Aus diesem Grund werden bereits beim Erreichen eines

bestimmten Anteils der Speichergrenze neue Listen zur Aufnahme der Eingabeelemente angelegt. Die Sortierung und Verarbeitung der alten Listen wird in einem separaten Thread vorgenommen. Um die Auslagerung sicher zu stellen und so nicht an die Speichergrenze zu stoßen, ist der Thread mit einer hohen Priorität versehen.

Das beschriebene Vorgehen hat zudem den Vorteil, dass es adaptiv hinsichtlich verschiedener und möglicherweise stark schwankender Datenraten der Eingaben ist. Stößt man an die Speichergrenze, so sortiert man und schreibt Runs auf den Externspeicher. Dadurch werden für Eingaben mit aktuell höherer Datenrate auch im Verhältnis längere Runs erzeugt. Die Tatsache, dass somit für jede Eingabe gleich viele Runs erzeugt werden, stellt eine gute Voraussetzung für den Merge-Schritt des PMJ dar.

Für den Fall endlicher Datenströme wäre eine Strategie, die Merge-Phase des passiven PMJs anzuwenden, nachdem alle Elemente der Eingaben in Runs auf dem Externspeicher vorliegen. Merge-Schritte in höheren Ebenen des Merge-Baumes erzeugen jedoch wegen der größeren Anzahl beteiligter Eingabeelemente mehr frühe Ergebnisse als solche in tieferen Ebenen. Somit eignet sich, wie schon beim klassischen PMJ, ein Postorder-Durchlauf eines Merge-Baums mit mindestens zwei Ebenen besser als ein Levelorder-Durchlauf [DSTW03].

Der Fan-In ist analog zum passiven Fall der limitierende Faktor für die Anzahl der Runs, die in einem Schritt verarbeitet werden können. Die konkrete Wahl des Fan-In hängt vom für den Joinoperator verfügbaren Hauptspeicher ab. Ein Teil dessen muss dabei für die Sortierschritte der eintreffenden Elemente reserviert werden. Ist die Anzahl der maximal verarbeitbaren Runs noch nicht erreicht, kann sich unter Umständen ein Merge-Schritt trotzdem schon lohnen. Im Hinblick auf schwankende Datenraten der beteiligten Ströme können sich nämlich Phasen geringer Systemlast ergeben. Führt man in diesen Merge-Schritte durch, so produziert man zum einen Ergebnisse und reduziert zum anderen die Zahl der Runs (und dadurch auch den verbleibenden Sortieraufwand). Um dies zu ermöglichen, ist der Merge-Schritt ebenfalls in einem separaten Thread realisiert. Dieser erhält eine geringere Priorität als der für die Sortierung initialer Runs, da die Elemente auf dem Externspeicher liegen und keine Ergebnisse verloren gehen, auch wenn längere Zeit kein Merge-Schritt durchgeführt wird. Die Ergebnisse werden lediglich später erzeugt, was in Phasen hoher Systemlast sogar von Vorteil sein kann, da die Joinergebnisse so erst später weiterverarbeitet werden müssen.

Eine Merge-Phase sollte möglichst komplett durchgeführt und nicht aufgrund auftretender Speicherknappheit abgebrochen werden. Tritt dies dennoch ein, kann man zunächst die Seiten zum Einlesen der Runs freigeben und sich stattdessen nur die aktuelle Seitennummer und die Position darin zu merken. Neben diesen Seiten und dem Sortieren benötigen die in Abschnitt 3.1 vorgestellten SweepAreas ebenso Hauptspeicher. Der Speicherbedarf dieser Datenstrukturen steigt im Gegensatz zum Speicherbedarf für die Seiten zum Lesen der Runs mit der Höhe des Merge-Knotens. Daher sollten Joins auf höheren Ebenen ggf. mit geringerem Fan-In und nur dann durchgeführt werden, wenn hinreichend viel Speicher zur Verfügung steht.

Letztendlich bietet sich eine Vielzahl an Möglichkeiten, die Anzahl der für einen Merge-Schritt zu verwendenden Runs zu wählen. Allerdings sollten Runs verschiedener Einga-

ben, die gemeinsam auf den Externspeicher geschrieben wurden, auch gemeinsam wieder eingelesen werden, um die vorgestellte Technik zur Duplikatelimierung verwenden zu können. Um die konkrete Wahl flexibel zu gestalten, lässt sich in unserer Implementierung die Auswahl der zu verschmelzenden Runs über eine Strategie steuern, die als Parameter übergeben wird. Eine konkrete Strategie legt demnach für jede Merge-Phase fest, welche Runs verarbeitet werden.

5 Joins über Zeitfenstern auf Datenströmen

Die potentielle Unbegrenztheit von Datenströmen hat zur Folge, dass man bei einem Join jedes Element der Eingabeströme speichern muss, um mögliche Joinergebnisse mit später eintreffenden Joinpartnern sicher zu stellen. Dadurch stößt der in Abschnitt 4.2 vorgestellte Algorithmus an seine Grenzen. Für den Fall einer hinreichend niedrigen Systemlast kann dieser den Join der bis dato konsumierten Eingabeströme berechnen. Die Länge der einzelnen Runs steigt jedoch proportional zur Anzahl der bereits konsumierten Elemente. Folglich wird der verfügbare Hauptspeicher irgendwann nicht mehr ausreichen, um den Join der längsten Runs zu berechnen. Das Verfahren eignet sich deswegen nur für endliche Datenströme, deren Länge aber vorher nicht bekannt sein muss.

Dem Problem des Joins über unendlichen Datenströmen haben sich mittlerweile verschiedene Ansätze gewidmet [GO03b, KNV03, ZRH04]. Da die Berechnung des kompletten Joinergebnisses nicht möglich ist, wird im Allgemeinen eine Einschränkung der Ergebnismenge vorgeschlagen. Hierfür empfiehlt sich der Einsatz von über den Datenstrom gleitenden Fenstern (*sliding windows*). Ein solches Fenster kann kontinuierlich in den internen Speicherstrukturen eines Join-Algorithmus gehalten werden. Ein einfacher Ansatz basiert auf Fenstern fester Größe. Diese werden mit den ersten Elementen gefüllt. Trifft danach ein neues Element ein, so wird das älteste Element im Fenster verdrängt. Die daraus resultierenden Joinergebnisse hängen jedoch von der Reihenfolge, in der die Elemente eintreffen, ab [KS05]. Dieser unerwünschte Effekt wird bei unserem im Folgenden vorgestellten Ansatz, der auf einer Zeitfenstersemantik beruht, vermieden.

