Philipps-Universität Marburg
Fachbereich Mathematik und Informatik

Frühe Ergebnisse bei Verbundoperationen

Diplomarbeit von
Michael Cammert

Betreuer: Prof. Dr. Bernhard Seeger

Marburg/Lahn, Dezember 2002

Zusammenfassung

In der Geschichte der Datenbanksysteme wurden zur Berechnung von Verbundoperationen (Joins) verschiedene Techniken entwickelt, darunter der einfache Nested-Loops-Join, der Sort-Merge-Join und der Hash-Join. U. a. mit zunehmender Bedeutung von Data-Warehouse-Systemen kam der Wunsch nach einer Anfragebearbeitung auf, die auch bei längeren Berechnungen früh Ergebnisse und Schätzungen über das Gesamtergebnis liefert. Da Joins bei Anfragen sehr häufig benötigt werden, mußten die Joinverfahren entsprechend angepaßt werden, was zunächst für den Nested-Loops- und den Hash-Join geleistet wurde [HH99] und diverse Weiterentwicklungen nach sich zog. Allerdings sind Algorithmen auf Basis von Nested-Loops nur bei komplett im Hauptspeicher ablaufenden Joins sinnvoll, während hashbasierte Joinverfahren nicht alle Arten von Joinprädikaten verarbeiten können.

Mit dem Progressive-Merge-Join (PMJ) [DSTW02b] wurde ein sortierbasierter Algorithmus entwickelt, der frühe Ergebnisse und Schätzer ermöglicht und diese somit dem erheblich größeren Spektrum der sortierbasierten Joinarten erschließt. Damit eröffnete sich ein neues Feld für Erweiterungen und Optimierungen, das in dieser Arbeit untersucht wird.

Beim Sort-Merge-Join werden zunächst die Eingaberelationen nacheinander extern sortiert, wozu jeweils Abschnitte in Hauptspeichergröße eingelesen, sortiert und wieder hinausgeschrieben werden. Der PMJ teilt dagegen den Hauptspeicher auf und sortiert dort je einen Abschnitt der Relationen, um dann vor dem Hinausschreiben deren Join zu berechnen. Dies liefert frühe Ergebnisse und ermöglicht eine Schätzung der Selektivität des Joins.

Als Weiterentwicklung des PMJ wird in dieser Arbeit der mehrdimensionale Progressive-Merge-Join (MPMJ) vorgestellt, der auch asymmetrische Joins von mehr als zwei Eingaberelationen berechnen kann. Dazu werden jeweils Abschnitte aller Relationen nebeneinander im Hauptspeicher sortiert und rekursiv deren Join berechnet.

Die Verteilung des Hauptspeichers unter den Relationen hat dabei wesentlichen Einfluß auf die Erzeugung früher Ergebnisse. Diese kann z.B. bezüglich Produktionsrate oder Gesamtzahl optimiert werden, welche noch erhöht werden kann, indem die sortierten Folgen nicht mehr gemeinsam, sondern nacheinander ausgetauscht werden, wobei zwischendurch jeweils der Join der im Hauptspeicher befindlichen Abschnitte berechnet wird.

Messungen bestätigen die Fähigkeit des MPMJ zur Produktion früher Ergebnisse und die Erhöhung ihrer Anzahl durch Einsatz der Optimierungen bei moderat erhöhter Gesamtlaufzeit.

Inhaltsverzeichnis

Abbildungsverzeichnis

Algorithmenverzeichnis

Tabellenverzeichnis

Kapitel 1

Grundlagen

1.1 Übersicht

Diese Arbeit gliedert sich in sechs Kapitel, beginnend mit einer Erläuterung grundlegender Begriffe und Verfahren. Im zweiten Kapitel wird der Algorithmus Progressive-Merge-Join (PMJ) [DSTW02b] besprochen, dessen Weiterentwicklung zum mehrdimenionalen Progressive-Merge-Join (MPMJ) im dritten Kapitel vorgestellt wird. Optimierungen beider Algorithmen werden im vierten Kapitel behandelt, im fünften werden dazu die Ergebnisse einiger Experimente vorgestellt. Das letzte Kapitel beschließt die Arbeit mit Fazit und Ausblick.

Übersicht

In diesem Kapitel werden die nötigen Grundlagen für diese Arbeit behandelt. Dazu werden in 1.2 einige wichtige Grundbegriffe, auch bezüglich des Titels dieser Arbeit, „Frühe Ergebnisse bei Verbundoperationen“, eingeführt. In 1.3 wird dann das Interesse am Thema motiviert und in 1.4 eine Voraussetzung für den Einsatz sogenannter Schätzer formuliert. Abschließend werden in 1.5 wichtige Arten von Verbundoperationen erläutert und ein Überblick über Verfahren zu deren Berechnung gegeben.

1.2 Grundbegriffe

Eine Verbundoperation ist, wie der Name schon sagt, eine Operation, die einen Verbund herstellt, also verbindet. Die Frage ist nun, was und nach welchen Kriterien verbunden wird.

1.2.1 Die relationale Algebra

Relationale Datenbanksysteme basieren auf dem relationalen Datenmodell. Aufbauend auf elementaren Wertebereichen wird dabei eine Relation R definiert durch eine Menge von Attributen $\{A_1, \ldots, A_k\}$, die das Relationenschema S_R von R genannt wird. Jedes dieser Attribute A_j hat einen eindeutigen Wertebereich D_j. Eine Teilmenge des kartesischen Produktes der Wertebereiche der Attribute von R bildet eine Instanz I_R der Relation.

$$I_R \subseteq D_1 \times \cdots \times D_k, \quad k \geq 1$$

Dem allgemeinen Sprachgebrauch folgend wird der Begriff Relation in dieser Arbeit auch für Instanzen einer Relation verwendet, da diese der Gegenstand der konkreten Verarbeitung sind. Mit $t \in R$ ist ebenso zu verfahren, dies bedeute $t \in I_R$. Die Elemente einer Relationeninstanz werden *Tupel* genannt.

Eine Menge N von Relationen bildet nun den Anker der relationalen Algebra, auf dem deren Operationen $\alpha_i : N^l \to N$ definiert sind. Das Schema der Ergebnisrelation wird dabei durch die beteiligten Relationen und die Operation bestimmt. Die sogenannten *fünf plus eins Grundoperationen* der relationalen Algebra sind:

Vereinigung $R_1 \cup R_2$ Die Vereinigung ist nur auf Instanzen I_{R_1} und I_{R_2} zweier Relationen R_1 und R_2 mit identischem Relationenschema S_R definiert. Ihr Ergebnis $R_1 \cup R_2$ ist eine Relation mit Relationenschema S_R und als Instanz der Vereinigung von I_{R_1} und I_{R_2}.

$$S_{R_1} = S_{R_2} \qquad \Rightarrow$$

$$S_{R_1 \cup R_2} = S_{R_1}, \qquad I_{R_1 \cup R_2} = I_{R_1} \cup I_{R_2}$$

Differenz $R_1 - R_2$ Unter den gleichen Voraussetzungen wie bei der Vereinigung wird hier eine Relation $R_1 - R_2$ mit Schema S_R und als Instanz der Mengendifferenz von $I_{R_1} \backslash I_{R_2}$ erzeugt.

$$S_{R_1} = S_{R_2} \qquad \Rightarrow$$

$$S_{R_1 - R_2} = S_{R_1}, \qquad I_{R_1 - R_2} = I_{R_1} \backslash I_{R_2}$$

Selektion $\sigma_F(R)$ Unter Erhaltung des Schemas von R werden diejenigen Tupel aus I_R in die Ergebnisinstanz ausgewählt, die der Bedingung F genügen.

$$S_{\sigma_F(R)} = S_R, \qquad I_{\sigma_F(R)} = \{t \in I_R \mid F(t)\}$$

Projektion $\pi(R)$ Die Projektion einer Relation R auf eine Teilmenge U ihres Relationenschemas entfernt die Attribute aus $S \backslash U$ des Schemas und die zugehörigen Werte der Attribute der entsprechenden Wertebereiche.

$$\{i_1, \ldots, i_m\} \subseteq S_R \qquad \Rightarrow$$

$$S_{\pi_{i_1,\ldots,i_m}(R)} = \{i_1, \ldots, i_m\},$$

$$I_{\pi_{i_1,\ldots,i_m}(R)} =$$

$$\{(c_1, \ldots, c_m) \mid \forall i \in \{\ 1, \ldots, m\} \ \exists (a_1, \ldots, a_n) \in I_R : \ c_j = a_{i_j}\}$$

Kartesisches Produkt $R_1 \times R_2$ Für Relationen disjunkter Schemata wird das kartesische Produkt [1] gebildet. Wegen der Assoziativität des kartesischen Produktes kann man das Resultat wieder als Relation bilden. Disjunkte Schemata können durch Umbenennung (siehe unten) immer erreicht werden, weshalb diese formale Bedingung im Folgenden etwas aufgeweicht wird.

$$S_{R_1} \cap S_{R_2} = \emptyset \qquad \Rightarrow$$

$$S_{R_1 \times R_2} = S_{R_1} \dot{\cup} S_{R_2},$$

$$I_{R_1 \times R_2} = \{(c_1, \ldots, c_{n+m}) \mid (c_1, \ldots, c_n) \in I_{R_1}, (c_{n+1}, \ldots, c_{n+m}) \in I_{R_2}\}$$

Umbenennung $\varrho_{R_2}(R_1)$ **und** $\varrho_{A \leftarrow B}(R_1)$ Bei dieser zusätzlichen Operation wird nur das Schema einer Relation verändert. Dabei können die Namen der Relation und die ihrer Attribute verändert werden. Dies kann nötig werden, um die Bedingungen der anderen Operationen an die Relationenschemata zu erfüllen.

Mit diesen Grundoperationen lassen sich nun weitere Operationen wie Durchschnitt oder Quotient ableiten.

1.2.2 Verbundoperation

Auch bei der Verbundoperation handelt es sich um eine Kombination aus den Grundoperationen der relationalen Algebra:

$$R_1 \bowtie_{M,F} R_2 := \pi_M(\sigma_F(R_1 \times R_2))$$

Die Verbundoperation $R_1 \bowtie_{M,F} R_2$ wählt also aus dem Kartesischen Produkt $R_1 \times R_2$ diejenigen Tupel aus, die dem Prädikat F genügen, und projiziert diese dann auf die Teilmenge M des Schemas von $R_1 \times R_2$. Charakteristisch für die Verbundoperation ist dabei die Kombination aus kartesischem

[1] auch Kreuzprodukt genannt

Produkt und Selektion.

$$R_1 \bowtie_F R_2 := \sigma_F(R_1 \times R_2) = \{(r_1, r_2) \mid r_1 \in R_1, r_2 \in R_2, F(r_1, r_2)\}$$

Statt einer Projektion könnte auf die selektierten Tupel nach solch einem Join auch eine andere Funktion angewendet werden.

Im der englischen Sprache nennt man die Verbundoperation *Join*[2]. Da diese Bezeichnung auch in den deutschen Sprachgebrauch eingeflossen ist, wird sie in dieser Arbeit ebenfalls verwendet.

1.2.3 Mehrdimensionale Verbundoperation

Für mehr als zwei Relationen $R_1, \ldots, R_r$ kann man aufgrund der Assoziativität der Kreuzproduktbildung analog einen Join definieren:

$$\bowtie_{M,F} (R_1, \ldots, R_r) := \pi_M(\sigma_F(R_1 \times \cdots \times R_r))$$

Die hier vorgenommenen Definitionen wurden in Anlehnung an [See97] getroffen. Sie sind in ähnlicher Form auch z. B. in [KE96] zu finden.

1.2.4 Selektivität

Die *Selektivität* einer Selektion entspricht dem Anteil der Tupel, die das Selektionsprädikat erfüllen, und ist somit ein Maß dafür, wie stark die Selektion filtert:

$$Sel(\sigma_F(R)) = \frac{|\sigma_F(R)|}{|R|}$$

Für Joins definiert man die Selektivität entsprechend als die Selektivität ihres Joinprädikates, also als das Verhältnis zwischen Kardinalität des Joinergebnisses und des kartesischen Produktes der beteiligten Relationen:

$$Sel(R_1 \bowtie_F R_2) := \frac{|R_1 \bowtie_F R_2|}{|R_1 \times R_2|}$$

Dies entspricht somit der Wahrscheinlichkeit für ein zufällig aus $R_1 \times R_2$ gewähltes Tupel, dem Joinprädikat F zu genügen, also Teil des Joinergebnisses zu sein.

[2] engl. join = verbinden, vereinigen

1.2.5 Physische Operatoren

Die vorangegangenen Definitionen stellen eine mathematische Spezifikation der Operationen der relationalen Algebra dar. Will man diese Operationen konkret ausführen lassen, so bedarf es einer geeigneten Berechnungsvorschrift, also eines Algorithmus, der die Spezifikation erfüllt. Natürlich kann man solche Algorithmen angeben, die den Join allgemein berechnen. Als einfachstes Beispiel sei die Hintereinanderausführung der Algorithmen für die am Join beteiligten elementaren Operationen Kreuzproduktbildung und Selektion genannt.

Algorithmen haben nun vielerlei Charakteristika. Die bekanntesten Vertreter sind wohl die Zeit- und die Speicherkomplexität, also einfach gesagt der Zeit- und Speicherbedarf des Algorithmus in Abhängigkeit von seinen Parametern. Im Falle des (binären) Joins sind das R_1, R_2, M und F.

Im Umfeld eines Datenbanksystems kommen implizit noch weitere Parameter hinzu. Die am Join beteiligten Relationen können dem Algorithmus in verschiedensten Formen zur Verfügung stehen. So können sie auf verschiedenen Ebenen der Speicherhierarchie abgelegt und mit unterschiedlichen Zugriffsstrukturen wie z. B. Indizes angereichert sein. Im Datenbanksystem können auch noch Metainformationen abgelegt sein, also Daten über die Eigenschaften der Daten, zu denen auch die am Join beteiligten Relationen zählen können.

Sind nun bestimmte Bedingungen an die Parameter und das Umfeld erfüllt, so können andere Algorithmen zur Berechnung der Operationen benutzt werden, die unter ausgewählten Gesichtspunkten erheblich günstigere Eigenschaften als die allgemeinen Verfahren haben.

Die unterschiedlichen Implementierungen der verschiedenen Algorithmen zur Berechnung der Operationen der relationalen Algebra nennt man *physische Operatoren*.

1.2.6 Frühe Ergebnisse

Beschäftigt man sich mit der Zeitkomplexität eines Algorithmus, so betrachtet man üblicherweise die Gesamtlaufzeit, bis die gesamte Verarbeitung abgeschlossen ist. In vielen Fällen, insbesondere wenn nur ein Wert zu berechnen ist, ist dies auch der zeitlicher Aspekt, der optimiert werden soll. Ein Join berechnet aber nicht nur ein Ergebnis, sondern üblicherweise eine Vielzahl von Tupeln. Nun kann man betrachten, wann diese während der Gesamtlaufzeit produziert werden. *Frühe Ergebnisse* nennt man nun solche, die relativ am Anfang der Gesamtlaufzeit produziert werden.

1.3 Motivation

Warum ist nun die Erzeugung früher Ergebnisse bei Verbundoperationen von Interesse? Die Berechnung eines Joins erfolgt nicht als Selbstzweck, sondern ist normalerweise Teil eines Datenverarbeitungsprozesses. Die produzierten Ergebnisse werden dabei verwendet; sie können Parameter weiterer Berechnungen sein oder menschlichen Benutzern mitgeteilt werden.

Menschen erwarten von Maschinen und Computern nun, daß sie ihnen die Arbeit erleichtern, während die Benutzer selber steuernd eingreifen. Unerwünscht sind dabei Wartezeiten, während derer der Benutzer nichts anderes tun kann, als auf das Eintreffen von Resultaten zu warten. Treffen diese dann schließlich auch noch alle binnen kurzer Zeit ein, so dauert es dann noch einmal, bis der Benutzer sie verarbeiten (im einfachsten Fall lesen) kann. Besser ist ein kontinuierliches Eintreffen von Ergebnissen während der gesamten Berechnung. Dies gilt auch, wenn die Ergebnisse einem anderen Konsumenten zugeleitet werden, der nur eine begrenzte Zahl von Ergebnissen pro Zeiteinheit verarbeiten kann, etwa einem Drucker oder einer Netzwerkverbindung.