5.1 Zeitfenstersemantik

Die Semantik fensterbasierter Operatoren basiert im Allgemeinen auf der Verwendung von Zeitstempeln. Neu eintreffende Elemente erhalten dabei einen synthetischen Zeitstempel (z. B. Ankunftszeit im System), falls sie nicht schon implizit über einen verfügen (z.B. den Zeitpunkt ihrer Messung). In beiden Fällen werden die Zeitstempel während ihrer Verarbeitung als logische Einheiten anstelle einer physischen Systemzeit aufgefasst.

Das Joinergebnis betreffend konzentriert man sich auf Elemente, die zeitlich nahe beieinander liegen, und somit auf eine Teilmenge des kompletten Ergebnisses. Dabei nutzt man die Information der Zeitstempel aus. In unserem Ansatz [KS05] erhält jedes Element ein halboffenes logisches Gültigkeitsintervall $[t_s, t_e)$, währenddessen das Element gültig ist.

Wir bezeichnen t_s als *Startzeitstempel* und t_e als *Endzeitstempel*. Damit die Operatoren der Zeitfenstersemantik genügen, müssen die Ergebnisse die aus dem Bereich temporaler Datenbanken bekannte Schnappschuss-Reduzierbarkeit (*Snapshot-Reducibility*) erfüllen. Darunter versteht man die Gleichheit zweier Mengen. Das ist zum einen die Menge der zum logischen Zeitpunkt t gültigen Ausgabeelemente. Zum anderen betrachtet man die Ausgabemenge, die entsteht, wenn man den entsprechenden Operator ohne Zeitfenstersemantik auf die Menge der zum Zeitpunkt t gültigen Eingabeelemente anwendet. Sei also Op ein Operator über Datenströmen und Op_w sein Gegenstück für die Zeitfenstersemantik. Sei weiter π_t die Projektion eines mit Gültigkeitsintervallen behafteten Datenstromes S_i auf die Relation R_i der zum logischen Zeitpunkt t gültigen Elemente (ohne Gültigkeitsintervalle). Für die Erfüllung der Schnappschuss-Reduzierbarkeit muss gelten:

$$\pi_t(Op_w(S_1, \ldots, S_n)) = Op(\pi_t(S_1), \ldots, \pi_t(S_n)) \tag{3}$$

Als Beispiel betrachten wir eine Selektion über einem Strom von natürlichen Zahlen. Das Selektionsprädikat teste, ob ein Element gerade ist. Der Strom enthalte die Elemente $(1, [4, 17))$ (also die Zahl 1 mit Gültigkeitsintervall 4 bis 17) und $(42, [8, 20))$. Die Selektion ohne Zeitfenstersemantik wird den Wert 1 nicht weitergeben und die mit Zeitfenstersemantik somit auch nicht. Der Wert 42 qualifiziert sich als Ergebnis und ist im Zeitintervall [8,20) gültig, weswegen im Ausgabestrom der Selektion wieder $(42, [8, 20))$ geliefert wird. Für eine ausführliche Diskussion der Zeitfenstersemantik sei auf [KS05] verwiesen.

Die Verwendung eines einzigen Zeitstempels pro Element zur Definition der Semantik eines Joins wirft Probleme auf. Es stellt sich nämlich die Frage, welchen Zeitstempel ein Joinergebnis erhält. Es ist nahe liegend, den minimalen oder maximalen Zeitstempel der beteiligten Elemente zu wählen. Damit ist die Definition einer exakten Semantik jedoch schwer zu realisieren.

Unter Verwendung unserer Zeitfenstersemantik ist klar definiert, welche Elemente in einem Schnappschuss der Ausgabe enthalten sein müssen. Das sind genau die Elemente, die beim Join der zum Zeitpunkt des Schnappschusses gültigen Eingabeelemente entstehen. Ein Ergebniselement ist folglich genau zu den logischen Zeitpunkten gültig, zu denen *alle* beteiligten Eingabeelemente gültig sind. Aus diesem Grund wählt man als Gültigkeitsintervall eines Joinergebnisses den Schnitt der Gültigkeitsintervalle der zugehörigen Eingabeelemente. Betrachten wir beispielsweise einen Equi-Join der Ströme $S_1 = ((42, [10, 15)), (3, [11, 14)))$ und $S_2 = ((42, [4, 12)), (3, [17, 22)))$. Während der Wert 42 von Zeitpunkt 10 bis 12 in beiden Strömen gültig ist, ist der Wert 3 zu keinem Zeitpunkt in beiden Strömen gleichzeitig gültig. Die Ausgabe enthält daher nur das Element $(42, [10, 12))$.

Leider beseitigen die bisher getroffenen Definitionen noch nicht das ursprüngliche Problem der Notwendigkeit der unbegrenzten Speicherung der Elemente der Eingaben des Joins. Es kann immer noch ein Element des ersten Stroms mit einem später eintreffenden Element des zweiten Stroms sowohl die Joinbedingung erfüllen als auch einen nichtleeren Schnitt der Gültigkeitsintervalle aufweisen. Daher wird noch eine globale Bedingung an alle zeitintervallbehafteten Datenströme im System gestellt, nämlich dass diese nach den Gültigkeitsintervallen ihrer Elemente sortiert eintreffen müssen. Die Ord-

nung $\leq_l$ auf Gültigkeitsintervallen ist dabei lexikographisch mit $[t_s, t_e) \leq_l [t'_s, t'_e) :\Leftrightarrow t_s < t'_s \vee (t_s = t'_s \wedge t_e \leq t'_e)$ und induziert eine Ordnung $\leq_\tau$ der Elemente mit $(v, [t_s, t_e)) \leq_\tau (v', [t'_s, t'_e)) :\Leftrightarrow [t_s, t_e) \leq_l [t'_s, t'_e)$. Diese Bedingung ermöglicht das Entfernen von Elementen aus den internen Datenstrukturen des Joinalgorithmus, ohne Ergebnisse zu verlieren. So muss man im Fall eines binären Joins ein Element $x = (v, [t_s, t_e))$ nur so lange aufbewahren, bis im anderen Strom ein Element $y = (w, [t'_s, t'_e))$ mit $t'_s \geq t_e$ auftritt und man alle Joinergebnisse von x mit den vor y eingetroffenen Elementen berechnet hat. Da sich das Intervall $[t_s, t_e)$ nicht mit $[t'_s, t'_e)$ schneidet und die Ordnungsbedingung sicherstellt, dass es sich auch nicht mit den Intervallen später eintreffender Elemente schneidet, führt das Löschen zu keinem Ergebnisverlust.