Sollen die Ergebnisse eines Joins in einem umfassenderen Prozeß weiterverwendet werden, so können die Operationen, welche die Joinergebnisse als Parameter benötigen, erst beginnen, sobald das erste Ergebnistupel vorliegt, und sind auch in der Folge immer wieder auf die Produktion ihrer Eingaben angewiesen. Algorithmen, die zunächst ihre kompletten Eingaben benötigen, bevor sie selbst Ergebnisse liefern, bezeichnet man als *blockierend*, da sie die Arbeit der auf die Ergebnisse angewiesenen Verarbeitung blockieren. Eine solche Blockade betrifft dann alle nachfolgenden Prozesse, die direkt oder indirekt von den Ergebnissen abhängen. Am Ende dieser Kette möchte man aber wie zuvor erläutert oft gerne frühe Ergebnisse erhalten, wozu also die Verwendung blockierender Algorithmen auszubleiben hat.

Bisher wurde der Fall betrachtet, daß wirklich alle Ergebnisse benötigt werden. Dies ist aber gar nicht immer der Fall. So kann ein Benutzer anhand erster Ergebnisse feststellen, daß weitere für seine Zwecke gar nicht mehr gebraucht werden. Oft genügt als Ergebnis eines Berechnungsprozesses auch eine mit annehmbarer Wahrscheinlichkeit richtige Schätzung des tatsächlichen Ergebnisses. Eine solche kann nun unter bestimmten Voraussetzungen (siehe 1.4) schon auf Basis früher Ergebnisse angegeben werden. Der Benutzer sollte die Verarbeitung daher zur Laufzeit abbrechen können, sobald ihm die Ergebnisse genügen, oder aber auch bis zum Abschluß laufen lassen können.

Relationale Datenbanksysteme sind derzeit marktdominierend. Von erheblicher Bedeutung sind auch sogenannte Data Warehouses, das sind spezielle Datenbanksysteme, die u. a. zur Entscheidungsfindung in Unternehmen

eingesetzt werden. Gerade darin werden umfangreiche Joins berechnet, bei denen frühe Ergebnisse oder Schätzer oft als Resultat bereits ausreichen. Frühe Joinergebnisse sind daher nicht nur von rein wissenschaftlichem, sondern auch von praktischem Interesse.

Wünschenswert ist also die Entwicklung nicht blockierender Algorithmen zur Joinverarbeitung, die ihre Ergebnisse möglichst früh produzieren, aber auch alle Ergebnisse in annehmbarer Zeit liefern können.

1.4 Voraussetzung für Schätzer

Wenn man sich nun einen Schätzer wünscht, so muß man dafür auch gewisse Einschränkungen hinnehmen. Betrachtet man einen Join mit genau einem Ergebnistupel $t = (t_1, t_2)$, welches das Joinprädikat F erfüllt, so kann man t erst erzeugen, wenn man $t_1 \in R_1$ und $t_2 \in R_2$ bereits aus den Eingaberelationen gelesen hat. Im schlechtesten Fall muß man also beide Relationen komplett betrachten, wonach man kaum noch von einem frühen Ergebnis sprechen kann. Leider kann dieser Effekt auch bei mehr Ergebnissen noch eintreten; so können 100 Ergebnistupel erst von den jeweils 10 letzten Eingabetupeln der beiden Relationen gebildet werden. Ungeachtet des Algorithmus kann also die Reihenfolge, in der die Eingaben gelesen werden, die Ergebnisproduktion verzögern.

Dieses Problem ist sogar noch unangenehmer, wenn man aufgrund der frühen Ergebnisse Schätzungen bezüglich des Joinergebnisses machen will, da das Ausbleiben früher Ergebnisse zu der Annahme führt, der Join habe keine oder nur wenige Ergebnisse.

Will man aus einer Teilmenge Rückschlüsse auf die Gesamtmenge ziehen, so muß die Teilmenge repräsentativ für die gesamte sein, also eine Stichprobe ihrer darstellen. Liest man also $T_i \subset R_i$, so muß für die Wahrscheinlichkeit P, daß ein Tupel $t_i \in R_i$ in T_i liegt, gelten:

$$P(t_i \in T_i) = \frac{|T_i|}{|R_i|} \tag{1.1}$$

Liegen die Eingaberelationen zu Beginn des Joins komplett vor und kann wahlfrei auf sie zugegriffen werden, so kann man diese Bedingung erfüllen, indem man ohne Zurücklegen zufällig Tupel zieht. Da die Relationen üblicherweise seitenweise vom Externspeicher gelesen werden, ist dies leider nicht besonders performant. Ohne wahlfreien Zugriff kann man die Eingaben komplett lesen und so ablegen, daß sie danach der Bedingung genügend gelesen werden können. Dieses Verfahren ist jedoch blockierend.

Techniken zur Lösung dieser Problematiken werden z. B. in [HAR99] und [Olk93] vorgestellt und hier nicht weiter diskutiert. Im Folgenden wird davon ausgegangen, daß die Eingaberelationen in geeigneter zufälliger Reihenfolge gelesen werden können, nach Lesen von Teilmengen T_i also Bedingung 1.1 erfüllt ist.

1.5 Einordnung von Joins

Natürlich gibt es für Joins im Anwendungsbereich von Datenbanksystemen typische Anwendungen, insbesondere kommen bestimmte Arten von Joinprädikaten häufig vor. Diese und darauf angepaßte physische Operatoren sollen hier vorgestellt werden.

1.5.1 Arten von Joins

Seien R_1 und R_2 Relationen mit Relationenschemata $S_{R_1} = \{A_1, \ldots, A_{r_1}\}$ bzw. $S_{R_2} = \{B_1, \ldots, B_{r_2}\}$.

1.5.1.1 Equi-Join

Beim Equi-Join wird im einfachsten Fall die Gleichheit auf je einem Attribut der Relationen gefordert:

$$R_1 \bowtie_{i=j} R_2 := \sigma_{A_i=B_j}(R_1 \times R_2)$$

Allgemeiner definiert man den Equi-Join auf mehreren Attributen durch:

$$R_1 \bowtie_{i_1=j_1,\ldots,i_n=j_n} R_2 := \sigma_{A_{i_1}=B_{j_1},\ldots,A_{i_n}=B_{j_n}}(R_1 \times R_2)$$

Stimmen die Schemata von R_1 und R_2 teilweise überein (o. B. d. A. $A_i = B_i,\ i = 1, \ldots, k$), so bietet es sich an, auf diesen Attributen Gleichheit zu verlangen und dann die Menge der gemeinsamen Spalten einmal herauszuprojizieren. Diese Gesamtoperation wird als *Natürlicher Verbund* bezeichnet und dient u. a. zur Auflösung sogenannter Schlüsselbeziehungen in relationalen Datenmodellen [KE96], weshalb sie von fundamentaler Bedeutung ist und in der Anwendung sehr häufig gebraucht wird:

$$R_1 \bowtie R_2 := \pi_{k+1,\ldots,r_1+r_2}(\sigma_{A_1=B_1,\ldots,A_k=B_k}(R_1 \times R_2))$$

1.5.1.2 Theta-Join

Theta steht hier als Platzhalter für ein zweistelliges Vergleichsprädikat, das wie im einfachsten Fall des Equi-Joins auf ein Paar von Attributen angewendet wird:

$$\theta \in \{=, <, >, \leq, \geq, \neq, \not<, \not>, \not\leq, \not\geq\} \qquad \Rightarrow$$
$$R_1 \bowtie_{i\theta j} R_2 := \sigma_{A_i \theta B_j}(R_1 \times R_2)$$

1.5.1.3 Band-Join

Beim Band-Join ist die Bedingung etwas lockerer als beim Equi-Join, hier müssen die joinenden Tupel auf einem Attribut nicht gleich sein, sondern dürfen um einen gewissen Betrag ε voneinander abweichen:

$$R_1 \bowtie_{i,j,\varepsilon}^{band} R_2 := \sigma_{|A_i - B_j| \leq \varepsilon}(R_1 \times R_2)$$

1.5.1.4 Räumlicher Verbund - Spatial-Join

Beim Räumlichen Verbund besitzen beide Relationen eine räumliche Struktur als Attribut, auf denen als Joinprädikat gefordert wird, daß sich diese schneiden. Von besonderem Interesse sind dabei wegen ihres einfachen Aufbaus achsenparallele Rechtecke, die üblicherweise als Schlüssel für komplexere Objekte genutzt werden:

$$R_1 \bowtie_{i,j}^{spatial} R_2 := \sigma_{A_i \cap B_j \neq \emptyset}(R_1 \times R_2)$$

1.5.1.5 Zeitlicher Verbund - Temporal-Join

Den zeitlichen Verbund kann man als den eindimensionalen Spezialfall des Räumlichen Verbunds interpretieren, nur daß den eindimensionalen Intervallen hier statt einer räumlichen eine zeitliche Bedeutung zugedacht wird:

$$R_1 \bowtie_{i,j}^{temporal} R_2 := \sigma_{A_i \cap B_j \neq \emptyset}(R_1 \times R_2)$$

1.5.2 Joinverfahren

Hier sollen einige gebräuchliche physische Operatoren zur Joinverarbeitung vorgestellt werden. Dabei soll keine tiefgreifende Analyse der Verfahren vorgenommen werden, sondern nur eine kurze Vorstellung unter besonderer Beachtung der für diese Arbeit relevanten Aspekte, also insbesondere der Fähigkeit zur Produktion früher Ergebnisse.

Berechnet werden soll $R_1 \bowtie_F R_2 := \sigma_F(R_1 \times R_2)$; auf eine zusätzliche Projektion wird hier verzichtet.

1.5.2.1 Nested-Loops-Join

Der Nested-Loops-Join (Algorithmus 1.1) ist der einfachste physische Joinoperator. Er bildet den Joinoperator als Verkettung der Bildung des kartesischen Produktes mit der Selektion nach. Dazu durchläuft eine äußere Schleife die Relation R_1 und testet für jedes darin enthaltene Tupel t_1 mittels einer inneren Schleife, die alle Tupel t_2 in R_2 durchläuft, ob (t_1, t_2) das Joinprädikat F erfüllt.

Algorithmus 1.1 Nested-Loops-Join

IN: R_1, R_2 die Eingaberelationen
OUT: *Result* die Joinergebnisse

```
1: Let Result := ∅;
2: for t1 ∈ R1 do
3:   for t2 ∈ R2 do
4:     if F(t1, t2) then
5:       Let Result := Result ∪ {(t1, t2)};
6:     end if
7:   end for
8: end for
```

Neben der einfachen Implementierung hat der Nested-Loops-Join den Vorteil, daß er ohne Einschränkung an das Prädikat F und die Relationen R_i immer verwendbar ist.

Wurde die äußere Schleife für eine Teilmenge $T_1 \subset R_1$ abgearbeitet, so ist zu diesem Zeitpunkt der Teiljoin $T_1 \bowtie_F R_2$ berechnet. Der Algorithmus ist also nicht blockierend und produziert seine Ergebnisse kontinuierlich.

Der entscheidende Nachteil des Nested-Loops-Joins ist die Tatsache, daß die innere Schleife $|R_1|$ mal durchlaufen wird und jedes Mal die gesamte Relation R_2 komplett gelesen werden muß. Paßt diese nicht in den Hauptspeicher, so muß dies (zumindest teilweise) vom Externspeicher geschehen. Dieser elementare Nachteil kann zwar durch Optimierungen [See98] leicht verringert werden, an der Zeit- und Externspeicherzugriffskomplexität von $O(|R_1| * |R_2|)$ ändert dies jedoch nichts, wodurch der Nested-Loops-Join für viele Anwendungen ungeeignet ist, da zu langsam.

1.5.2.2 Index-Nested-Loops-Join

Voraussetzung für die Verwendung des Index-Nested-Loops-Joins (Algorithmus 1.2) ist das Vorhandensein einer geeigneten Indexstruktur über einer

Algorithmus 1.2 Index-Nested-Loops-Join

IN: R_1, R_2 die Eingaberelationen
OUT: *Result* die Joinergebnisse

1: Let $Result := \emptyset$;
2: **for** $t_1 \in R_1$ **do**
3: Let $Result := Result \cup Index_{R_2}.query(t_1, F)$;
4: **end for**

der beiden Relationen, o. B. d. A. R_2. Diese Struktur muß eine Anfrage *query* ermöglichen, die zu einem Tupel $t_1 \in R_1$ alle Tupel $t_2 \in R_2$ liefert, für die $F(t_1, t_2)$ gilt. Die innere Schleife des Nested-Loops-Joins 1.5.2.1 stellt gewissermaßen eine solche Struktur dar. Um Vorteile gegenüber dieser zu erhalten, muß die Indexstruktur die Anfrage mit einer besseren Komplexität als $O(|R_2|)$ erlauben. Eine Anfrage mit k Resultaten kann natürlich nicht besser als $O(k)$ sein.

Für die Relation, auf der solch eine Indexstruktur vorliegt, entfällt die in 1.4 gestellte Bedingung an die zufällige Lesereihenfolge, da nun mittels des Index alle mit einem Anfragetupel joinenden Tupel der Relation gefunden werden können, ohne die Relation komplett lesen zu müssen.

Liegt zum Beispiel für die Relation R_2 ein nach dem Attribut B_1 sortierter Index (z. B. in Form eines Suchbaumes) vor, und soll ein Equi-Join zu dem Attribut A_1 von R_1 berechnet werden, so können die k mit $t_1 \in R_1$ joinenden Tupel in $O(log|R_2| + k)$ gefunden werden.

Wie beim Nested-Loops-Join 1.5.2.1 erfolgt die Produktion von Ergebnissen von Beginn an und auch noch relativ kontinuierlich; je nach Art der Indexstruktur brauchen die Anfragen an diese üblicherweise eine Anlaufzeit, um dann die Ergebnisse zu liefern. Auch hier ist nach Abarbeitung der Schleife für $T_1 \subset R_1$ jeweils der Teiljoin $T_1 \bowtie_F R_2$ berechnet.

Liegt auf keiner der Relationen ein für F geeigneter Index vor, so kann man einen solchen natürlich erzeugen, um dann den Index-Nested-Loops-Join anwenden zu können. Da von einem solchen temporären Index Anforderungen an normale Indizes wie Erweiterbarkeit nicht erfüllt sein müssen, kann der Aufbau eines für den Join spezialisierten Index bezüglich der Gesamtlaufzeit des Joins sogar günstiger sein als die Nutzung eines bestehenden. Während des Indexaufbaus werden aber keine Joinergebnisse produziert, wodurch das Gesamtverfahren blockierend ist. Verwendet man als Indexstruktur eine Hashtabelle, so spricht man von einem *Hash-Join*.

Algorithmus 1.3 Ripple-Join

IN: R_1, R_2 die Eingaberelationen
OUT: *Result* die Joinergebnisse

1: Let $Result := \emptyset$;
2: Let $T_1 := \emptyset$, $T_2 := \emptyset$;
3: **while** $R_1 \neq \emptyset$ OR $R_2 \neq \emptyset$ **do**
4: Let $T_1^* \subset R_1$, $R_1 := R_1 \backslash T_1^*$, $T_1 := T_1 \cup T_1^*$;
5: Let $Result := Result \cup (T_1^* \bowtie_F T_2)$
6: Let $T_2^* \subset R_2$, $R_2 := R_2 \backslash T_2^*$, $T_2 := T_2 \cup T_2^*$;
7: Let $Result := Result \cup (T_2^* \bowtie_F T_1)$
8: **end while**

1.5.2.3 Ripple-Joins

1.5.2.3.1 Grundlegende Ripple-Joins Die Eigenschaft der vorstehenden Algorithmen, zwischenzeitlich für $T_1 \subset R_1$ den Teiljoin $T_1 \bowtie_F R_2$ berechnet zu haben, ist für die Berechnung von Schätzungen zwar vorteilhaft, da dies den Join einer Stichprobe mit der gesamten anderen Relation darstellt, hat jedoch auch entscheidende Nachteile. Zum einen muß beim Nested-Loops-Join 1.5.2.1 dazu R_2 komplett betrachtet werden, zum anderen ist das Verfahren am Anfang sehr anfällig gegen statistische Schwankungen in R_1.

Daher wünscht man sich ein Verfahren, bei dem zwischenzeitlich der Join aus je einer Stichprobe des Eingaberelationen berechnet ist. Eine Klasse solcher Algorithmen wurde 1999 von Peter J. Haas und Joseph M. Hellerstein in [HH99] unter dem Namen *Ripple*[3] *Joins* vorgestellt und bildete die Grundlage für viele Weiterentwicklungen, von denen hier in 1.5.2.3.2, 1.5.2.3.3 und 1.5.2.3.4 einige vorgestellt werden.

Hat der Ripple-Join (Algorithmus 1.3) also für $T_1 \subset R_1$ und $T_1 \subset R_1$ den Teiljoin $T_1 \bowtie_F T_2$ berechnet, so wählt er Teilmengen T_1^* und T_2^* des noch nicht verarbeiteten Teils der Eingaberelationen und joint diese mit den bereits vorher gelesenen Teilen und miteinander.

Die Variante des Ripple-Joins wird zunächst durch die Vorgehensweise zur Wahl von T_i^* bestimmt:

Square-Ripple-Join Im einfachsten Fall werden jeweils einelementige Mengen T_i^* gewählt.

Rectangular-Ripple-Join Der Rectangular-Ripple-Join berücksichtigt, daß die Eingaberelationen verschieden groß sein können oder Tupel aus

[3] engl. ripple = kleine Welle

ihnen mit verschiedenen Raten zur Verfügung gestellt werden. Dazu wählt er eine der Teilmengen T_i^* leer und die andere einelementig und wechselt die Relation, aus der ein Tupel verarbeitet wird, in Abhängigkeit von Eingabegröße oder -rate.

Block-Ripple-Join Beim Block-Ripple-Join können beide Mengen T_i^* mehrere Elemente haben. Damit trägt er der Tatsache Rechnung, daß die Relationen häufig blockweise, also mehrere auf einmal, vom Externspeicher gelesen werden. Durch die blockweise Verarbeitung werden aber seltener in der Ausführung des Algorithmus Punkte erreicht, an denen ein Teiljoin $T_1 \bowtie_F T_2$ berechnet ist, wodurch sich die statistische Auswertbarkeit verschlechtert.

Rectangular Block-Ripple-Join Der Rectangular Block-Ripple-Join nun vereint die Eigenschaften der beiden Varianten zuvor und verarbeitet in bestimmtem Rhythmus abwechselnd Blöcke der Eingaberelationen.

Die Abbildungen 1.1 bis 1.4 verdeutlichen die Arbeitsweise der verschiedenen Ripple-Join Varianten. Die Gesamtfläche steht dabei jeweils für das kartesische Produkt $R_1 \times R_2$, das nach Tupeln, die F erfüllen, durchsucht wird. Grau überdeckte Bereiche stehen für Tupel, für die bereits entschieden ist, ob sie im Join liegen. Die dunkelgraue Fläche symbolisiert dabei den zum betrachteten Zeitpunkt bereits berechneten Teiljoin $T_1 \bowtie_F T_2$ und die hellgrauen Flächen jeweils den Teil des Joins, der während jeweils eines der nächsten Durchläufe der *while* Schleife von Algorithmus 1.3 berechnet wird.

Neben der immer noch relativ einfachen Implementierung liegt der Vorteil der in diesem Abschnitt vorgestellten Ripple-Joins in der sehr gut skalierbaren Updaterate für Schätzer. Ein Update des Schätzers kann auf jeden Fall nach jedem Durchlauf der *while* Schleife erfolgen. Die Größe der dabei verarbeiteten Blöcke kann man neben systembedingten Parametern auch von Einstellungen des Benutzers abhängig machen und auch während der Laufzeit dynamisch ändern. Sogar ein Wechsel zwischen den Varianten wäre möglich.

Die Nachteile des Algorithmus treten in den Vordergrund, sobald T_1 und T_2 so groß werden, daß sie nicht mehr gemeinsam in den Hauptspeicher passen und teilweise auf den Externspeicher ausgelagert werden müssen. Haas und Hellerstein meinen in [HH99], diesem Problem komme in der Praxis wenig Bedeutung zu, da zu diesem Zeitpunkt die Qualität der Schätzer schon hinreichend gut sei. Will man diese jedoch weiter verbessern oder den Join komplett berechnen, so bereitet das Einbrechen der Performanz beim Speicherüberlauf erhebliche Probleme.

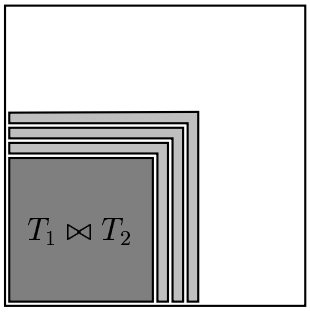


Abbildung 1.1: Square-Ripple-Join

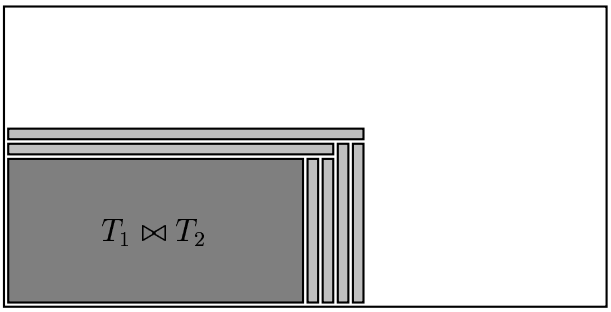


Abbildung 1.2: Rectangular-Ripple-Join

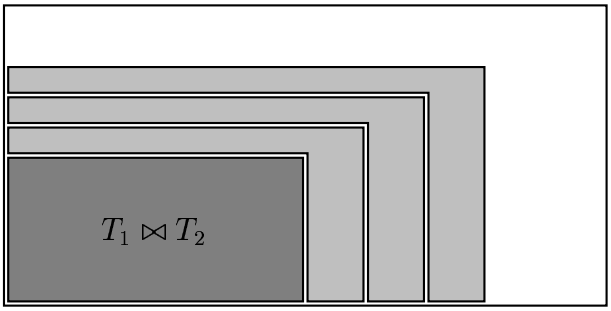


Abbildung 1.3: Block-Ripple-Join

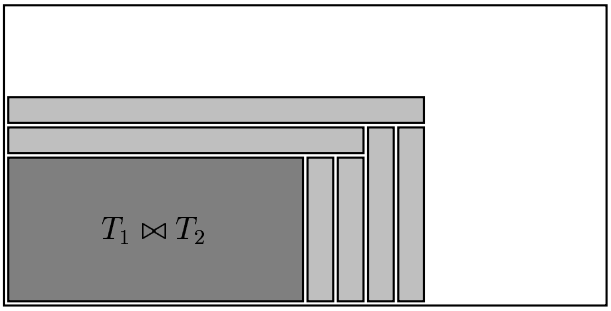


Abbildung 1.4: Rectangular-Block-Ripple-Join

Algorithmus 1.4 Hash-Ripple-Join

IN: R_1, R_2 die Eingaberelationen
OUT: *Result* die Joinergebnisse

1: Let $Result := \emptyset$;
2: Let H_1 , H_2 be empty Hashtables;
3: Let $T_1 := \emptyset$, $T_2 := \emptyset$;
4: **while** $R_1 \neq \emptyset$ OR $R_2 \neq \emptyset$ **do**
5: Let $T_1^* \subset R_1$, $R_1 := R_1 \backslash T_1^*$, $T_1 := T_1 \cup T_1^*$;
6: $H_1.insert(T_1^*)$;
7: Let $Result := Result \cup H_2.query(T_1^*, F)$;
8: Let $T_2^* \subset R_2$, $R_2 := R_2 \backslash T_2^*$, $T_2 := T_2 \cup T_2^*$;
9: $H_2.insert(T_2^*)$;
10: Let $Result := Result \cup H_1.query(T_2^*, F)$;
11: **end while**

1.5.2.3.2 Hash-Ripple-Join Der *Hash-Ripple-Join* (Algorithmus 1.4) ist die gebräuchlichste Variante, die in 1.5.2.2 vorgestellte Idee des Aufbaus eines Index auf den Ripple-Join 1.5.2.3.1 zu übertragen [HH99]. Da auf nicht blockierende Weise ein Index über eine der Relationen aufgebaut werden soll, bietet es sich an, diesen jeweils für die Teilmengen T_i bereitzustellen. Um wie der Ripple-Join 1.5.2.3.1 die Asymmetrie zwischen der Behandlung der inneren und äußeren Relation beim Nested-Loops-Join 1.5.2.1 aufzulösen, muß hier im Gegensatz zum Index Nested-Loops-Join 1.5.2.2 über beide Relationen ein Index aufgebaut werden.

Die im Schleifendurchlauf neu ausgewählten Tupel einer Relation werden also in die zur Relation gehörige Hashtabelle eingefügt und fragen die Hashtabelle der anderen Relation nach Joinergebnissen an.

Eine parallele Variante des Hash-Joins wird in [SD89] vorgestellt, eine Erweiterung zum parallelen Hash-Ripple-Join in [LEHN02].

Durch die Verwendung der speziellen Indexstruktur wird zwar die Performanz erhöht, aber es können nur noch die Arten von Joins berechnet werden, die mit einer Hashtabelle realisiert werden können (siehe 1.5.3.1). Die Vor- und Nachteile der in 1.5.2.3.1 beschriebenen Ripple-Joins bleiben erhalten.

1.5.2.3.3 XJoin Mit dem *XJoin* [UF99], [UF00] wird hier eine Weiterentwicklung des Hash-Ripple-Joins 1.5.2.3.2 vorgestellt, die sich auch der Problematik annimmt, daß die Hashtabellen nach einer Anlaufzeit nicht mehr in den Hauptspeicher passen. Sobald dies eintritt, müssen Teile der Hashtabellen auf den Externspeicher ausgelagert werden. Aus Gründen der Sym-

metrie werden Teile beider Hashtabellen im Hauptspeicher belassen. Damit weiterhin die Tatsache, wie früh ein Ergebnis produziert wird, i. w. von der zufälligen Eingabereihenfolge abhängt, darf auch keine der Partitionen der Hashtabellen komplett ausgelagert werden. Demnach behält jede Partition einen Teil des Hauptspeichers, der so lange Tupel aufnimmt, bis er voll ist. Sollen nun weitere Tupel in die Partition eingefügt werden, so werden sie in eine Erweiterung der Partition auf dem Externspeicher ausgelagert. Die Anfrage nach Joinpartnern an die andere Relation erfolgt jetzt jedoch nicht mehr bezüglich der gesamten Hashtabelle, sondern nur noch bezüglich der Teile der Partitionen, die sich im Hauptspeicher befinden.

So könnte man prinzipiell verfahren, bis die Eingaberelationen alle komplett gelesen wurden. Die Spezialität des XJoin ist es nun, daß er Verzögerungen bei deren Eintreffen nutzt, um zusätzliche frühe Ergebnisse zu produzieren. Die verfügbare Zeit nutzt er, um zuvor ausgelagerte Teile der Partitionen einzulesen und gegen die Hashtabelle der anderen Relation zu testen, wo sie mit nach ihrer Auslagerung eingetroffenen Tupeln joinen können.

Sind dann alle Eingaberelationen komplett eingetroffen, so müssen in einer abschließenden Phase noch die fehlenden Ergebnisse berechnet werden, die zuvor noch nicht erzeugt wurden, weil beteiligte Tupel ausgelagert wurden. Um dieses Problem jetzt überschaubar lösen zu können, verwendet man in beiden Hashtabellen dieselbe Partitionierungsfunktion, so daß Tupel, die miteinander joinen könnten, sich jetzt in den Partitionen mit gleicher Nummer der beiden Hashtabellen finden. Es genügt nun, nacheinander diese Paare von Partitionen vom Externspeicher einzulesen und deren Join zu berechnen[4]. In dieser letzten Phase erfolgt die Ergebnisproduktion deutlich abhängig von der Partitionierung; eine statistische Auswertung macht daher keinen Sinn mehr.

Sowohl während der zwischenzeitlichen als auch der abschließenden Wiedereinlesephase können bereits zuvor erzeugte Ergebnisse erneut auftreten. Die Eliminierung dieser Duplikate kann mit Zeitstempeln realisiert werden. Für Details sei z. B. auf [VNB02] verwiesen, hier nur der Hinweis, daß diese Zeitstempel zusätzlichen Speicher verbrauchen und zusätzliches I/O verursachen, da sie mit den Tupeln ausgelagert werden.

1.5.2.3.4 MJoin Der *MJoin* [VNB02] ist eine Weiterentwicklung des XJoins (1.5.2.3.3) zur Berechnung mehrdimensionaler Joins. Dazu wird nun für jede der r Eingaberelationen eine Hashtabelle aufgebaut. Die verschiedenen Phasen laufen analog zum XJoin ab. Ein Tupel t kann aber jetzt nicht

[4] Paßt keine davon in den Hauptspeicher, löst man dieses Problem wieder mittels eines externen Joinverfahrens.

mehr einfach nur die Hashtabelle der anderen Relation anfragen, sondern muß sich seine Joinpartner aus mehreren Hashtabellen zusammensuchen. Dazu wird dem Algorithmus für jede Eingaberelation eine Anfragereihenfolge vorgegeben, nach der er die Hashtabellen abfragt. Findet er dabei einmal kein passendes Tupel, so wird die Anfrage abgebrochen, anderenfalls das Kreuzprodukt aus $\{t\}$ mit den Ergebnismengen der Anfrage gebildet und dessen Elemente mit der Joinbedingung getestet. In der letzten Phase sollten jetzt $r-1$ Partitionen in den Hauptspeicher passen, die dann von Tupeln, die aus der Partition der verbleibenden Relation vom Externspeicher gelesen werden, angefragt werden. Zur Duplikatbeseitigung sei wiederum auf [VNB02] verwiesen.

1.5.2.4 Sort-Merge-Join

Neben dem Nested-Loops-Join (1.5.2.1) und dem Hash-Join (Seite 11) zählt der *Sort-Merge-Join* [BE77] zu den klassischen Joinalgorithmen. Er wird hier zunächst in der einfachsten Variante zur Berechnung von Equi-Joins vorgestellt; eine erweiterte Diskussion erfolgt in 2.8.

1.5.2.4.1 Algorithmus Die grundlegende Idee des Sort-Merge-Joins ist es, die Eingaberelationen zunächst bezüglich einer totalen Ordnung O auf der Menge von Attributen, deren Gleichheit von der Joinbedingung gefordert wird, zu sortieren. Steht nun ein Tupel a in der Ordnung vor einem Tupel b mit $a \neq_O b$, so stehen auch alle Joinpartner von a in der sortierten anderen Relation vor denen von b. Durchläuft man die sortierten Folgen nun derart, daß man immer das kleinste unbearbeitete Element bezüglich O auswählt, so findet man dessen Joinpartner genau am Anfang der anderen Folge.

Von Interesse ist auch hier natürlich der Fall, daß die Relationen nicht in den Hauptspeicher passen. Der Sortiervorgang muß dann durch ein externes Sortierverfahren erfolgen. Dabei wird der Hauptspeicher mit Tupeln einer Relation gefüllt, diese werden sortiert und als sogenannter *Run* auf den Externspeicher geschrieben. Dies wird wiederholt, bis die Relation komplett gelesen wurde. Dann erfolgen Merge-Schritte, die jeweils aus mehreren sortierten Runs einen machen, bis die Relation als ein sortierter Run auf dem Externspeicher vorliegt. Die andere Relation wird analog behandelt, und dann kann der zuvor beschriebene Merge der beiden verbliebenen Runs zur Produktion der Joinergebnisse erfolgen. Abbildung 1.5 veranschaulicht diesen Ablauf.

1.5.2.4.2 Verbesserungen Dieser elementare Algorithmus wurde durch diverse Optimierungen verbessert, von denen einige hier vorgestellt werden.

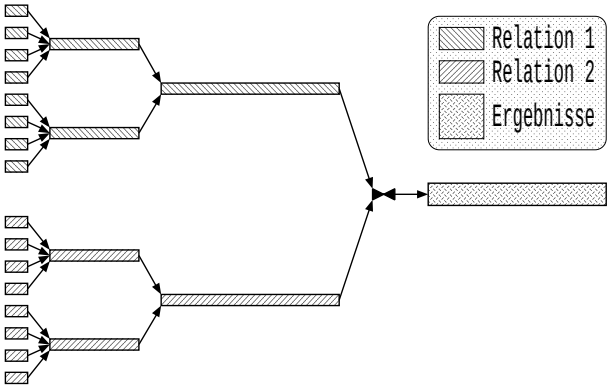


Abbildung 1.5: Sort-Merge-Join

Seitenweises Lesen Bei den Merge-Schritten der Sortierung könnte man theoretisch so viele Runs zu einem vereinen, wie Tupel gleichzeitig in den Hauptspeicher passen, da man beim Lesen der Runs nur jeweils das aktuell kleinste[5] Element jedes Runs verfügbar haben muß. Da sich der Hauptspeicher aber üblicherweise in p Seiten unterteilt und auch seitenweise gelesen und geschrieben wird, benutzt man aus Effizienzgründen eine Seite für den zu produzierenden und die verbleibenden Seiten für $p-1$ zu lesende Runs.