Die Bedingung der Sortierung nach Gültigkeitsintervallen stellt keine Einschränkung an die verarbeitbaren Datenquellen dar. Der Startzeitstempel ist entweder implizit mit dem Element gegeben oder wird synthetisch generiert. Der erste Fall erfordert eventuell noch eine Wiederherstellung der Ordnung der Elemente [SW04], z. B. bei Übertragungsproblemen.

Den Zeitstempel eines gegebenen Elementes fassen wir bei der Umsetzung auf Zeitintervalle als Startzeitstempel auf. Den Endzeitstempel setzen wir auf $+\infty$, womit jedes Element zunächst eine unendliche Gültigkeitsdauer hat. Diese Zuordnung korrespondiert mit der sozusagen genauest möglichen Berechnung von zeitstempelbehafteten Operatoren. Von diesem Optimum weichen wir so weit ab, wie es nötig ist, um mit den dem DSMS zur Verfügung stehenden Ressourcen auszukommen. Dazu wenden wir auf jede Datenquelle einen Zeitfensteroperator an, der den Endzeitstempel auf einen endlichen Wert zurücksetzt. Mit diesem Fensteroperator ist es möglich, die gängigsten Definitionen von Zeitfenstern nachzubilden. Für den Fall gleitender Fenstern bleibt jedes Element für einen festen Zeitraum r gültig. Aus dem Element $(v, [t_s, +\infty))$ wird also $(v, [t_s, t_s{+}r))$. Bei so genannten festen Fenstern (*fixed windows*) wird die Zeitachse in Abschnitte fester Länge m unterteilt. Ein Element erhält als Endzeitstempel das ihm nächste Abschnittsende zugewiesen. Aus dem Element $(v, [t_s, +\infty))$ wird also $(v, [t_s, m * n))$ mit $n = min\{x \mid m * x > t_s\}$. In beiden Fällen werden die monoton eintreffenden Startzeitstempel nicht verändert und die Endzeitstempel monoton steigend vergeben. Daher ist die an die Datenströme gestellte Ordnungsbedingung erfüllt.

5.2 Temporal Join

Um zum Joinergebnis beizutragen, müssen Elemente das Joinprädikat erfüllen und überlappende Gültigkeitsintervalle besitzen. Wenn man Datenströme verarbeitet, die lexikographisch nach Gültigkeitsintervallen geordnet sind, kann man zur sortierbasierten Auswertung der Elemente das in Abschnitt 3.1 vorgestellte Joinrahmenwerk verwenden. Allerdings verarbeitet dieses die Elemente synchronisiert nach der globalen Ordnung. In der datengetriebenen Verarbeitung nach unserer Zeitfenstersemantik sind zwar die Datenströme jeder für sich nach Zeitintervallen sortiert, sie geben aber selber untereinander die Ankunftsreihenfolge vor. Trifft also ein Element x des ersten Stroms ein, so können aus diesem Strom nur noch Elemente x' mit $x' \geq_\tau x$ eintreffen. Möglicherweise liefert

aber ein anderer Strom noch Elemente y mit $y <_\tau x$. Für dieses Problem gibt es zwei prinzipielle Lösungen.

Zum einen kann man die eintreffenden Elemente am Eingang des Joins puffern und global geordnet verarbeiten. Dabei kann immer dann ein Element verarbeitet werden, wenn pro Eingabe noch mindestens ein neues Element vorliegt. Sortiert man die bei der Anfrage der SweepAreas mit dem aktuellen Minimum entstehenden Ergebnisse nach Gültigkeitsintervall, so kann man diese dann direkt weiterleiten. Damit ist in diesem Fall wegen $[x_s, x_e) \leq_l [x'_s, x'_e) \wedge [y_s, y_e) \leq_l [y'_s, y'_e) \Rightarrow [x_s, x_e) \cap [y_s, y_e) \leq_l [x'_s, x'_e) \cap [y'_s, y'_e)$ bereits die Ordnung des gesamten Ausgabedatenstroms sichergestellt.

Zum anderen erlaubt eine leichte Modifikation des Algorithmus, dass er auch ohne global sortierte Ordnung der Elemente korrekt arbeitet. Dazu muss man verhindern, dass ein Element $\tilde{x}$ von einem Element x der gleichen Eingabe aus einer SweepArea gelöscht wird, das mit einem später eintreffenden Element y mit $y <_\tau x$ noch ein Joinergebnis produzieren könnte. Um das für den binären Fall sicherzustellen, wird der Reorganisationsschritt nur noch für die jeweils andere, aber nicht mehr für die zur selben Eingabe wie das eintreffende Element gehörigen SweepArea durchgeführt. Bei dieser Vorgehensweise können beim Eintreffen eines Elementes Ergebnisse entstehen, die bezüglich der Ordnung vor Ergebnissen stehen, die bereits vorher beim Eintreffen eines anderen Elements produziert wurden. Aus diesem Grund muss man die Ergebnisse an der Ausgabe des Joinoperators puffern. Generell kann ein Ergebnis mit Gültigkeitsintervall $[t_s, t_e)$ herausgegeben werden, sobald in *allen* Eingaben ein Element mit Gültigkeitsintervall J und $[t_s, t_e) <_l J$ aufgetreten ist.

Die konkrete Wahl einer der beiden Varianten hängt dabei von dem Anwendungsszenario und insbesondere der Selektivität des Joins ab. So ist bei expansiven Joins eine Pufferung der Eingaben günstiger, andernfalls die der Ausgabe. Ein wesentlicher Parameter des Rahmenwerks zur sortierbasierten Joinverarbeitung sind zudem die verwendeten SweepAreas. Diese müssen das *Einfügen*, *Reorganisieren* und *Anfragen* effizient unterstützen. Für den speziellen Fall des zeitintervallbasierten Joins sind die Operationen *Reorganisation* und *Anfrage* dabei i. W. orthogonal zueinander. Die Reorganisation erfolgt unabhängig vom konkreten Wert des Elements anhand des Gültigkeitsintervalls. Dabei werden alle Elemente entfernt, deren Gültigkeitsintervall vor dem des anfragenden Elements endet (siehe Beispiel für Intervall-Join in Abschnitt 3.1). Bei global sortierter Abarbeitung schneiden nun die Gültigkeitsintervalle aller verbliebenen Elemente in der SweepArea das des anfragenden Elements, anderenfalls i. A. zumindest ein großer Teil davon. Ist die Selektivität des Joins entsprechend gering, so sind nur wenige Joinpartner für das anfragende Element in der SweepArea. Daher fügen wir in den SweepAreas die Elemente abhängig vom Joinprädikat in einen geeigneten Index ein, um die Anfragen effektiv zu unterstützen. Zusätzlich werden sie noch in einer Prioritätswarteschlange aufsteigend nach Endzeitstempeln verwaltet, was eine effiziente Reorganisation ermöglicht. Demgemäß fügt man ein Element in beide Strukturen ein. Verwendet man beispielsweise zur Berechnung eines Equi-Joins eine Hashtabelle als Indexstruktur für die Anfragen, so erhält man eine Variante des symmetrischen Hash-Joins, die zusätzlich mit der Zeitfenstersemantik ausgestattet ist.

5.3 Temporal PMJ (TPMJ)

In Abschnitt 5.2 haben wir einen sortierbasierten Join über Datenströmen vorgestellt. Der Einsatz von Zeitfenstern gestattet dabei die Berechnung einer wohldefinierten Teilmenge der Ergebnismenge des Joins über den kompletten Datenströmen. Von praktischer Relevanz ist dabei die Größe der Ergebnismenge, d.h. wie viele Ergebnisse verliert man gegenüber der kompletten Ergebnismenge. Um diese Zahl einzuschränken, muss man die Gültigkeitsintervalle genügend groß wählen. Mit der Länge der Zeitintervalle steigt jedoch die Verweildauer der Elemente und somit die Größe der SweepAreas. Dies wird problematisch, wenn die SweepAreas nicht mehr in den für den Joinoperator reservierten Hauptspeicheranteil passen. Dann müssen Teile in den Externspeicher ausgelagert werden, weswegen die Bestimmung der Joinergebnisse für ein neues Element sehr teuer wird. Aus diesem Grund könnte man den Externspeicheranteil der SweepArea mit einer Menge vorher gesammelter Elemente anfragen. Dabei könnte man aus Effizienzgründen den externen Teil der SweepAreas und die Menge der anfragenden Elemente nach Werten sortieren und einen Merge-Join durchführen. Diese auf Sortierung nach Werten beruhende Joinberechnung legt, konsequent fortgesetzt, die Idee nahe, den in Abschnitt 4.2 vorgestellten PMJ für die Verwendung der Zeitfenstersemantik zu einem *Temporal PMJ* (TPMJ) zu adaptieren.

Um dies zu bewerkstelligen, muss man die drei folgenden Aspekte berücksichtigen: Zunächst muss man bei der Sortierung und den Operationen auf den SweepAreas die Werte der Elemente benutzen und zusätzlich deren Gültigkeitsintervalle erhalten. Bei der Berechnung der Joinergebnisse muss die Schnittbedingung darauf geprüft werden und der entsprechende Schnitt als zugehöriges Gültigkeitsintervall bestimmt werden. Von entscheidender Bedeutung ist die abschließende Forderung, dass die Ergebnisse sortiert nach Gültigkeitsintervallen ausgegeben werden müssen. Diese steht im Widerspruch zu der Tatsache, dass der PMJ in jedem Merge-Schritt die Ergebnisse lokal sortiert nach der globalen Ordnung auf den beteiligten Werten ausgibt und insgesamt die Ergebnisreihenfolge i. A. keiner Ordnung folgt.

Bei der Sortierung der Ergebnisse kann man sich die schon in Abschnitt 3.2 vorgestellte Eigenschaft des PMJ zunutze machen: In jedem Merge-Schritt werden alle Ergebnisse erzeugt, die sich aus den am Merge beteiligten Elementen berechnen lassen. Wann Ergebnisse weitergeleitet werden, entscheidet die Strategie zur Duplikateliminierung. Im Normalfall gibt man diese frühst möglich aus. Jetzt allerdings müssen die Ergebnisse zu einem Zeitpunkt herausgegeben werden, der für die nach Zeitintervallen sortierte Ausgabe günstig ist. Dazu unterteilt man die Zeitachse in disjunkte Intervalle $[s_i, e_i)$ und ordnet diese Merge-Schritten geeignet zu. In den jeweiligen Merge-Schritten werden dann genau die Ergebnisse produziert, deren Startzeitstempel in dem zugeordneten Intervall liegen. Die disjunkte Unterteilung stellt dabei sicher, dass keine Duplikate entstehen. Um ein vollständiges Ergebnis zu gewährleisten, müssen im jeweiligen Merge-Schritt auch alle entsprechenden Ergebnisse produziert werden können. Dazu müssen alle benötigten Eingabeelemente beteiligt sein. Um ein Ergebnis mit Startzeitstempel t zu produzieren, muss der Startzeitstempel aller beteiligten Eingabeelemente kleiner oder gleich t sein. Nutzt man die Sortierbedingung der Eingaben aus, so kann man nun für bestimmte Knoten im Merge-Baum geeignete Aussagen garantieren. Dazu betrachten wir einen Knoten auf der