Replacement Selection Die Anzahl der Merge-Schritte hängt nun von der der initialen Runs ab, die durch das Sortieren eines Hauptspeichers voller Tupel entstehen. Knuth [Knu98] schlägt ein Verfahren namens *Replacement Selection* vor, um die Zahl der initialen Runs zu verringern. Dazu wird der Hauptspeicher ebenfalls mit Tupeln gefüllt, um dann immer das kleinste unmarkierte Tupel im Hauptspeicher auszuwählen[6] und in den zu produzierenden Run zu schreiben. Nach jedem Schreiben wird jedoch in den freigewordenen Platz ein neues Tupel nachgeladen. Ist dieses größer oder gleich dem zuletzt geschriebenen, so kann es in der Folge auch noch für den Run verwendet werden. Anderenfalls wird es markiert und verbleibt im Speicher. Befinden sich dort nur noch markierte Tupel, so werden die Markierungen entfernt, und es wird mit der Produktion eines neuen Runs begonnen. Die so erzeugten Runs erreichen bei zufälliger Eingabereihenfolge im Mittel doppelte Hauptspeichergröße [Knu98].

[5] Hier wird eine aufsteigende Sortierung beschrieben.
[6] Dazu kann man die Tupel z. B. in einem Heap organisieren.

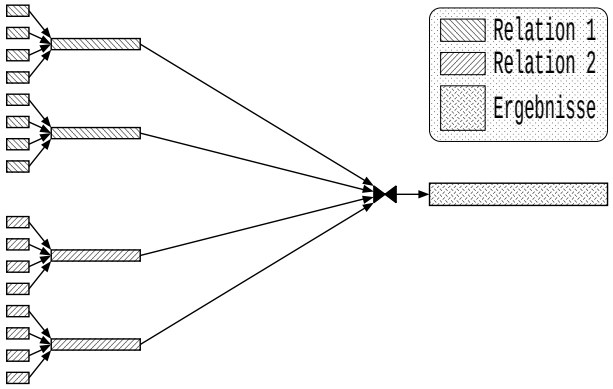


Abbildung 1.6: Sort-Merge-Join mit *Join-During-Merge* an der Wurzel

Join-During-Merge Die Merge-Schritte zum Sortieren und zum Joinen ähneln sich sehr; in beiden wird jeweils das kleinste unverarbeitete Element aus Runs ausgewählt und bearbeitet. Da man zumeist keine weitere Verwendung für die fertig sortiert als je ein Run auf dem Externspeicher liegenden Folgen hat, wird in [NP85] ein *Join-During-Merge* vorgeschlagen, wobei die jeweils letzten Merge-Schritte der Sortierung beider Eingaben nun verzahnt ausgeführt werden, indem das kleinste Element für den Join jetzt also aus mehreren Runs der beiden Relationen ausgewählt wird, wie in Abbildung 1.6 dargestellt.

Da die Relationen sich dafür den Hauptspeicher teilen müssen, fallen nun aber ggf. noch Merge-Schritte an, um die Zahl der Runs hinreichend zu reduzieren. Insgesamt wird die Zahl der Externspeicherzugriffe aber auf jeden Fall reduziert, in den meisten Fällen entfällt je ein komplettes Lesen und Schreiben beider Relationen.

Diverse Verbesserungen Der Algorithmus wurde immer wieder verbessert. Z. B. kann man, wenn die Größe der Eingaben bekannt ist, die Zahl der Runs und Merge-Schritte vorausberechnen und feststellen, wie viele Seiten man im letzten Schritt (Join-During-Merge) für verbliebene Runs braucht. Der übrige Speicher kann in diesem Schritt dann für Tupel verwendet werden, die man zuvor keinem Run zuschlägt und nun erst einliest und wie einen zusätzlichen Run behandelt. Weiteres Optimierungspotential bietet die Wahl des Replacement-Selection Verfahrens, siehe dazu z. B. [NBC+94] und [Lar02].

Ein Problem tritt auf, wenn ein Tupel t mit sehr vielen anderen Tupeln joint (dieser Effekt wird als *Skew* bezeichnet). Die Joinpartner befinden sich

zwar am Anfang der Sortierung der anderen Relation, von dieser ist jedoch in der Joinphase nur eine Seite pro verbliebenem Run geladen. Steht das letzte Tupel in einer Seite bezüglich O nicht hinter t, so können sich weitere Joinpartner auf dem Externspeicher befinden, die nachgeladen werden müssen. Mit dieser Problematik setzt sich [LGS02] auseinander; in dieser Arbeit wird sie in 2.4 wieder aufgenommen.

1.5.3 Einsatzgebiete von Jointechniken

Welcher physikalische Operator zur Berechnung eines konkreten Joins in einem Datenbanksystem verwendet wird, entscheidet zumeist eine spezielle Komponente, die Optimierer genannt wird [Ber02]. Dieser zieht neben den Relationenschemata und der Joinbedingung noch weitere Kriterien hinzu, wie z. B. Metainformationen über die Relationeninstanzen. An dieser Stelle soll nur ein allgemeiner Überblick darüber gegeben werden, welche grundlegenden Jointechniken sich zur Berechnung welcher Arten von Joins anbieten.

1.5.3.1 Hashbasierte Joinverfahren

Hashbasierte Joinverfahren basieren auf dem Divide-and-Conquer Prinzip. Die zu joinenden Relationen werden mit Hilfe von Hashfunktionen f_1 und f_2 jeweils in Partitionen geteilt, und das Joinproblem wird dann auf den Paaren korrespondierender Partitionen gelöst. Damit dabei keine Joinresultate ausgelassen werden, müssen joinende $r_1 \in R_1$ und $r_2 \in R_2$ auf dieselbe Partitionsnummer abgebildet werden:

$$F(r_1, r_2) \Rightarrow f_1(r_1) = f_2(r_2) \tag{1.2}$$

Betrachtet man die Urbildbereiche U_1 und U_2 von f_1 und f_2, also die Vereinigung aller möglichen Relationeninstanzen R_1 bzw. R_2 für einen Join nach einem Prädikat F, so zerfallen diese in Äquivalenzklassen bezüglich ihres Joinverhaltens durch Bildung der reflexiven transitiven Hülle von[7]:

$$u_1 \equiv u_1' :\Leftarrow \exists u_2 \in U_2 : F(u_1, u_2) \wedge F(u_1', u_2)$$

$$u_2 \equiv u_2' :\Leftarrow \exists u_1 \in U_1 : F(u_1, u_2) \wedge F(u_1, u_2')$$

Damit Bedingung 1.2 genügt wird, müssen die Hashfunktionen f_i diese Äquivalenzrelation erhalten:

$$u_i \equiv u_i' \Rightarrow f_i(u_i) = f_i(u_i')$$

[7] Dies entspricht der Bestimmung der Zusammenhangskomponenten des vollständig bipartiten ungerichteten Graphen mit Knotenmenge $U_1 \dot\cup U_2$ und Kantenmenge $\{(a, b) \mid F(a, b)\}$.

Joinprädikat $F(u,v)$	Menge der Äquivalenzklassen
$u = v$	$\{\{z\} \mid z \in \mathbb{Z}\}$
$u - v = 2$	$\{\{z\} \mid z \in \mathbb{Z}\}$
$\lvert u - v\rvert = 2$	$\{4\mathbb{Z},\ 4\mathbb{Z}+1,\ 4\mathbb{Z}+2,\ 4\mathbb{Z}+3\}$
$\lvert u - v\rvert < 2$	$\{\mathbb{Z}\}$

$$U_1 = U_2 := \mathbb{Z}$$

Tabelle 1.1: Beispiele für Äquivalenzklassen bei Joins

Die Anzahl der möglichen nichtleeren Partitionen für das Hashing ist also nach oben durch die kleinere Zahl der Äquivalenzklassen der U_i begrenzt, die wiederum von F, also insbesondere auch der Art des Joins, bestimmt wird. Für $U_1 = U_2 := \mathbb{Z}$ sind einige Beispiele für Äquivalenzklassen zu verschiedenen Joinprädikaten in Tabelle 1.1 angegeben.

Bei Equi-Joins erzeugt jedes Element des Wertebereichs des Attributes, auf dem die Gleichheit verlangt wird, eine Äquivalenzklasse. Daher sind hashbasierte Joinverfahren hier i. a. sehr gut einsetzbar und werden in der Praxis auch zumeist verwendet.

Für viele Arten von Joins tritt jedoch der Extremfall auf, daß nur je eine Äquivalenzklasse besteht, so auch bei der Mehrzahl der in 1.5.1 beschriebenen. Dies gilt insbesondere dann, wenn es in einer Relation ein Tupel geben kann, das mit allen der anderen joinen würde. Ein solches kann man mit dem alle Rechtecke bzw. Intervalle der anderen Relation umfassenden Rechteck bzw. Intervall für Spatial- und Temporal-Join leicht angeben. Aber auch beim Band-Join liegen i. a. alle Tupel in einer Äquivalenzklasse, die man ausgehend von einem Tupel wegen der Transitivität der Äquivalenzrelation durch sukzessive Hinzunahme aller Tupel in der Epsilonumgebung konstruieren kann.

Für solche Fälle einer zu geringen Anzahl von Äquivalenzklassen muß ein hashbasiertes Joinverfahren daher modifiziert werden. Dazu kann man z. B. die Äquivalenzklassen aufteilen und Tupel, die mehrere dieser Teile schneiden, in alle entsprechenden einfügen. Dieses Vorgehen entspricht demjenigen bei einigen mehrdimensionalen Indexstrukturen [BSS98].

1.5.3.2 Sortierbasierte Joinverfahren

Wie bereits in 1.5.2.4 erläutert, eignet sich auch der Sort-Merge-Join zur Berechnung von Equi-Joins. In 2.8 wird anhand des Progressive-Merge-Joins aufgezeigt, daß sortierbasierte Joinverfahren sich auch zur Berechnung von Band-, Spatial- und Temporal-Join eignen.

Allgemein lassen sich als Einsatzgebiet von sortierbasierten Joinverfahren die Arten von Joins identifizieren, bei denen man eine gemeinsame totale Ordnung auf den Relationen definieren kann, bezüglich derer zwischen joinenden Tupeln mit hoher Wahrscheinlichkeit nur ein kleiner Teil der anderen Tupel liegt.

1.5.3.3 Weitere Joinverfahren

Der einfache Nested-Loops-Join produziert alle Elemente des kartesischen Produkts der Relationen und testet diese mit dem Joinprädikat. Damit ist er zur Berechnung aller Arten von Joins geeignet. Reale Effizienzgewinne demgegenüber lassen sich nur bei Joins mit geringerer Selektivität erzielen, da dann das Zusammenbringen und Testen vieler Kombinationen eingespart werden kann. Joins mit hoher Selektivität (wie viele Varianten des Theta-Joins) kann man daher durchaus mit Hilfe einer Variante des Nested-Loops-Joins berechnen.

Zu einem Join $R_1 \ltimes_F R_2$ mit Selektivität s ist der Anti-Join durch $R_1 \overline{\ltimes}_F R_2 := R_1 \ltimes_{\neg F} R_2$ definiert, dessen Selektivität damit $1-s$ ist. Kann ein Join gut mit einem bestimmten Verfahren berechnet werden, so eignet sich dieses zumeist für den Anti-Join nicht, so daß auch dafür oft ein allgemein verwendbares Joinverfahren die geeignete Wahl ist.

Zusammenfassung

Eine Verbundoperation (Join) ist eine in Datenbanksystemen häufig benötigte Operation, welche die ein gegebenes Prädikat erfüllenden Tupel des Kreuzproduktes ihrer Eingaberelationen berechnet. Dabei ist es wünschenswert, schon nach kurzer Zeit Ergebnisse und Schätzungen über deren Gesamtzahl zu erhalten. Zu diesem Zweck wurden verschiedene insbesondere auf Hashing basierende Joinalgorithmen angepaßt.

Kapitel 2

Progressive-Merge-Join

Übersicht

Dieses Kapitel beschäftigt sich mit dem Algorithmus *Progressive-Merge-Join*, der in [DSTW02b] vorgestellt wurde. In 2.1 werden zunächst die grundlegenden Konzepte des Progressive-Merge-Joins motiviert, der dann in 2.2 vorgestellt wird. In 2.3 wird ein Beweis für die Korrektheit des Algorithmus angegeben, welcher mit weniger Voraussetzungen als der in [DSTW02b] auskommt. Bevor in 2.5 Aspekte seiner Implementierung erläutert werden, wird in 2.4 die Problematik des sogenannten Skew besprochen. Abschnitt 2.6 beschreibt das Verfahren zur Gewinnung eines Schätzers und damit verbundene Problematiken. Abschließend wird der Progressive-Merge-Join in 2.7 auf seine Effizienz und in 2.8 auf seine Einsatzgebiete hin untersucht.

2.1 Einführung

In 1.5.2.4.2 wurde das Konzept vorgestellt, durch die Verschmelzung der Merges des Sortierens und des Joinens Externspeicherzugriffe einzusparen. Diese Idee läßt sich nun auch auf die vorangehenden Merge-Schritte des Sortierens übertragen. Statt also die Merge-Schritte des externen Sortierens getrennt nach Relationen durchzuführen, werden mehrere Runs beider Relationen zu je einem Run zusammensortiert und dabei der Join dieser erzeugten Runs berechnet, wie beispielhaft in Abbildung 2.1 dargestellt. Diese zeigt auch den dabei angestrebten Vorteil, daß nun Joinergebnisse bereits während der frühen Merge-Phasen produziert werden.

Erzeugt man nur beim Mergen Ergebnisse, so müssen doch zunächst einmal die dazu nötigen Runs berechnet werden. Dies kann immer noch beträchtliche Zeit in Anspruch nehmen; das Verfahren ist also weiterhin als

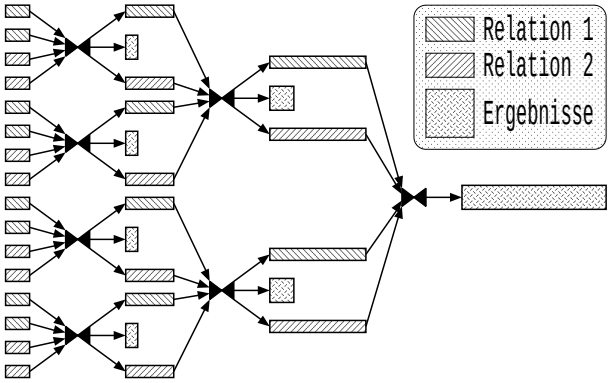


Abbildung 2.1: Sort-Merge-Join mit *Join-During-Merge*

blockierend anzusehen. Wenn man nun das im Merge-Schritt eingeführte frühe Joinen der produzierten Runs betrachtet, so braucht man von jeder Relation einen. Bei der Generierung der initialen Runs benutzt man jedoch den gesamten Speicher für Tupel einer Relation. Teilt man den Speicher nun auf und produziert Runs beider Relationen, so kann man auch hier nach dem Sortieren die beiden generierten Runs joinen. Abbildung 2.2 zeigt, wie nun bereits bei der Rungenerierung Ergebnisse produziert werden.

Diese Änderungen machen aus dem blockierenden Sort-Merge-Join ein Verfahren, mit dem man frühe Ergebnisse erzeugen kann. In diesem Kapitel wird nun erläutert, wie dies im Detail funktioniert und welche Arten von Joins damit berechnet werden können [DSTW02b], [Dit02].

2.2 Der Algorithmus PMJ

Der *Progressive-Merge-Join* wird hier in Anlehnung an [DSTW02b] vorgestellt, da diese Arbeit auf jener Variante aufbaut. Die Algorithmen dieses Kapitels im Pseudocode und die zugehörigen Bezeichnungen wurden dazu i. w. wörtlich übernommen, die Algorithmenbeschreibung und die weiteren Abschnitte dieses Kapitels hingegen, soweit nicht anderweitig gekennzeichnet, neu erstellt.

2.2.1 Grundlegender Algorithmus

Der Algorithmus PMJ (Algorithmus 2.1) teilt sich in zwei Phasen auf, die i. w. den in 2.1 beschriebenen Optimierungen entsprechen. Seine Parameter sind neben den Eingaberelationen R_1 und R_2 die Anzahl M der Tupel, die

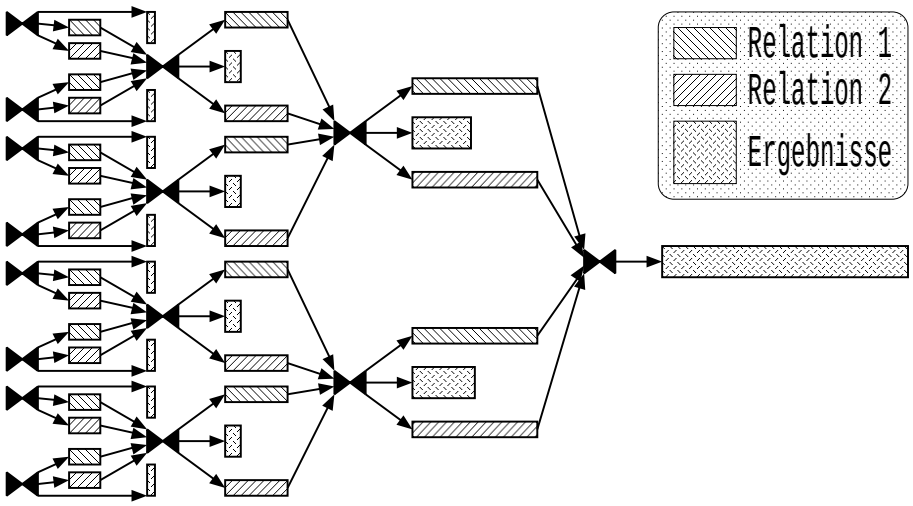


Abbildung 2.2: Progressive-Merge-Join

maximal in den Hauptspeicher paßt, und die maximale Anzahl an Hauptspeicherseiten F, die zum Lesen vom Externspeicher verwendet werden können. Diese nennt man auch *Fan-In*.