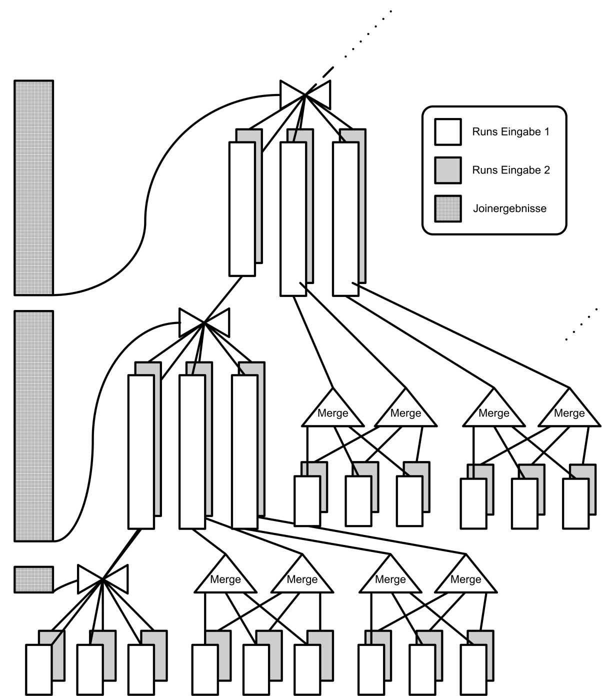

Abbildung 1: Merge-Baum der Runs beim TPMJ

linken Flanke des Merge-Baums. Unterhalb dieses Knotens liegen alle Runs, die bis zu einem bestimmten Auslagerungsschritt aus dem Hauptspeicher erzeugt wurden. Für jede Eingabe j gibt es somit ein maximales Gültigkeitsintervall $[a_j, b_j)$, das noch am Merge beteiligt ist. Folglich haben alle beteiligten Elemente ein Gültigkeitsintervall mit Startzeitstempel $\leq min\{a_j\}$. Somit sind wiederum alle Elemente beteiligt, deren Startzeitstempel $\leq min\{a_j\} - 1$ ist. Daher eignet sich $min\{a_j\}$ als offenes Ende des dem Knoten zuzuordnenden Zeitintervalls, für das die Ergebnisse herausgegeben werden. Den Startpunkt bildet jeweils das Ende des links darunter liegenden Knoten im Merge-Baum, wodurch man die disjunkte Unterteilung der Zeitachse erreicht. Der Start des ersten Intervalls wird dabei auf $-\infty$ gesetzt. Für den Fall endlicher Datenströme existiert eine echte Wurzel des Merge-Baums, an der die restlichen Ergebnisse produziert werden müssen. Das Ende ihres Intervalls wird daher auf $+\infty$ gesetzt.

Ergebnisse werden also nur noch an der linken Flanke des Merge-Baums produziert. Jedem Knoten auf der Flanke ist aufsteigend von unten nach oben ein Teilintervall der Zeitachse zugeordnet, in dem die Startzeitstempel der Ergebnisse liegen müssen, die produziert werden. Die linke Flanke des Merge-Baums wird ebenfalls von unten nach oben verarbeitet. Deswegen genügt es, jeweils die in einem Mergeschritt produzierten Ergebnisse zu sammeln und dann nach Gültigkeitsintervallen sortiert auszugeben. Damit ist bereits eine globale Sortierung der Ausgabe nach Zeitintervallen sichergestellt.

Nach dem Nachweis der Korrektheit der Ausgabe bleibt die Frage, ob auch diese mit einer Zeitfenstersemantik ausgestattete Joinvariante garantiert, dass Elemente nicht dauerhaft gespeichert werden müssen. Da ein Eingabeelement mit Gültigkeitsintervall $[t_s, t_e)$ an keinem Ergebnis beteiligt sein kann, dessen Startzeitstempel $\geq t_e$ ist, können beim Herausschreiben während des Merge-Schrittes i mit Ergebnisintervall $[s_i, e_i)$ alle Elemente mit $t_e \leq e_i$ verworfen werden. Da diese nicht mehr zu Ergebnissen beitragen können, bleibt die Vollständigkeit des Ergebnismenge erhalten. Allerdings ändert sich eine Eigenschaft, die alle Merge-Join-Verfahren von externen Sortieren übernommen haben, nämlich dass die Menge der Ausgaberuns eines Merge-Schrittes jeweils die Vereinigung der Eingaberuns ist. Um die Eingabeelemente nur logarithmisch oft vom Externspeicher lesen und darauf schreiben zu müssen, werden jeweils Runs verschmolzen, die zuvor gleich viele Merge-Schritte durchlaufen haben und somit ähnlich lang sind. Auf den höheren Merge-Ebenen des TPMJ werden die Runs durch das Verwerfen von Elementen nun aber ausgedünnt. Daher liegt es nahe, die so gekürzten Runs wieder mit Runs ähnlicher Länge zu verschmelzen. Dabei kann man die Wurzel eines Merge-Teilbaums der linken Flanke an beliebiger Stelle in der linken Flanke des restlichen Merge-Baums einsetzen. Der entstehende, in Abbildung 1 beispielhaft gezeigte, schiefe Merge-Baum gewährleistet, dass alle Elemente etwa gleich viele Ebenen des Merge-Baums durchlaufen, bis sie beim Herausschreiben verworfen werden können. Dabei bleibt zudem die Möglichkeit erhalten, wie beim normalen PMJ auf Datenströmen Merge-Schritte bei nicht voller Ausnutzung des Fan-Ins durchzuführen, solange die aktuelle Systemlast des DSMS dies gestattet.

Dieser Vorgang lässt sich noch optimieren. Man macht sich die Tatsache zunutze, dass nur noch an der linken Flanke des Merge-Baums Ergebnisse produziert werden. Deswegen muss man in anderen Teilen des Baums den Merge-Schritt des PMJ nicht mehr durchführen, da dabei entstehende Ergebnisse sämtlich verworfen würden. Dort genügt die Durchführung des herkömmlichen Merge-Schritts des externen Sortierens. Dadurch kann der Fan-In sogar gesteigert werden, da nun die einzelnen Eingaben wieder nacheinander statt ineinander verzahnt zusammengeführt werden können. Durch die auf den Zeitintervallen basierende Duplikateliminierung müssen nicht mehr immer gleich viele Runs pro Eingabe verarbeitet werden. Dies erlaubt, dynamisch zwischen zwei Merge-Schritten auf eine geänderte Speicherzuordnung zu reagieren.

Zuletzt bleibt die Frage, wann sich der Einsatz des TPMJ lohnt. Das ist sicher nicht der Fall, falls die SweepAreas beim normalen Temporal Join in den zur Verfügung stehenden Hauptspeicher passen. Anderenfalls eignet sich der TPMJ als Joinoperator, wenn zur Berechnung des Joins Externspeicher verwendet werden muss. Auch dabei können jedoch in manchen Fällen Probleme auftreten. Zunächst müssen die nun wertbasierten SweepAreas der Merge-Schritte jederzeit gemeinsam in den Hauptspeicher passen. Zusätzlich müssen die Ergebnisse gepuffert werden, da diese vor ihrer Weiterleitung nach Zeitintervallen sortiert werden müssen. Reicht dazu bei hoher Selektivität des Joins der zur Verfügung stehende Hauptspeicher nicht mehr aus, kann man wieder externes Sortieren einsetzen. Damit ermöglicht der TPMJ den Einsatz größerer Zeitfenster als die reine Hauptspeichervariante.