2.2.1.1 Phase 1: Join während Runerzeugung

Die erste Phase ist für das Erzeugen der initialen Runs für beide Relationen und die Produktion erster früher Ergebnisse zuständig. Dazu wird bis zur vollständigen Konsumierung beider Relationen R_1 und R_2 jeweils eine Teilmenge beider gelesen und zu Folgen $\hat{R}'_1$ bzw. $\hat{R}'_2$ sortiert. Bevor jene nun auf den Externspeicher ausgelagert werden, wird zur Produktion früher Ergebnisse ihr Join durch die Funktion `earlyJoinInitialRuns` berechnet, die in Abschnitt 2.2.3 erläutert wird. Die Menge Q dient zur Verwaltung der Runs, die in ihr paarweise entsprechend ihrer gemeinsamen Verarbeitung in Phase 1 gespeichert werden.

2.2.1.2 Phase 2: Join während Merge

In der zweiten Phase werden nun die Runpaare aus Q nach und nach gemerget. Dazu wird jeweils eine Teilmenge $\hat{Q}$ davon so gewählt, daß für jeden Run eine der F verfügbaren Seiten im Hauptspeicher zum Lesen vom Externspeicher zur Verfügung gestellt werden kann. Diese wird dann von der Funktion `earlyJoinMergedRuns`, wie in Abschnitt 2.2.4 beschrieben, zu frühen Joinergebnissen und einem Paar von Runs, das wieder in Q registriert wird,

Algorithmus 2.1 Progressive-Merge-Join

IN:	R_1, R_2	die Eingaberelationen
	M	maximale Anzahl von Tupeln, die in den Hauptspeicher passen
	F	maximale Seitenzahl zum Lesen der Runs
OUT:	*Result*	die Joinergebnisse

1: Let $Result := \emptyset$, $Q := \emptyset$;
2: {Phase 1: Join während Runerzeugung}
3: **while** $R_1 \neq \emptyset \vee R_2 \neq \emptyset$ **do**
4: Let $\hat{R}_1 \subset R_1$ and $\hat{R}_2 \subset R_2$ where $|\hat{R}_1| + |\hat{R}_2| \leq M$;
5: Let $R_1 := R_1 \backslash \hat{R}_1$, $R_2 := R_2 \backslash \hat{R}_2$;
6: Sort $\hat{R}_1$ into sequence $\hat{R}'_1$, Sort $\hat{R}_2$ into sequence $\hat{R}'_2$;
7: Let $Result := Result \cup earlyJoinInitialRuns(\hat{R}'_1, \hat{R}'_2)$;
8: Write $\hat{R}'_1$ and $\hat{R}'_2$ to external memory;
9: Let $Q := Q \cup \{(\hat{R}'_1, \hat{R}'_2)\}$
10: **end while**
11: {Phase 2: Join während Merge}
12: **while** $|Q| > 1$ **do**
13: Let $\hat{Q} \subset Q$, where $|\hat{Q}| \leq F/2$;
14: Let $(\hat{R}'_1, \hat{R}'_2)$ be a tuple of empty sequences;
15: Let $Result := Result \cup earlyJoinMergedRuns(\hat{Q}, (\hat{R}'_1, \hat{R}'_2))$;
16: Write $\hat{R}'_1$ and $\hat{R}'_2$ to external memory;
17: Let $Q := (Q \backslash \hat{Q}) \cup \{(\hat{R}'_1, \hat{R}'_2)\}$;
18: **end while**

verarbeitet werden. Ist nur noch ein Runpaar übrig, so wurde im vorangegangenen Schritt die Sortierung beider Relationen abgeschlossen und deren Join fertig berechnet.

Werden die komplett sortierten Eingaberelationen nicht als Ergebnis des Algorithmus benötigt, so kann das Hinausschreiben auf den Externspeicher im Falle $\hat{Q} = Q$, also beim letzten Durchlauf der `while`-Schleife, entfallen.

2.2.2 SweepAreas

Teil der Aufgabe von `earlyJoinInitialRuns` und `earlyJoinMergedRuns` ist jeweils, den Join je einer sortierten Folge beider Relationen zu berechnen. Um dies für ein breites Spektrum von Joinarten zu ermöglichen, wird in [DSTW02b] ein allgemeines Konzept vorgestellt, dessen Grundlage sogenannte *SweepAreas* bilden, die in einer spezialisierten Form besonders zur

Datentyp 2.2 SweepArea

```
1: Predicate P_join, P_rm;
2: Let X := ∅;
3: Procedure insert(Element v) [
4:       Let X := X ∪ {v};
5: ];
6: Function query(Element v) [
7:       Let X := X \ {u | P_rm(u, v), u ∈ X};
8:       return {(u, v) | P_join(u, v), u ∈ X};
9: ];
```

Berechnung von Spatial-Joins (1.5.1.4) Verwendung finden [BSS98].

Eine SweepArea ist ein Zwischenspeicher für diejenigen Tupel einer Relation, die im Zuge der Sortierung bereits verarbeitet und somit auf den Externspeicher ausgelagert wurden, für deren potentielle Joinpartner dies jedoch noch nicht zwingend gilt. Zu jeder Relation wird nun eine solche SweepArea benutzt. Tupel der Relation, die bei deren Sortierung ausgelagert werden, werden in die zugehörige SweepArea mit Hilfe der Funktion `insert` eingefügt. Tupel der anderen Relation können mit der Funktion `query` ihre derzeit verfügbaren Joinpartner erfragen, also die, die mit ihnen das Joinprädikat P_{join} erfüllen. Diese Anfragen sorgen zudem dafür, daß alle Elemente aus der SweepArea entfernt werden, die in der weiteren Sortierfolge an keinen Joins mehr beteiligt sein können; sie erfüllen dann das Entfernungsprädikat P_{rm}. Dadurch wird verhindert, daß die SweepAreas immer größer werden.

2.2.3 Das Joinen initialer Runs

Aufgabe der Funktion `earlyJoinInitialRuns` ist die Berechnung des Joins zweier bereits sortierter Runs R'_1 und R'_2. Dazu wird zuerst für jede der Relationen eine leere SweepArea angelegt. Dann erfolgt die Traversierung der Folgen in Reihenfolge der gemeinsamen Ordnung. Zur Verarbeitung wird also jeweils das kleinere der beiden Tupel ausgewählt, die den Anfang der Folgen bilden. Dieses wird in die zu seiner Relation gehörige SweepArea eingefügt und befragt dann die SweepArea der anderen Relation nach Joinpartnern. Diese werden als Ergebnisse zurückgeliefert, und das nächste Tupel wird zur Verarbeitung ausgewählt.

Funktion 2.3 earlyJoinInitialRuns

IN: R'_1, R'_2 sortierte Eingabefolgen
OUT: $Result$ Joinergebnisse

```
1: Let Result := ∅;
2: Let S_1, S_2 be empty SweepAreas;
3: while R'_1 ≠ ∅ ∨ R'_2 ≠ ∅ do
4:    Let r_1 := First(R'_1) , r_2 := First(R'_2);
5:    if r_1 < r_2 then
6:       S_1.insert(r_1);
7:       Let Result := Result ∪ S_2.query(r_1);
8:       Let R'_1 := R'_1\{r_1};
9:    else
10:      S_2.insert(r_2);
11:      Let Result := Result ∪ S_1.query(r_2);
12:      Let R'_2 := R'_2\{r_2};
13:   end if
14: end while
```

2.2.4 Das Joinen vereinigter Runs

Die Aufgabe der Funktion `earlyJoinMergedRuns` entspricht i. w. der von `earlyJoinInitialRuns` bis auf den Unterschied, daß nun nicht mehr nur ein, sondern mehrere Runs $\{R'^1_1 \dots R'^n_1\}$ bzw. $\{R'^1_2 \dots R'^n_2\}$ pro Relation verarbeitet werden. Diese sollen im Zuge der Sortierung zu jeweils einem gemeinsamen zusammengefügt werden. Dazu erfolgt die Auswahl des kleinsten Elementes r_k nun also über alle diese Runs. Dieses ist bezüglich der Sortierfolge das nächste seiner Relation und wird an den zugehörigen Ausgaberun $\hat{R}'_k$ angehängt. r_k wird analog zu 2.2.3 als kleinstes unverarbeitetes Tupel beider Relationen nun auch zum Joinen verwendet.

Wichtig ist dabei zu bedenken, daß es sich bei den hier verarbeiteten Paaren (R'^i_1, R'^i_2) von Runs um Resultate vorangegangener Aufrufe von `earlyJoinInitialRuns` oder `earlyJoinMergedRuns` handelt, deren Joins also schon berechnet und als frühe Ergebnisse zurückgeliefert wurden. Damit diese Ergebnisse nicht nochmals geliefert werden, werden sie dem Ergebnis der Anfragen an die SweepAreas wieder entnommen.

Funktion 2.4 earlyJoinMergedRuns

IN:	$\{(R_1'^1, R_2'^1) \ldots (R_1'^m, R_2'^m)\}$	Paare sortierter Eingabefolgen
OUT:	$Result$	Joinergebnisse
	$(\hat{R}_1', \hat{R}_2')$	sortierte Ausgabefolgen

1: Let $Result := \emptyset$;
2: Let S_1, S_2 be empty SweepAreas;
3: **while** $\exists m : R_1'^m \neq \emptyset \vee R_2'^m \neq \emptyset$ **do**
4: Let $r_1 := First(R_1'^{m_1})$ where $r_1 = minimum\{First(R_1'^i)\}$;
5: Let $r_2 := First(R_2'^{m_2})$ where $r_2 = minimum\{First(R_2'^i)\}$;
6: **if** $r_1 < r_2$ **then**
7: $S_1.insert(r_1)$;
8: Let $Result := Result \cup (S_2.query(r_1) \setminus \{(r_1, t) \mid t \in R_2'^{m_1}\})$;
9: Let $R_1'^{m_1} := R_1'^{m_1} \backslash \{r_1\}$;
10: Let $\hat{R}_1' := \hat{R}_1 \cup \{r_1\}$;
11: **else**
12: $S_2.insert(r_2)$;
13: Let $Result := Result \cup (S_1.query(r_2) \setminus \{(t, r_2) \mid t \in R_1'^{m_2}\})$;
14: Let $R_2'^{m_2} := R_2'^{m_2} \backslash \{r_2\}$;
15: Let $\hat{R}_2' := \hat{R}_2 \cup \{r_2\}$;
16: **end if**
17: **end while**

2.3 Korrektheit des Algorithmus

Dieser Abschnitt dient dem Korrektheitsbeweis des Algorithmus PMJ. Auf eine abschließende Projektion (siehe 1.2.2) beim Join wird hier verzichtet, das Joinprädikat ist P_{join}, die Eingaberelationen sind R_1 und R_2. Für die Ergebnismenge $Result$ des Algorithmus muß also nach der Ausführung gelten:

$$Result = \sigma_{P_{join}}(R_1 \times R_2) = \{(r_1, r_2) \mid r_1 \in R_1, r_2 \in R_2,\ P_{join}(r_1, r_2)\} \quad (2.1)$$

Zunächst wird eine Bedingung formuliert, die für die Korrektheit des Algorithmus hinreichend ist (2.3.1), danach wird in drei Schritten bewiesen, daß nur richtige Ergebnisse berechnet werden (2.3.2), daß alle Ergebnisse berechnet werden (2.3.3) und daß kein Ergebnis doppelt ausgegeben wird (2.3.4).

Analog zu [DSTW02b] wird die vereinfachende Annahme getroffen, daß es sich um einen symmetrischen Join handelt, die Relationen also gegeneinander austauschbar sind.

2.3.1 Bedingung

Die Bedingung für die Korrektheit bezieht sich auf das Prädikat P_{rm}, welches das Entfernen von Tupeln aus den SweepAreas regelt. Sie bringt die einleuchtende Tatsache zum Ausdruck, daß kein Tupel ausgelagert werden darf, das anderenfalls später[1] noch zur Produktion von Joinergebnissen beitragen könnte:

$$\forall w \geq v : P_{rm}(u, v) \Rightarrow \neg P_{join}(u, w) \tag{2.2}$$

Dabei bedeute $P_{rm}(x, y)$, daß eine Anfrage des Tupels y an eine SweepArea das Tupel x aus dieser entfernt. Die Bedingung ist äquivalent zu

$$\forall v \leq w : P_{join}(u, w) \Rightarrow \neg P_{rm}(u, v) \tag{2.3}$$

In [DSTW02b] wird zusätzlich die Bedingung gestellt, daß ein Tupel nicht entfernt werden darf, wenn es auch später nicht entfernt werden dürfte:

$$\forall w \leq v : \neg P_{rm}(u, v) \Rightarrow \neg P_{rm}(u, w) \tag{2.4}$$

Diese kann zwar nicht aus 2.2 hergeleitet werden, wird aber, wie der folgende Beweis zeigt, für die Korrektheit des Algorithmus nicht benötigt.

2.3.2 Korrektheit der Ergebnisse

Die Ergebnismenge wird leer initialisiert und danach nur um Ergebnisse von `earlyJoinInitialRuns` und `earlyJoinMergedRuns` erweitert. Diese nun liefern nur Ergebnisse von Aufrufen von `query` der SweepAreas zurück, welche Tupel der einen Relation enthalten und mit Tupeln der anderen angefragt werden und dabei nur Paare zurückliefern, die dem Joinprädikat P_{join} genügen. Demnach gilt:

$$Result \subseteq \{(r_1, r_2) \mid r_1 \in R_1, r_2 \in R_2,\ P_{join}(r_1, r_2)\} \tag{2.5}$$

2.3.3 Vollständigkeit der Ergebnismenge

Entscheidend für die Vollständigkeit der Ergebnismenge ist, daß die Funktionen `earlyJoinInitialRuns` und `earlyJoinMergedRuns` jeweils den Join der Teilmengen von R_1 und R_2 berechnen, die ihnen zur Verfügung stehen. Beide bearbeiten diese Tupel, wie in 2.2.3 und 2.2.4 erläutert, in der durch die gemeinsame Ordnung vorgegebenen Reihenfolge. Betrachtet man nun zwei

[1] „später“ bezieht sich hier auf die Bearbeitungsreihenfolge, die von der Ordnung auf den Relationen bestimmt wird.

solche Tupel a und b mit $P_{join}(a, b)$ und o. B. d. A.[2] $a \leq b$, so wird a zuerst gewählt und in die zu seiner Relation gehörige SweepArea eingefügt, wo es so lange verbleibt, bis ein Tupel c mit $P_{rm}(a, c)$ diese anfragt und dabei a entfernt. Wenn b also die SweepArea anfragt, so wird (a, b) genau dann nicht als Ergebnis zurückgeliefert, wenn c vor b verarbeitet wurde, wozu $c \leq b$ gelten müßte, was aber gar nicht sein kann, da es wegen $P_{join}(a, b)$ im Widerspruch zu Bedingung 2.2 stünde. Also wird (a, b) als Ergebnis erkannt.

Die beiden Relationen werden nun zunächst zu initialen Runs sortiert und dann nach und nach zusammengeführt, bis beim letzten Aufruf von `earlyJoinMergedRuns` jeweils ein Run pro Relation erzeugt wird, der alle derer Tupel enthält. Demnach treffen also spätestens dann alle Kombinationen (a, b) zusammen, werden als Joinergebnis erkannt und in *Result* eingefügt. Dies erfolgt nur dann nicht, wenn a und b aus den Runs nur eines Runpaars in Q kommen. Dann ist (a, b) aber schon vorher als Ergebnis eines Aufrufs entweder von `earlyJoinMergedRuns` oder `earlyJoinInitialRuns` erzeugt worden. Dies folgt per struktureller Induktion über den Aufbau des Merge-Baumes, wobei der zweite Fall den Induktionsanfang bildet, da in `earlyJoinInitialRuns` alle Resultate von `query` Teil von `Result` werden. Somit folgt

$$Result \supseteq \{(r_1, r_2) \mid r_1 \in R_1, r_2 \in R_2,\ P_{join}(r_1, r_2)\} \tag{2.6}$$

2.3.4 Einmaligkeit der Ergebnisse

Es bleibt zu zeigen, daß jedes Ergebnis (a, b) nur genau einmal in `Result` eingefügt wird. Wie in 2.3.3 gezeigt, geschieht dies mindestens beim ersten Aufeinandertreffen der zugehörigen Runs, da das Ergebnis dann produziert und nicht aussortiert wird. Danach gehören a und b aber zu einem Runpaar aus Q, was sich durch weitere Merges nicht mehr ändert. Daher wird das Ergebnis in der Folge in `earlyJoinMergedRuns` immer aus der Ergebnismenge von `query` aussortiert.

Da die Eingaberelationen vom Algorithmus als Folgen gelesen werden, gilt die Einmaligkeit der Ergebnistupel natürlich nur dann, wenn die beteiligten Tupel auch nur einmal gelesen wurden. Vorangegangene physische Operatoren müssen daher ebenfalls die Eigenschaft haben, Tupel nur exakt einmal zu produzieren.

Desweiteren sei darauf hingewiesen, daß die in der allgemeineren Joindefinition erlaubte nachfolgende Anwendung einer Funktion, insbesondere einer Projektion, die Einmaligkeit zerstören kann, wenn sie nicht injektiv ist. Die

[2] aufgrund der Symmetrie möglich

Herausprojektion doppelt auftretender Attribute beim natürlichen Verbund als wichtiges Beispiel ist jedoch bijektiv.

2.4 Skewbehandlung

Die SweepAreas als Datenstrukturen, die im Hauptspeicher gehalten werden, benötigen dort natürlich auch Speicherplatz, was in den bisherigen Ausführungen hier noch nicht berücksichtigt wurde. Will man den dafür nötigen Speicherbedarf ermitteln, so stößt man auf ein grundlegendes Problem, wenn man als Extrembeispiel einen Join mit Joinprädikat $F_\top := True$ betrachtet[3]. Demnach joint jedes Tupel mit allen der anderen Relation, weshalb es nie möglich sein kann, Tupel aus den SweepAreas zu entfernen (Bedingung 2.3). Da in `earlyJoinMergedRuns` die beteiligten Runs zusammen mehr als den verfügbaren Hauptspeicher einnehmen (da sonst ein externer Join unnötig wäre), übersteigt der Speicherbedarf der SweepAreas irgendwann die Speichergröße. Dieses Problem betrifft somit insbesondere auch jegliches Verfahren, das einen bestimmten Anteil des Speichers für die SweepAreas reserviert.

Dennoch muß man immer wieder entscheiden, wieviel Speicherplatz den SweepAreas zur Verfügung gestellt wird. Bei der Produktion initialer Runs, insbesondere beim ersten Paar, ist man mit der Hälfte des Speichers auf der sicheren Seite, da so die beiden Runs nochmals komplett in die SweepAreas passen. Sobald ein Selektivitätsschätzer verfügbar ist, kann man mit dessen Hilfe für alle weiteren Schritte den Speicherbedarf für die SweepAreas abschätzen und so auch bei Joins höherer Selektivität ein Überlaufen des Speichers verhindern.

Leider löst dies immer noch nicht das Problem des sogenannten Skew (siehe Seite 19), bei dem nicht zwingend die Selektivität hoch ist, sondern ggf. nur einzelne Tupel sehr viele Joinpartner haben. Dies erschwert zum einen die Selektivitätsschätzung, zum anderen können alleine die Joinpartner dieses einen Tupels mehr als den gesamten Hauptspeicher benötigen. Dies zeigt nun die Notwendigkeit, in solchen Fällen einen Teil der SweepArea auf den Externspeicher auszulagern. Da jedes anfragende Tupel aber ggf. mit jedem in der SweepArea joinen kann, muß der ausgelagerte Teil eventuell bei jeder Anfrage neu gelesen werden. Das Verfahren entartet damit zu einer Art Ripple-Join; durch das Sammeln von Anfragetupeln kann man es zu einem Block-Ripple-Join umformen. Weil dieses Problem auch im Joinschritt eines gewöhnlichen Sort-Merge-Joins auftritt, wird es z. B. in [LGS02] ausführlich behandelt.

[3] Dies bedeutet eine Selektivität des Joins von 1.

Beim PMJ tritt das Problem überlaufender SweepAreas aber ggf. schon beim frühen Joinen von Runs auf, was beim Auslagern der SweepAreas echten Mehrbedarf an I/O bedeutet. In diesem Fall kann man auch versuchen, die Problematik durch Umverteilung der Hauptspeicheranteile anzugehen. Man bricht das Verarbeiten einiger Runs ab und stellt deren ungelesenen Rest als neue Runs für die spätere Verarbeitung zurück. Die freigewordenen Seiten können für die SweepAreas benutzt werden. Da damit der Schätzer statistisch wertlos wird, bietet sich als Radikallösung an, die Produktion früher Ergebnisse für diesen Schritt komplett abzubrechen.

2.5 Implementierungsaspekte

In diesem Abschnitt wird auf spezielle Aspekte der Implementierung eingegangen und u. a. erläutert, wie diese in der Referenzimplementierung zu [Dit02] gelöst sind.

2.5.1 Early Joins

Die Funktionen `earlyJoinInitialRuns` und `earlyJoinMergedRuns` weisen starke Ähnlichkeiten auf, die man zu einer gemeinsamen Implementierung nutzen kann. `earlyJoinInitialRuns` stellt dann einen Spezialfall mit nur einem Run pro Relation und ohne Duplikateliminierung dar.

Die gemeinsame Implementierung muß nun jeweils das kleinste Element bezüglich der gemeinsamen Ordnung O über alle Runs bestimmen. Eine solche Auswahl erledigt man üblicherweise mit Hilfe eines MinHeaps. Zur Verarbeitung muß aber rekonstruierbar bleiben, aus welchem Run das Tupel stammt. Die Referenzimplementierung fügt daher in den Heap Elemente einer Datenstruktur ein, die ein Paar von Runs aus Q verwaltet, intern deren Minimum ermittelt und in der Heap-Sortierung wie dieses auftritt. Bei Entnahme der Heapwurzel kann somit rekonstruiert werden, welcher Run das globale Minimum enthält.

2.5.2 Überlappendes I/O

Das Herausschreiben der Tupel auf den Externspeicher sollte im Gegensatz zum Pseudocode natürlich möglichst überlappend mit der Verarbeitung in den Funktionen geschehen; im Falle des Mergens ist dies sogar zwingend erforderlich, da mehrere Runs nicht komplett im Hauptspeicher abgelegt werden können. Daher wird jedes Tupel nach Einfügen in seine und Anfragen der anderen SweepArea sofort dem Schreibpuffer für den Externspeicher zugeführt.

2.5.3 Duplikateliminierung

Ein besonderes Problem stellt die Eliminierung der Duplikate dar. In der Funktion `earlyJoinMergedRuns` müssen aus den Resultaten der Anfragen `query` diejenigen herausgenommen werden, bei denen das anfragende Tupel aus demselben Runpaar aus Q stammt wie das Tupel, mit dem es in der SweepArea gejoint wurde. Die Referenzimplementierung verwendet zu diesem Zwecke mehrere SweepAreas pro Relation, und zwar je eine pro Run. Jedes Tupel wird nun in die zu seinem Run gehörige SweepArea eingefügt und fragt dann alle SweepAreas der anderen Relation an, mit Ausnahme derjenigen mit der eigenen Runnummer, so daß die Erzeugung der Duplikate genau entfällt. Der Nachteil dieser Methode ist jedoch, daß die Zahl der SweepAreas und die der Anfragen an diese um den Faktor F ansteigt. Aus Effizienzgründen wird daher zusätzlich sichergestellt, daß leere SweepAreas erst gar nicht angefragt werden.

2.5.4 Kleinhalten der SweepAreas

Beim Anfragen einer SweepArea werden zunächst diejenigen Tupel aus dieser entfernt, die mit dem anfragenden Tupel das Entfernungsprädikat P_{rm} erfüllen. Genauso kann man aufgrund der Symmetrie aber auch mit den Tupeln verfahren, die eingefügt werden. Diese können ebenso diejenigen Tupel aus der zur eigenen Relation gehörigen SweepArea löschen, die keine Joinpartner unter den in der Ordnung hinter dem eingefügten Tupel stehenden finden können. Daher entfernt die Referenzimplementierung auch beim Einfügen eines Tupels t in die SweepArea S diejenigen Tupel s aus S, für die $P_{rm}(s,t)$ gilt.

2.6 Schätzer

Um die Selektivität einer Selektion $\sigma_F(R)$ zu schätzen, zieht man eine repräsentative Stichprobe $\tilde{R} \subset R$ und bestimmt die Selektivität von $\sigma_F(\tilde{R})$. Genauso könnte man nun beim Join vorgehen wollen. Um eine Stichprobe des kartesischen Produktes $R_1 \times R_2$ zu erhalten, genügt es jedoch leider nicht, zwei Stichproben $\tilde{R}_1 \subset R_1$ und $\tilde{R}_2 \subset R_2$ mit $|\tilde{R}_i| =: g_i$ zu ziehen und deren Kreuzprodukt $K := \tilde{R}_1 \times \tilde{R}_2$ zu bilden, denn dies stellt i. a. keine Stichprobe von $R_1 \times R_2$ dar, weil Korrelationen zwischen den Elementen bestehen können:

$$\check{K} := \{(a_1, b_1), (a_1, b_2), (a_2, b_1)\} \quad \subset \quad K \Rightarrow \{a_2, b_2\} \subset K$$

$\check{K}$ ist zwar eine dreielementige Stichprobe von $R_1 \times R_2$, kann aber nicht als kartesisches Produkt zweier Stichproben gebildet werden (siehe auch [AGPR99]).

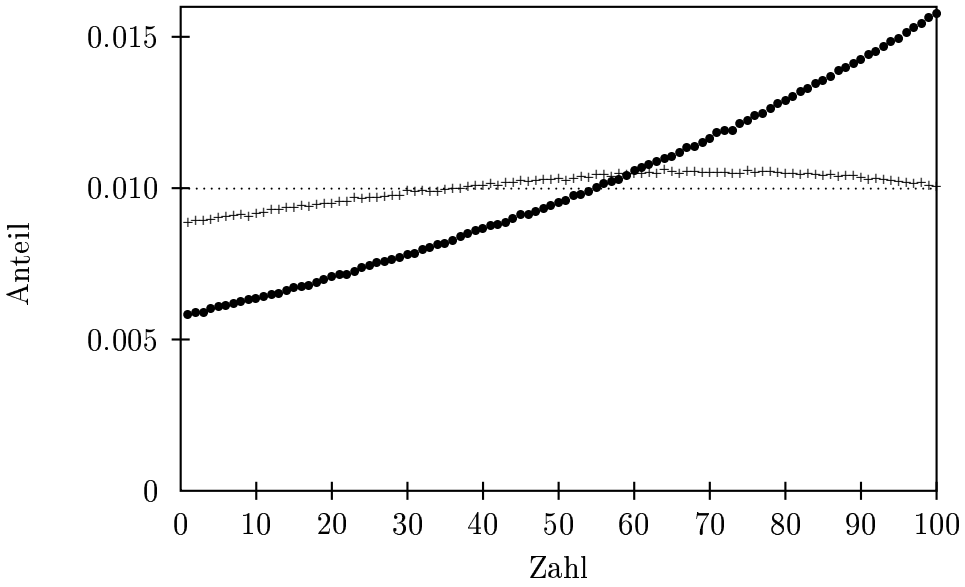
Beim PMJ wird allerdings in jedem Funktionsaufruf der Join über den zugrundeliegenden Runs, also dem Kreuzprodukt zweier Stichproben der Eingaberelationen, berechnet. Zwar kann man aus diesen Stichproben eine Stichprobe S des Kreuzproduktes generieren, doch benötigt man dazu die Angabe über die Größen der kompletten Relationen R_i. S hätte zudem etwa die Kardinalität der kleineren Stichprobe $|\tilde{R}_j|$, im Gegensatz zum kartesischen Produkt der Stichproben, dessen Größe $g_1 \cdot g_2$ ist.

Natürlich soll in den Selektivitätsschätzer möglichst die gesamte verfügbare Information einfließen, und die besteht beim PMJ aus der Kardinalität der Joins von Stichproben. Nun gilt immerhin für ein Element $e \in R_1 \times R_2$, daß es mit gleicher Wahrscheinlichkeit auftritt, wenn man ein Element aus $R_1 \times R_2$ oder eines aus dem kartesischen Produkt zweier Stichproben zieht. Erfüllt ein solches Element e mit einer Wahrscheinlichkeit p das Joinprädikat, so gilt für eine Menge $E \subset R_1 \times R_2$, daß der Erwartungswert für die Anzahl darin enthaltener Resultate des Joins $\sum_E p = |E| \cdot p$ ist. p entspricht jedoch der Selektivität des Joins, womit $\frac{|\{e \in E \mid F(e)\}|}{|E|}$ ein erwartungstreuer Schätzer für diese ist.

Einen solchen erwartungstreuen Selektivitätsschätzer kann man nun aus dem Ergebnis des ersten Aufrufs von `earlyJoinInitialRuns` berechnen, nämlich als Quotient aus der Zahl der dabei erzeugten Ergebnisse und der Kardinalität des dabei betrachteten kartesischen Stichprobenproduktes. Wegen der Additivität des Erwartungswertes kann man bei weiteren Aufrufen jeweils die neuen Ergebnisse bzw. den dabei durchsuchten Bereich hinzunehmen und damit den Schätzer verbessern.

2.6.1 Auswirkungen

In 1.5.2.4.2 wurde mit Replacement Selection ein Verfahren vorgestellt, mit dessen Hilfe sich die Zahl initialer Runs für den Sort-Merge-Join im Mittel halbieren läßt. Natürlich würde man diese Optimierung gerne auch beim PMJ einsetzen. Betrachtet man jedoch das erste Paar der dabei erzeugten Runs, so enthält dies mit höherer Wahrscheinlichkeit Tupel, die in der Ordnung hinten stehen, als solche, die bei der Sortierung früh erreicht werden. Dies resultiert aus der Tatsache, daß die nachgeladenen Tupel nur dann noch den ersten Runs zugeordnet werden können, wenn noch kein größeres Tupel hinausgeschrieben wurde. Daher stellt das erste Runpaar nicht einmal mehr Stichproben der Eingaberelationen dar und kann somit nicht für den zuvor



Verteilung der Zahlen $1, \ldots, 100$ im ersten ($\cdot$) und zweiten ($+$) Run bei Anwendung von Replacement Selection auf gleichverteilte Eingabe

Abbildung 2.3: Auswirkungen von Replacement Selection

beschriebenen Schätzer verwendet werden. Abbildung 2.3 zeigt das Problem der ungleichmäßigen Verteilung.

2.7 Effizienz

Für die Effizienz eines externen Joinverfahrens ist i. w. dessen I/O-Verhalten entscheidend, da Externspeicherzugriffe in heutigen Systemen im Vergleich zu Hauptspeicher oder gar Cacheoperationen extrem zeitaufwendig sind. Beim ursprünglichen Sort-Merge-Join 1.5.2.4 fällt dabei das externe Sortieren beider Relationen an, gefolgt vom Einlesen der beiden sortierten Runs. Wie in 1.5.2.4.2 erläutert, kann man davon mittels Join-During-Merge im optimalen Fall ein komplettes Lesen und Schreiben der Relationen einsparen. An diesem Verfahren muß sich der PMJ nun messen lassen.

Für N Eingabetupel, eine Seitenkapazität von B Tupeln, eine Speicherkapazität von M Tupeln und einen Speicherbedarf von P Tupeln für die SweepAreas werden in [DSTW02b] unter der Annahme $B|M$ und $B|N$ und der Voraussetzung, daß die SweepAreas nie die Größe von P Tupeln über-

schreiten, die I/O-Kosten des PMJ angegeben:

$$\Theta\left(\frac{N}{B} \cdot log_{\frac{M-P}{B}} \frac{N}{B}\right)$$

Dieses asymptotische Verhalten entspricht den Kosten für das externe Sortieren, also denen der anderen sortierbasierten externen Joinverfahren. Diese Aussage schließt jedoch nicht aus, daß der PMJ den I/O-Bedarf anderer Verfahren übertrifft, z. B. um einen konstanten Faktor. Daher erfolgt hier eine praxisbezogene Diskussion des Mehrbedarfs.

Statt zweier getrennter externer Sortierbäume wird beim PMJ nur einer aufgebaut. Bei der Generierung initialer Runs können jetzt zwei statt einem Run pro verarbeitetem Hauptspeicherinhalt entstehen. Zusätzlich muß auf die Verwendung von Replacement Selection verzichtet werden (2.6.1), wodurch die mittlere Runlänge sich nochmals halbiert [Knu98]. Insgesamt können bei der Rungenerierung also viermal mehr Runs entstehen, als dies beim klassischen Sort-Merge-Join mit Replacement Selection der Fall wäre.