6 Mehrdimensionale Joinoperatoren

Die bisherige Vorstellung von Joinalgorithmen wurde bezüglich einiger Aspekte auf den binären Join spezialisiert. Im Folgenden zeigen wir die notwendigen Modifikationen zu mehrdimensionalen Joinoperatoren, obwohl die bisher vorgestellten Joinalgorithmen natürlich auch als Kaskade binärer Joins eingesetzt werden können. Bei der binären Kaskade des PMJ wird jedoch die Ausgabe des ersten, die eine Eingabe des zweiten darstellt, erneut extern sortiert, da sie nur lokal einer Sortierung folgt.

Aus den binären Joinbedingungen lässt sich ein zusammengesetztes Joinprädikat über allen beteiligten Eingaben bestimmen. Meist lässt sich auf den Eingaben eine globale Ordnung definieren, bezüglich der Elemente unterschiedlicher Eingaben nahe beieinander stehen, welche gemeinsam das so zusammengesetzte Joinprädikat erfüllen. Das gilt etwa beim Equi-Join dreier Eingaben bezüglich eines gemeinsamen Attributs. Eine solche globale Ordnung vorausgesetzt, kann man das in Abschnitt 3.1 vorgestellte Rahmenwerks zur sortierbasierten Joinverarbeitung geeignet erweitern. Dabei werden alle Eingaben gemeinsam der Ordnung folgend verarbeitet. Das Einfügen in die zur eigenen Eingabe gehörigen SweepArea kann man unverändert übernehmen. Beim Anfragen muss man nicht nur wie bisher *eine*, sondern *mehrere* andere SweepAreas berücksichtigen. Diese werden in einer geeignet zu wählenden Reihenfolge angefragt. Die Anfrage der ersten SweepArea mit einem Element liefert die Elemente zurück, die bei Kombination mit dem anfragenden Element nicht der späteren Erfüllung des mehrdimensionalen Joinprädikates widersprechen. Setzt sich dieses Prädikat aus binären Prädikaten zusammen, fragt man die SweepAreas in der Reihenfolge ab, die der Kombination aus binären Joins entspricht. Die Anfragebedingung entspricht dann jeweils dem binären Prädikat. Mit den Anfrageergebnissen der ersten SweepArea und dem anfragenden Element bildet man Tupel, die zur Anfrage der nächsten SweepArea benutzt werden. Wieder werden die Ergebnisse geliefert, die in Kombination mit den Anfragenden nicht der Erfüllung des Joinprädikates widersprechen. Die rekursive Fortsetzung des Verfahrens besucht alle SweepAreas bis auf die des ursprünglich anfragenden Elementes. Bei der letzten SweepArea kann dabei das komplette mehrdimensionale Joinprädikat ausgewertet werden, weshalb nur korrekte Ergebnisse geliefert werden. Analog zum eindimensionalen Fall darf man beim Reorganisieren keine Elemente verwerfen, die später in der global sortierten Verarbeitung der Eingabe noch zu Joinergebnissen beitragen könnten. Auf die formalen Bedingungen und den Korrektheitsbeweis [Cam02] verzichten wir aus Platzgründen.

Wir wollen das Prinzip der Reorganisation am Beispiel des Temporal-Join verdeutlichen. Da dieser auf dem sortierbasierten Rahmenwerk basiert, kann man auch eine mehrdimensionale Variante adaptieren. Wir betrachten zunächst den Fall, in der die Eingaben so gepuffert werden, dass sie global der Ordnung auf ihren Gültigkeitsintervallen folgend verarbeitet werden. Jede SweepArea kann analog zum binären Fall reorganisiert werden, da nach der Verarbeitung eines Elementes mit Gültigkeitsintervall $[t_s, t_e)$ keine Ergebnisse mit Startzeitstempel kleiner oder gleich $t_s - 1$ mehr produziert werden können. Die Gültigkeitsintervalle der nach der Reorganisation verbliebenen Elemente schneiden sich, weswegen man die rekursive Abfrage der SweepAreas allein wertbasiert durchführen kann. Im Fall der direkten Verarbeitung der Elemente ist der Reorganisationsschritt etwas

komplizierter. Hier darf man nicht aufgrund des Startzeitstempels des eingetroffenen Elements, sondern lediglich bezüglich des Minimums der zuletzt eingetroffenen Elemente aller Eingaben reorganisieren. Das liegt daran, dass dieses Minimum den aktuell verarbeiteten Anteil der Eingaben bezüglich der global sortierten Verarbeitung abgrenzt.

Die Vor- und Nachteile des mehrdimensionalen Temporal-Joins sieht man beim Vergleich mit zwei kaskadierten binären Joins. Diese enthalten vier SweepAreas: drei enthalten die Elemente der Eingaben und eine die Ergebnisse des ersten binären Joins. Diese Ergebnisse sind somit gewissermaßen im Strom zwischen den binären Joins und in der vierten SweepArea materialisiert. In der ternären Variante gibt es nur drei SweepAreas. Erfüllen zwei Elemente einer Eingabe aber mit den gleichen Kombinationen der anderen das Joinprädikat, so müssen diese bei der rekursiven Berechnung der Joinergebnisse jeweils neu berechnet werden. Im binär kaskadierten Fall werden die Elemente mit materialisierten Zwischenergebnissen auf Erfüllung des Joinprädikats getestet. Somit konsumiert die multidimensionale Variante weniger Speicher, während die Kaskadierung binärer Joins einen geringeren Berechnungsaufwand aufweist.

Ein fundamentales Problem der Anfrageoptimierung in klassischen DBMS stellt die richtige Platzierung und Anordnung binärer Joins dar. Dazu nutzt man Metadaten der beteiligten Relationen aus und optimiert statisch für eine einmal auszuführende Anfrage. In einem DSMS liegen jedoch meist Anfragen vor, die kontinuierlich auf den Datenströmen laufen. Da deren Charakteristika sich zur Laufzeit grundlegend ändern können, ändern sich möglicherweise auch die Selektivitäten der Anfrage. Die Umgruppierung binärer Joins in DSMS ist zwar möglich [ZRH04], gestaltet sich aber als schwierig. Die vorgestellte mehrdimensionale Variante erlaubt eine flexible und unproblematische Umstellung zur Laufzeit. Dazu muss nur die Anfragereihenfolge der SweepAreas geändert werden, was prinzipiell für jedes Element möglich ist. Dies erlaubt z. B. eine adaptive Anpassung aufgrund neuer Ergebnisse der Berechnungen eines Optimierers basierend auf aktualisierten Metadaten. Da es zudem möglich ist, innerhalb des mehrdimensionalen Joins Zwischenergebnisse zu materialisieren und diese bei einer Änderung der Anfragereihenfolge einfach wieder zu verwerfen und stattdessen wieder auf die unterliegenden SweepAreas zuzugreifen, kann man sogar den Nachteil gegenüber den binären Joins bei Bedarf kompensieren.