In den Merge-Schritten werden die Relationen wieder verzahnt verarbeitet, wodurch sie sich den Fan-In teilen müssen und der Fan-In der jeweiligen Merge-Bäume nur noch $F/2$ ist. Die Höhe eines Merge-Baumes für das externe Sortieren ist für eine Zahl von I initialen Runs der Relation durch $\lceil log_F I \rceil$ gegeben. Bei halbem Fan-In und viermal mehr Runs ergibt sich eine Höhe von $\lceil log_{F/2}(4 \cdot I) \rceil$.

In realistischen Szenarien wird für einen externen Sortieralgorithmus nicht nur eine sehr geringe Zahl von Seiten zur Verfügung stehen, da dies die Effizienz katastrophal verringern würde. Daher darf man davon ausgehen, daß sich die Baumhöhen um maximal 1 unterscheiden, was je einmal zusätzliches Lesen und Schreiben der entsprechenden Relation bedeutet. Dies bedeutet aber ggf. einen erheblichen Mehraufwand, da $F > I$, also $\lceil log_F I \rceil = 1$, in heutigen Systemen als realistisch, wenn nicht sogar als der Regelfall anzusehen ist.

Die Problematik wird dadurch relativiert, daß beim Einsatz von Join-During-Merge (1.5.2.4.2) im letzten Schritt auch beim Sort-Merge-Join die Einsparung des je einmaligen Schreibens und Lesens der Relationen nur durch Verteilen des Fan-Ins auf beide Relationen erreicht wird, bei einer Baumhöhe von 1 im alten Verfahren also für den Vergleich zum PMJ nur die Vervierfachung der Zahl der initialen Runs relevant bleibt. Im Falle $I \leq F < 4I$ erfolgt beim PMJ trotzdem eine Verdopplung des I/O-Aufwandes gegenüber dem klassischen Verfahren, was natürlich vermieden werden sollte. Techniken dazu werden in 4.3.1 vorgestellt.

2.8 Anwendungsgebiete

2.8.1 Vergleich zu anderen Verfahren

Die Frage nach den mit PMJ berechenbaren Arten von Joins kann man zunächst durch Vergleiche mit anderen Joinverfahren beantworten:

2.8.1.1 Sort-Merge-Join

Der letzte Schritt im Merge-Baum beim PMJ entspricht bezüglich der als Joinergebnisse identifizierten Tupel dem des reinen Sort-Merge-Joins, man kann also dieselben Joins wie dieser berechnen.

2.8.1.2 Nested-Loops-Join

Nimmt man auf beiden Relationen die triviale Ordnung an, die alle Tupel als gleich ansieht, dann wäre die zufällige Anordnung der Relationen bereits eine sortierte Folge, und im PMJ würde sinnfrei sortiert und letztendlich der Join dieser initialen Runs berechnet. Die Auslagerungsproblematik, daß diese nicht in den Hauptspeicher passen, würde dabei auf die Jointechnik der SweepAreas verschoben. Dort könnte man das Entfernungsprädikat P_{rm} als $False$ ansetzen, womit keine Tupel entfernt werden. Damit würde der PMJ dann alle Elemente des Kreuzproduktes bilden und gegen das Joinprädikat testen, womit er zur Berechnung jeder Art von Join in der Lage wäre.

2.8.1.3 Hashbasierte Joinverfahren

Gibt es zur Berechnung eines Joins ein geeignetes hashbasiertes Joinverfahren, so kann man die Ordnung auf den Tupeln definieren, indem man jedem Tupel seine Hashpartitionsnummer zuweist und darauf die totale Ordnung auf $\mathbb{N}$ verwendet. Das Entfernungsprädikat P_{rm} würde dann jeweils alle Tupel mit kleinerer Partitionsnummer entfernen, so daß sich in jeder SweepArea jeweils nur die Tupel einer Partition gleichzeitig befinden könnten. Durch die zum Lesen der Runs benötigten Hauptspeicherseiten wäre dieses Verfahren im Vergleich zu speziellen hashbasierten Joinverfahren jedoch weniger effizient.

2.8.2 Anpassung auf spezielle Arten

Wie der voranstehende Abschnitt schon zeigt, ist für die Anpassung des PMJ an ein bestimmtes Joinprädikat $F = P_{join}$ die geeignete Wahl der Ordnung auf den Relationen und die des Entfernungsprädikats P_{rm} sehr wichtig. Als

dritter wichtiger Faktor kommt in vielen Fällen die geeignete Wahl der internen Datenstruktur für die SweepAreas hinzu.

Die hier diskutierten Beispiele sind [DSTW02b] entnommen und betrachten dementsprechend nur symmetrische Joins, was auch der Beschreibung des Algorithmus in diesem Kapitel entspricht.

2.8.2.1 Equi-Join

Beim Equi-Join bietet sich als Ordnung kanonisch die auf dem Attribut A an, auf dem Gleichheit verlangt wird; für den allgemeineren Fall der Gleichheit auf mehreren Attributen verwendet man entsprechend eine darauf aufgebaute lexikographische Ordnung.

Bei der Wahl des Aufbaus der SweepArea kann man hier eine spezielle Eigenschaft des Equi-Joins ausnutzen. Um Bedingung 2.2 zu genügen, kann zu $P_{join}(u, v) = (u.A = v.A)$ die Bedingung $P_{rm}(u, v) = (u.A < v.A)$ gewählt werden. Da die SweepArea aber beim Test gegen v aufgrund der sortierten Verarbeitung nur Elemente kleiner oder gleich v enthalten kann, ist diese Bedingung äquivalent zu $P'_{rm}(u, v) = (u.A \neq v.A)$ mit der Besonderheit $\neg P'_{rm}(u, v) \Rightarrow P_{join}(u, v)$. Die SweepArea enthält wegen 2.5.4 immer nur eine Menge bezüglich der Ordnung gleicher Elemente, und bei jeder Anfrage mit einem Tupel v werden daher entweder alle Tupel aus der SweepArea entfernt und keines zurückgeliefert oder keines entfernt und alle zurückgeliefert. Zur Unterscheidung der Fälle genügt ein Vergleich mit einem Repräsentanten der Elemente in der SweepArea; zu deren Speicherung kann eine einfache Liste verwendet werden.

2.8.2.2 Band-Join

Beim Band-Join dürfen die Tupel bezüglich des Wertes eines Attributes A nur um einen Betrag ε voneinander abweichen, so daß man sie, um sie entsprechend zusammenzubringen, entsprechend der Ordnung auf diesem Attribut sortiert. Aus der SweepArea entfernt werden dürfen Tupel, die gegenüber dem anfragenden Tupel den Abstand ε überschreiten, da später anfragende noch größere Tupel aufgrund der Dreiecksungleichung nur noch größere Abstände aufweisen und auch nicht joinen (damit ist wieder Bedingung 2.2 erfüllt). Alle nicht entfernten Tupel sind wiederum aufgrund der Dreiecksungleichung Joinergebnisse.

Die Tupel in der SweepArea können somit in der Reihenfolge entfernt werden, in der sie eingefügt wurden; als Speicherstruktur eignet sich also eine Queue. Effizienter ist eine Datenstruktur, bei der dasjenige Tupel in der Ordnung, bis zu dem entfernt und wonach ausgegeben wird, schneller

gefunden werden kann. Da sortiert eingefügt wird, ist dazu z. B. eine Skipliste geeignet.

2.8.2.3 Temporal-Join

Beim Temporal-Join enthalten beide Relationenschemata eine Darstellung eines Intervalls $I = [min, max]$. Tupel joinen, wenn sich ihre Intervalle schneiden, also $P_{join}(u, v) := (u.I \cap v.I \neq \emptyset)$. Sortiert man diese nun nach ihrer niedrigeren Komponente, so werden sie in die SweepArea eingefügt, sobald ihr Bereich von der Sortierung erreicht wird. Entfernt werden können sie wieder, sobald ein anfragendes Tupel erst hinter ihnen beginnt, da das dann auch alle folgenden tun werden (Bedingung 2.2), also $P_{rm}(u, v) := (u.I.max < v.I.min)$.

Die Menge der Tupel in der SweepArea unterteilt sich dabei wieder in zwei Mengen, von denen die eine entfernt wird und der Rest mit dem anfragenden Tupel joint. Wie beim Band-Join muß die entsprechende Aufteilung gefunden werden, die sich nach den Endpunkten der Intervalle richtet, welche allerdings sortiert nach ihren Anfängen eingefügt werden. Wegen der einfachen Entfernbarkeit des Anfangsbereiches könnte man auch dazu eine Skipliste verwenden.

2.8.2.4 Spatial-Join

Sweep-Verfahren spielen insbesondere im Bereich der Geodatenbanksysteme [BSS98] eine wichtige Rolle, da dort Punkte in der Ebene betrachtet werden, auf denen man keine geeignete Ordnung definieren kann, die den euklidischen Abstand widerspiegelt, also in der aus $a \leq b < c$ folgt, daß $d(a, c) > d(a, b)$. Daher entwickelte man dort Plane-Sweep-Verfahren, die eine Ordnung in einer Dimension benutzen, um den Algorithmus zu steuern (Event Points), und bezüglich der anderen Dimension mit einer speziellen Indexstruktur operieren.

Im Falle des Spatial-Joins enthalten beide Relationen die Darstellung eines Rechtecks R in Form eines Paares von Intervallen (I_x, I_y), das Joinprädikat fordert den Schnitt der Rechtecke, also $P_{join}(u, v) := (u.R \cap v.R \neq \emptyset)$. Die Algorithmensteuerung beim PMJ folgt der Sortierung, sortiert wird also anhand des Intervalls in der einen Dimension, analog zum Temporal-Join bezüglich der niedrigeren Komponente $I_x.min$. Das Entfernen kann demnach wieder am Ende des Intervalls erfolgen, also mit $P_{rm}(u, v) := (u.I_x.max < v.I_x.min)$. Das Intervall I_y spielt also weder für die Sortierung, noch für P_{rm} eine Rolle. Im Gegensatz zu den vorherigen Beispielen verbleiben damit nun auch nach dem Entfernen mittels P_{rm} noch Tupel in den SweepAreas, die

nicht mit dem anfragenden Tupel joinen. Dafür müssen sie nämlich auch bezüglich I_y das anfragende Rechteck schneiden. Die SweepArea muß die Rechtecke daher so speichern, daß effektive Schnittanfragen bezüglich I_y und effektives Löschen bezüglich $I_x.max$ möglich ist. Siehe dazu [Dit02].

Zusammenfassung

Der Progressive-Merge-Join (PMJ) [DSTW02b] stellt eine Weiterentwicklung des Sort-Merge-Joins dar, die frühe Ergebnisse und Schätzer ermöglicht. Dazu werden die Eingaberelationen nicht mehr getrennt sortiert, sondern verzahnt, um zwischenzeitlich den Join sortierter Teile zu berechnen und auszugeben. Mit Hilfe spezieller Speicherstrukturen (SweepAreas) ist dieser sortierbasierte Algorithmus auf ein breiteres Spektrum von Jointypen anwendbar als bisherige Joinverfahren, die frühe Ergebnisse liefern.

Kapitel 3

Mehrdimensionaler Progressive-Merge-Join

Übersicht

Die Struktur dieses Kapitels lehnt sich an die von Kapitel 2 an, da hier eine Weiterentwicklung namens *Mehrdimensionalen Progressive-Merge-Join* des dort behandelten Algorithmus PMJ vorgestellt wird. Zunächst werden in 3.1 die Ziele der Veränderungen motiviert, bevor dann der neue Algorithmus in 3.2 vorgestellt und dessen Korrektheit in 3.3 bewiesen wird. In 3.4 und 3.5 werden wieder Implementierungsaspekte und der Schätzer besprochen, und abschließend wird in 3.6 auf die erweiterten Einsatzgebiete gegenüber dem PMJ eingegangen.

3.1 Zielsetzung

Wie in 2.8 erläutert, eignet sich der PMJ zur Berechnung eines breiten Spektrums von verschiedenen Joinarten. Dennoch gibt es solche, für die zwar sortierbasierte Joinverfahren geeignet sind, auf die der PMJ jedoch in der in 2 vorgestellten Form nicht oder nur mit Einschränkungen anwendbar ist. Diese Fälle werden hier erläutert und dann diejenigen identifiziert, für die in diesem Kapitel eine erweiterte Variante des PMJ vorgestellt wird.

3.1.1 Einschränkungen des PMJ

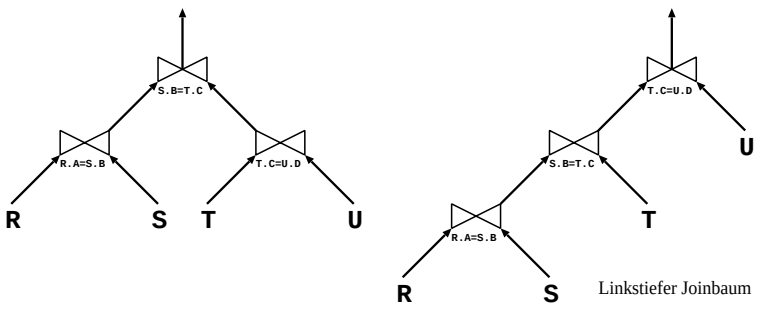
3.1.1.1 PMJ ist symmetrisch

Der PMJ benutzt je nach Joinart verschiedene SweepAreas, Joinprädikate P_{join} und Entfernungsprädikate P_{rm}. Allerdings wird die gewählte SweepArea mit P_{rm} dann für beide Relationen verwendet. Daher können damit nur Joins berechnet werden, für die sich die Relationen bezüglich des Joinverhaltens symmetrisch verhalten, also mit $P_{join}(a, b) \Leftrightarrow P_{join}(b, a)$. Formal streng genommen bedeutet dies, daß sogar beide Relationen das gleiche Schema haben müssen, damit die Joinbedingung überhaupt symmetrisch ausgewertet werden kann. Offensichtlich beziehen sich die vorgestellten Joinarten aber immer auf bestimmte Attribute, z. B. diejenigen, die beim Equi-Join in beiden Relationen gleich sein müssen. Aber auch, wenn die Relationen für die Auswertung des Joins vom Joinprädikat asymmetrisch auf die relevanten Attribute projiziert werden, bleibt das Joinverhalten symmetrisch bezüglich dieser relevanten Attribute. Beim Temporal Join muß ein Intervall ein Intervall schneiden, beim Spatial-Join ein Rechteck ein Rechteck usw.

3.1.1.2 PMJ ist binär

Neben den binären (1.2.2) kann man allgemeiner auch mehrdimensionale Joins $\bowtie_F (R_1, \ldots, R_n)$ betrachten (1.2.3), die jedoch von vielen physischen Joinoperatoren nicht allgemein unterstützt werden. Eine Berechnung des mehrdimensionalen Joins durch mehrmalige Bildung des Kreuzproduktes und abschließende Selektion ist hier jedoch oft noch ineffizienter als im binären Fall, so daß der Einsatz optimierter physischer Operatoren geboten ist.

Eine Methode dazu ist, den mehrdimensionalen Join als Hintereinanderausführung mehrerer binärer Joins zu berechnen. Dazu wählt man aus der Vielzahl der Möglichkeiten, das Kreuzprodukt der Eingaberelationen zu bilden (die Kreuzproduktbildung ist assoziativ und in diesem Kontext, da die Reihenfolge der Attribute unerheblich ist und zudem später noch verändert werden kann, auch kommutativ), eine aus. Die binären Kreuzproduktoperatoren werden dabei soweit möglich mit einer Joinbedingung versehen, die als Teilbedingung an die dort beteiligten Relationen aus der Joinbedingung gewonnen werden kann und der Einschränkung der Größe der Zwischenergebnisse dient. Die Wahl geeigneter Reihenfolgen und Bedingungen ist Aufgabe des Optimierers [Ber02], kann nach verschiedenen Kriterien erfolgen sowie ohne Hinzunahme von Metainformationen oft nur sehr vage getroffen werden und stellt selber ein umfangreiches Forschungsgebiet dar [KS00], [MP01]. Wegen der Komplexität des Problems beschränken sich viele Systeme auf einen Teil des Ergebnisraumes und betrachten zum Beispiel nur sogenannte



$\bowtie_{R.A=S.B=T.C=U.D} (R, S, T, U)$

Abbildung 3.1: Verschiedene binäre Joinbäume

linkstiefe Joinbäume, bei denen in jeden Joinoperator auf einer Seite nur eine Relation eingeht. Ein Beispiel zeigt Abbildung 3.1.