Den offensichtlichsten Vorteil bietet die Verwendung der mehrdimensionalen Variante des TPMJ. Kaskadiert man mehrere binäre Instanzen davon, so werden die nach Gültigkeitsintervallen sortierten Eingaben nach Werten umsortiert, zu Ergebnissen verschmolzen, wieder nach Gültigkeitsintervallen sortiert ausgegeben und für den nächsten binären Join wieder nach Werten sortiert. Liegt eine geeignete globale Ordnung vor, ist es günstiger, die Elemente nur einmal bezüglich dieser Ordnung zu sortieren und die Ergebnisse des gesamten mehrdimensionalen Joins dann wieder nach Gültigkeitsintervallen sortiert auszugeben.

7 Experimente

Die vorgestellten Joinoperatoren sind im Paket PIPES [KS04] unserer Java-Bibliothek XXL implementiert. Unsere Experimente wurden auf einem PC mit Pentium IV 2,3 GHz

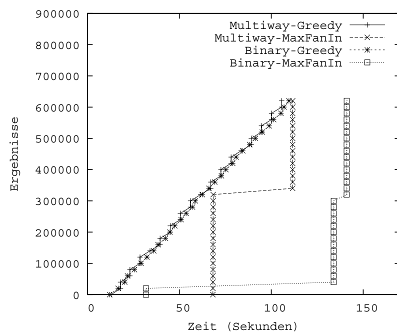

Abbildung 2: Strategievergleich

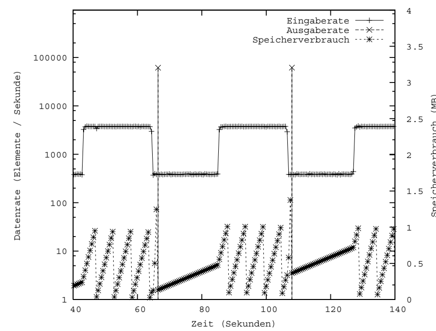

Abbildung 3: Adaptivität

und 1 GB Hauptspeicher unter Windows XP durchgeführt.

In beiden Experimenten berechnen wir mit dem TPMJ jeweils einen Intervall-Join; die Werte der Elemente der Eingabeströme sind also Wertintervalle (siehe Abschnitt 3.1). Im binären und mehrdimensionalen Fall erfüllen zwei oder mehr Eingabeintervalle das Joinprädikat, wenn sie einen gemeinsamen Schnitt haben. Wir verwenden vier Eingabeströme mit jeweils 100.000 Elementen. Als Anfänge dieser Wertintervalle wird für jede Eingabe eine unabhängige zufällige Permutation der Zahlen 1 bis 100.000 gewählt. Die Wertintervalllänge ist konstant gewählt. Es werden gleitende Fenster der Länge 10.000 Zeiteinheiten verwendet und die Wertintervalle aller Eingaben aufsteigend von 1 bis 100.000 mit Startzeitstempeln versehen. Durch die Zeitfenstersemantik bilden Elemente ein Joinergebnis, wenn sich sowohl ihre Wert- als auch Gültigkeitsintervalle schneiden. Den verwendeten Instanzen des TPMJ ist insgesamt 1 MB Speicher zur Generierung der initialen Runs zugeteilt.
Zur Auswahl der Runs in den Merge-Schritten sind zwei Strategien implementiert. Die konservative *MaxFanIn*-Strategie führt Merge-Schritte nur bei voller Ausnutzung eines vorgegebenen Fan-Ins aus, während die *Greedy*-Strategie immer Merge-Schritte ausführt, wenn der Merge-Thread zum Zuge kommt.

Im ersten Experiment vergleichen wir den quaternären Join auf unseren vier Quellen bei einer Datenrate von je einem Element pro Millisekunde mit Wertintervalllänge 75 mit einem linkstiefen Baum aus binären Joins. Für diese werden jeweils beide Strategien untersucht. Abbildung 2 zeigt die Zahl der produzierten Ergebnisse in Abhängigkeit von der verstrichenen Zeit. Man sieht, dass alle Varianten erst Ergebnisse produzieren können, wenn nach 10 s die ersten Elemente aus den Zeitfenstern entfernt werden können. Beide Varianten können mit der *Greedy*-Strategie zu jeder Zeit i. W. mit der optimalen zu diesem Zeitpunkt berechenbaren Ergebniszahl mithalten, wobei der mehrdimensionale Join geringfügig schneller Ergebnisse liefert. Bei Verwendung von *MaxFanIn* warten die Joins zu lange mit der Durchführung von Merge-Schritten und führen ihre Berechnungen der 631.130 Ergebnisse erst zu Ende, wenn alle Quellen versiegt sind. Neben den in Abbildung 2 gezeigten Werten haben wir zusätzlich die Zahl der Externspeicherseitenzugriffe protokolliert. Dabei benötigte die konservative *MaxFanIn*-Strategie im binären Fall 227.872 und im mehrdimensionalen nur 42.004 Zugriffe. Die *Greedy*-Variante brauchte mit 492.240 im

binären Fall und 107.214 jeweils deutlich mehr Zugriffe, wobei jedoch wieder die mehrdimensionale Variante wegen ausbleibender Materialisierung von Zwischenergebnissen mit ca. fünfmal geringerem I/O-Aufwand deutlich ressourcenschonender arbeitet.

Das zweite Experiment demonstriert eine Adaption auf sich ändernde Eingaberaten. Dazu wechseln die Quellen jetzt nach je 20 Sekunden zwischen einer Verzögerung von 1 ms und 10 ms zwischen ihren Elementen. Im mehrdimensionalen TPMJ wird in Zeiten hoher Last die *MaxFanIn*-, in Zeiten geringer Last die *Greedy*-Strategie aktiviert. Abbildung 3 zeigt den Rechteck-Verlauf der Eingaberaten und die Produktion von Ergebnisse nach Wechsel der Strategie. Der ebenfalls dargestellte Speicherverbrauch steigt jeweils bis zum Erreichen der 1MB-Grenze an, an der die Auslagerung erfolgt. Bei der Berechnung der Ergebnisse sieht man deutlich, dass jetzt auch die SweepAreas und der Externspeicherpuffer merklich Speicher benötigen.

8 Fazit und Ausblick

In dieser Arbeit haben wir die in DBMS bewährte sortierbasierte Joinverarbeitung auf Datenströme übertragen. Dazu haben wir zunächst aufgezeigt, wie der nichtblockierende PMJ an die Anforderungen aktiver Datenverarbeitung angepasst werden kann. Da die Berechnung des vollständigen Joins unendlicher Datenströme mit begrenzten Ressourcen i. A. nicht möglich ist, haben wir dargelegt, wie unsere Zeitfenstersemantik die Berechnung einer klar definierten Teilmenge des Ergebnisses ermöglicht, die den Join im zeitlichen Zusammenhang relevanter Daten darstellt. Zur Umsetzung der Semantik haben wir zunächst zwei Varianten eines hauptspeicherbasierten Joinoperators vorgestellt, welche die zeitliche Sortierung der Daten ausnutzen. Für Fälle, in denen diese ungeeignet sind, haben wir zusätzlich mit dem TPMJ eine Erweiterung des PMJ vorgestellt, die den zeitfensterbasierten Join mit Hilfe wertbasierter Sortierung unter Einsatz des Externspeichers berechnet. Wir haben verschiedene Strategien zur Parametrisierung der Algorithmen mit Hinblick auf die Auslastung eines DSMS untersucht und die Überlegenheit der *Greedy*-Strategie bei hinreichend verfügbaren Ressourcen demonstriert. Durch den Einsatz eines flexiblen Rahmenwerks zur sortierbasierten Joinberechnung sind die vorgestellten Weiterentwicklungen des PMJ zur Verarbeitung einer Vielzahl von Joinprädikaten wie Equi-, Similarity- oder Spatial-Join geeignet. Wir haben die vorgestellten Algorithmen zu mehrdimensionalen Joinoperatoren verallgemeinert und gezeigt, dass diese in vielen Fällen der Kaskade ihrer binären Varianten überlegen sind.

Bei der Weiterentwicklung von PIPES wollen wir die Adaptivität unserer Algorithmen im Zusammenspiel mit der komplexen Laufzeitumgebung eines DSMS weiter verbessern.

Danksagung

Diese Arbeit wurde von der Deutschen Forschungsgemeinschaft (DFG) unter Projektnummer SE 553/4-1 gefördert. Zusätzlich danken wir allen, die an der Entwicklung von XXL und insbesondere PIPES beteiligt sind.

Literatur

[ACG+04] Arvind Arasu, Mitch Cherniack, Eduardo F. Galvez, David Maier, Anurag Maskey, Esther Ryvkina, Michael Stonebraker und Richard Tibbetts. Linear Road: A Stream Data Management Benchmark. In *VLDB*, Seiten 480–491, 2004.

[Cam02] Michael Cammert. Frühe Ergebnisse bei Verbundoperationen. Diplomarbeit, http://www.mathematik.uni-marburg.de/˜cammert/da.pdf, Philipps-Universität Marburg, 2002.

[DS01] Jens-Peter Dittrich und Bernhard Seeger. GESS: a scalable similarity-join algorithm for mining large data sets in high dimensional spaces. In *KDD*, Seiten 47–56, 2001.

[DSTW02] Jens-Peter Dittrich, Bernhard Seeger, David Scot Taylor und Peter Widmayer. Progressive Merge Join: A Generic and Non-Blocking Sort-Based Join Algorithm. In *VLDB*, Seiten 299–310, 2002.

[DSTW03] Jens-Peter Dittrich, Bernhard Seeger, David S. Taylor und Peter Widmayer. On producing join results early. In *PODS*, Seiten 134–142, 2003.

[GO03a] Lukasz Golab und M. Tamer Özsu. Issues in data stream management. In *SIGMOD*, Seiten 5–14, 2003.

[GO03b] Lukasz Golab und M. Tamer Özsu. Processing Sliding Window Multi-Joins in Continuous Queries over Data Streams. In *VLDB*, Seiten 500–511, 2003.

[KNV03] Jaewoo Kang, Jeffrey F. Naughton und Stratis Viglas. Evaluating Window Joins over Unbounded Streams. In *ICDE*, Seiten 341–352, 2003.

[KS04] Jürgen Krämer und Bernhard Seeger. PIPES - A Public Infrastructure for Processing and Exploring Streams. In *SIGMOD*, 2004.

[KS05] Jürgen Krämer und Bernhard Seeger. A Temporal Foundation for Continuous Queries over Data Streams. In *COMAD*, 2005.

[LG98] Per-Åke Larson und Goetz Graefe. Memory Management During Run Generation in External Sorting. In *SIGMOD*, Seiten 472–483, 1998.

[MLA04] Mohamed F. Mokbel, Ming Lu und Walid G. Aref. Hash-Merge Join: A Non-blocking Join Algorithm for Producing Fast and Early Join Results. In *ICDE*, Seiten 251–263, 2004.

[SW04] Utkarsh Srivastava und Jennifer Widom. Flexible Time Management in Data Stream Systems. In *PODS*, Seiten 263–274, 2004.

[UF00] Tolga Urhan und Michael J. Franklin. XJoin: A Reactively-Scheduled Pipelined Join Operator. In *IEEE Data Eng. Bull.*, Jgg. 23, Seiten 27–33, 2000.

[VNB03] Stratis D. Viglas, Jeffrey F. Naughton und Josef Burger. Maximizing the Output Rate of Multi-Way Join Queries over Streaming Information Sources. In *VLDB*, Seiten 285–296, 2003.

[WA93] Annita N. Wilschut und Peter M. G. Apers. Dataflow Query Execution in a Parallel Main-memory Environment. In *Distributed and Parallel Databases*, Seiten 103–128, 1993.

[ZRH04] Yali Zhu, Elke A. Rundensteiner und George T. Heineman. Dynamic Plan Migration for Continuous Queries Over Data Streams. In *SIGMOD Conference*, Seiten 431–442, 2004.