Soll dieses Verfahren zur Berechnung eines mehrdimensionalen Joins nicht blockierend sein, so dürfen es auch die verwendeten binären physischen Operatoren nicht sein, und diese müssen verzahnt ausgeführt werden. Zur effektiven Berechnung z. B. früher Ergebnisse oder Schätzer müssen die Parameter der binären Operatoren zudem global optimiert werden. Z. B. hängt die Wahl der Eingaberaten bei Ripple-Joins von den vorstehenden Joins ab [HH99].

Beim Kaskadieren des binären PMJ tritt nun ein Problem auf. Damit sein Schätzer funktioniert, müssen die Eingaberelationen in zufälliger Reihenfolge gelesen werden (siehe 1.4). Die Ausgabe des sortierbasierten PMJ teilt sich jedoch in mehrere Abschnitte, die jeweils als Ergebnis der `earlyJoin`-Funktionen entstehen und nach dem Kriterium sortiert sind, nach dem es die Eingaberelationen wurden. Die Ergebnisse von `earlyJoinInitialRuns` stellen immerhin noch den Join zweier Stichproben dar und könnten mit einer Stichprobe einer dritten Relation gejoint werden. Allerdings könnten diese Zwischenergebnisse je nach Selektivität des ersten Joins zu groß werden.

Ein unmodifizierter PMJ könnte als Eingabe also nur dann die Ausgabe eines anderen Aufrufs des PMJ benutzen, wenn deren Sortierung statistisch völlig unkorreliert mit dem zweiten Join wäre. Dieser Fall kommt jedoch nicht nur eher selten vor, sondern kann insbesondere fast nie ohne zuvorige Berechnung des kompletten Joins erkannt werden. Somit ist eine Modifizierung des PMJ zur Berechnung mehrdimensionaler Joins geboten. Der einfachste Ansatz, die sortierte Eingabe in eine zufällige Reihenfolge zu durchmischen, scheitert daran, daß er blockierend ist, da auch das letzte Tupel bezüglich der Sortierung Teil der ersten Stichprobe werden können muß.

3.1.2 Motivation

3.1.2.1 Asymmetrie

Enthält eine Relation R_1 Punkte und eine andere R_2 Intervalle, so könnte man diese nach der Bedingung joinen, daß der Punkt $r_1 \in R_1$ im Intervall $r_2 \in R_2$ liegen muß. Dies stellt einen Spezialfall des Temporal-Join dar, bei dem alle Intervalle der ersten Relation die Länge 0 haben, und kann so auf jeden Fall mit dem PMJ berechnet werden. Dabei würde jedoch die Zusatzinformation verschenkt und unnötiger Aufwand betrieben. Betrachtet man eine SweepArea zu R_1, so enthält diese punktförmige Intervalle. Bei der Auswahl des nächsten Intervalles I bezüglich der Ordnung können nun zwei grundlegende Fälle auftreten. Kommt I aus R_1, so ist es entweder gleich den bereits in der SweepArea enthaltenen Punkt-Intervallen und wird eingefügt, oder liegt dahinter und löscht zunächst alle Punkte in der SweepArea. Kommt I hingegen aus R_2, so beginnt es entweder an dem Punkt, der gerade in der anderen SweepArea enthalten ist und schneidet somit alle Elemente darin, oder beginnt dahinter und entfernt somit alle Elemente aus der anderen SweepArea. Diese zu R_1 gehörige SweepArea zeigt somit das Verhalten einer SweepArea für den Equi-Join, die mit Punkten aus R_1 gefüllt (und dabei ggf. zuvor von diesen bereinigt) und den Anfangspunkten der Intervalle aus R_2 angefragt wird.

Dieses Beispiel zeigt, daß eine der beiden SweepAreas durch eine effektivere andere ersetzt werden könnte, womit jedoch zwei verschiedene Arten von SweepAreas bei einem Join im Einsatz wären. Man kann zwar jeden binären Join durch einen symmetrischen Join (die Kreuzproduktbildung) und eine nachfolgende Selektion ersetzen, doch i. a. ist für eine asymmetrische Joinbedingung P_{join} auch ein asymmetrisches Joinverfahren effektiver.

3.1.2.2 Multidimensionalität

In 3.1.1.2 wird aufgezeigt, warum der PMJ i. a. nicht zur Berechnung multidimensionaler Joins verwendbar ist. Auf der Suche nach einer geeigneten Erweiterung bietet sich ein Blick auf die elementare Idee des PMJ an: Im Gegensatz zum Sort-Merge-Join wird der Hauptspeicher in allen Schritten zwischen den beiden Relationen aufgeteilt. Dieses Prinzip läßt sich ohne weiteres auf mehr als zwei Relationen erweitern. Bei der Erzeugung initialer Runs können Teile aller r Relationen geladen und als sortierte Runs herausgeschrieben werden, beim Mergen ebenfalls Runs aus r Relationen zusammengefügt.

Das Ergebnis der Erweiterung soll aber wieder ein sortierbasiertes Joinverfahren sein. Die Wahrscheinlichkeit für Tupel, an einem Join beteiligt zu sein, sollte also wieder in Zusammenhang mit ihrer Stellung bezüglich

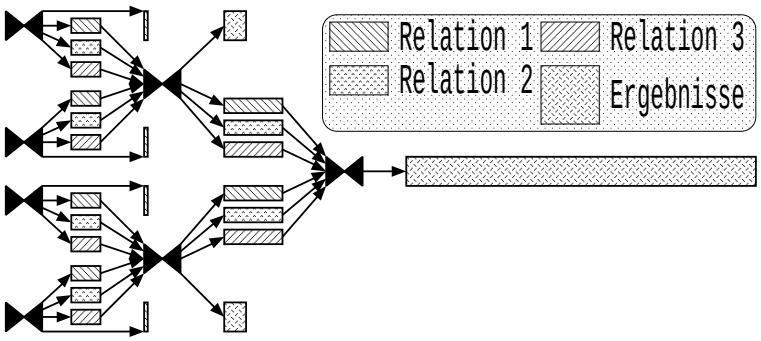


Abbildung 3.2: Mehrdimensionaler Progressive-Merge-Join

Sortierungen stehen. Bei zwei Relationen verwendet man eine gemeinsame Ordnung, nun kann man eine gemeinsame totale Ordnung O auf allen r beteiligten Relationen fordern. Gilt dann $P_{join}(t_1, \ldots, t_r),\ t_i \in R_i$, so sollten alle Tupel t_i bezüglich O in einem relativ kleinen Bereich liegen.

Eine dafür geeignete Ordnung O läßt sich leider nicht in allen Fällen finden. Jede Kombination aus binären Joins stellt ihrerseits einen multidimensionalen Join dar. Dabei werden ggf. verschiedene Sortierungen verwendet, die nicht geeignet vereinbar sind. Auf eine Anpassung des PMJ an diese Joins wird in Kapitel 6.2.2 eingegangen, dieses beschränkt sich auf den Fall, daß eine geeignete gemeinsame Ordnung auf allen Relationen definierbar ist.

3.1.3 Konzept

Dieser multidimensionale Progressive-Merge-Join soll also den Join von r Relationen $R_1, \ldots, R_r$ nach einem Joinprädikat P_{join} berechnen. Dabei wird vorausgesetzt, daß eine gemeinsame totale Ordnung O auf den Relationen definiert ist. Für den Vergleich zweier Tupel $t_i \in R_i$ und $t_j \in R_j$ mit O sei die Schreibweise $t_i \leq_{i,j} t_j$ eingeführt.

Bei der Erzeugung initialer Runs soll der Hauptspeicher zwischen den Relationen aufgeteilt und jeder Teil $\hat{R}_i$ mittels $\leq_{i,i}$ sortiert werden, um dann den Join $\bowtie_{P_{join}} (\hat{R}_1, \ldots, \hat{R}_r)$ zu berechnen. Zur Unterstützung asymmetrischer Joins und gesteigerter Performanz durch Flexibilität soll für jede Relation eine eigene Art SweepArea unterstützt werden. Dies soll sich analog zum PMJ entsprechend auch in der Merge-Phase fortsetzen.

Für drei Relationen zeigt Abbildung 3.2 beispielhaft, wie die Berechnung ablaufen soll.

3.2 Der Algorithmus MPMJ

Die Umsetzung der beschriebenen Zielsetzung ergibt nun eine mehrdimensionale Version des PMJ, genannt *MPMJ*, die in diesem Abschnitt vorgestellt wird. Die Beschreibung geht dabei vorwiegend auf die Änderungen gegenüber dem ursprünglichen PMJ (2.2) ein.

3.2.1 Grundlegender Algorithmus

Beim Grundgerüst des erweiterten PMJ (Algorithmus 3.1) sind die Änderungen durch den Übergang auf mehrdimensionale Joins bedingt. Was im PMJ für beide beteiligten Relationen explizit angegeben wurde, wird nun durch Schleifen über die k Relationen erledigt.

Den Hauptspeicher in Phase 1 und den Fan-In in Phase 2 müssen sich alle Relationen teilen, es wird also über ihren jeweiligen Bedarf summiert.

Für zwei Relationen gibt es beim PMJ eine Menge Q, um Runpaare zu verwalten. Diese Paarung wurde nicht auf größere Tupel erweitert, stattdessen werden die Runs zu jeder Relation R_i nun in einer eigenen Menge Q_i verwaltet. Diese Änderung macht auf den ersten Blick einen eher technischen Eindruck, ist aber im Hinblick auf spätere Optimierungen (siehe Kapitel 4) ein wichtiger Schritt hin zu einer flexibleren Verwaltung und Verarbeitung der Runs.

3.2.2 Das Joinen von Runs

Die in 2.5.1 beschriebene Implementierungsoption, den Join initialer und späterer Runs mit Hilfe einer gemeinsamen Funktion zu berechnen, wurde hier bereits im Pseudocode umgesetzt; das Ergebnis ist die Funktion `earlyJoin` (Funktion 3.2).

Diese verarbeitet nun also nicht mehr Runpaare, sondern bekommt eine Menge von Runs aus jeder Relation übergeben, deren Join sie berechnet. Zudem wird für jede der Relationen der Merge der zugehörigen Runs gebildet. In der Phase der Erzeugung initialer Runs werden dabei natürlich genau wieder die übergebenen Runs produziert.

In `earlyJoin` werden zunächst leere SweepAreas angelegt, und zwar zu jeder Relation eine dem Join angemessene Variante. Dann wird bis zur vollständigen Konsumierung aller Runs das kleinste unverarbeitete Element t aus allen übergebenen Runs bestimmt. a und b geben dabei jeweils an, aus welcher Relation und welchem derer Runs t entnommen wurde. Jenes wird nun zunächst zum Reorganisieren aller SweepAreas verwendet und entfernt

Algorithmus 3.1 Mehrdimensionaler Progressive-Merge-Join

IN:	$R_1, \ldots, R_r$	die Eingaberelationen
	M	maximale Anzahl von Tupeln, die in den Hauptspeicher passen
	F	maximale Seitenzahl zum Lesen der Runs
OUT:	$Result$	die Joinergebnisse

1: Let $Result := \emptyset$;
2: **for** $k := 1$ to r **do**
3: Let $Q_k := \emptyset$;
4: **end for**
5: {Phase 1: Join während Runerzeugung}
6: **while** $\exists i : R_i \neq \emptyset$ **do**
7: Let $\hat{R}_j \subset R_j,\ j = 1, \ldots, r$, where $\sum_{k=1}^{r} |\hat{R}_k| \leq M$;
8: **for** $k := 1$ to r **do**
9: Let $R_k := R_k \backslash \hat{R}_k$;
10: Sort $\hat{R}_k$ into sequence $\hat{R}'_k$;
11: **end for**
12: $earlyJoin((\{\hat{R}'_1\}, \ldots, \{\hat{R}'_r\}), True, Result, (\emptyset, \ldots, \emptyset))$;
13: **for** $k := 1$ to r **do**
14: Let $Q_k := Q_k \cup \{\hat{R}'_k\}$;
15: **end for**
16: **end while**
17: {Phase 2: Join während Merge}
18: **while** $\exists i : |Q_i| > 1$ **do**
19: Let $\hat{Q}_j \subset Q_j,\ j = 1, \ldots, r$, where $\sum_{k=1}^{r} |\hat{Q}_k| \leq F$;
20: Let $(\hat{R}'_1, \ldots, \hat{R}'_r)$ be a tuple of empty sequences;
21: $earlyJoin((\hat{Q}_1, \ldots, \hat{Q}_r), (\sum_{k=1}^{r} |\hat{Q}_k| = \sum_{k=1}^{r} |Q_k|), Result, (\hat{R}'_1, \ldots, \hat{R}'_r))$;
22: **for** $k := 1$ to r **do**
23: Let $Q_k := (Q_k \backslash \hat{Q}_k) \cup \{\hat{R}'_k\}$;
24: **end for**
25: **end while**

Funktion 3.2 earlyJoin

IN:	$(\{R'^1_1, \ldots, R'^{n_1}_1\}, \ldots, \{R'^1_r, \ldots, R'^{n_r}_r\})$	Mengen sortierter Eingabefolgen
	writeout	Runs schreiben?
IN/OUT:	*Result*	Menge der Joinergebnis
OUT:	$(\hat{R}'_1, \ldots, \hat{R}'_r)$	sortierte Ausgabefolgen

1: **for** $k := 1$ to r **do**
2: Let S_k be an empty SweepArea for Relation R_k;
3: **end for**
4: **while** $\exists i\ \exists m:\ R'^m_i \neq \emptyset$ **do**
5: Let $t := First(R'^b_a)$
where $r = minimum\{First(R'^d_c) \mid c \in \{1, \ldots, r\},\ d \in \{1, \ldots, n_c\}\}$;
6: **for** $k := 1$ to r **do**
7: $S_k.reorganize(t, a)$;
8: **end for**
9: $S_a.insert(t)$;
10: Let $QueryResult := S_{qo(a,1)}.query((t), a, (S_1, \ldots, S_r)) \setminus Result$;
11: Let $Result := Result \cup \{u \in QueryResult \mid P_{join}(u)\}$;
12: Let $R'^b_a := R'^b_a \setminus \{t\}$;
13: Let $\hat{R}'_a := \hat{R}_a \cup \{t\}$;
14: **if** *writeout* **then**
15: Write t to external memory for $\hat{R}'_a$;
16: **end if**
17: **end while**

dabei alle Einträge, die fortan nicht mehr an Joins beteiligt sein können. Dann wird t in seine SweepArea eingefügt.

Nun liegt das Joinen an. Es müssen alle Tupel $(y_1, \ldots, y_{a-1}, t, y_{a+1}, \ldots, y_r)$ gefunden werden, die das Joinprädikat erfüllen, wobei die y_i die bereits in den SweepAreas befindlichen mit t joinenden Tupel sind. Man könnte alle solchen Tupel erzeugen (also das Kreuzprodukt aus $\{t\}$ und den SweepAreas außer der zu Relation R_a bilden) und gegen P_{join} testen. Dies widerspräche aber dem als effektiv erkannten Prinzip, die innere Struktur der SweepAreas zur besseren Joinverarbeitung auszunutzen. Also ist die Befragung der SweepAreas angesagt. Beim binären Join ist klar, welche zu befragen ist; bei mehr als zwei beteiligten Relationen stellt sich jedoch die Frage nach der Reihenfolge. Diese wird wieder so beantwortet, daß dem Zusammenspiel der SweepAreas keine unnötigen Limits auferlegt werden sollen, sie wird also flexibel gehalten. Eine Funktion $qo(c, n)$ gibt an, welche SweepArea bei der

Abarbeitung eines Tupels aus Relation R_c als n-te angefragt wird. Die Anfrage richtet sich also zunächst an die SweepArea $S_{qo(a,1)}$, der das Tupel t als einelementige Folge und die Nummer a der aufrufenden Relation R_a mitgeteilt wird. Die Anfrage wird dort intern weiterverarbeitet, bis alle Ergebnisse in *QueryResult* vorliegen, wobei zuvor bereits produzierte Ergebnisse nicht nochmals erzeugt werden. Nun werden genau die Tupel in die Ergebnismenge `Result` eingefügt, die das Joinprädikat erfüllen.

Die Abarbeitung von t endet damit, daß es dem Ausgaberun zu seiner Relation zugeordnet und auf den Externspeicher herausgeschrieben wird. Dabei kommt hier auch klar zum Ausdruck, daß die Externspeicherzugriffe überlappend mit den Berechnungen erfolgen können und die erzeugten Runs nicht im Hauptspeicher verbleiben. Beim letzten Aufruf von `earlyJoin` wird das Herausschreiben der Tupel mit Hilfe des Flags `writeout` unterdrückt.

3.2.3 SweepAreas

Bei der veränderten Version der SweepAreas (Algorithmus 3.3) zeigt sich zunächst, daß diese nun wissen, zu welcher Relation z sie gehören. Während sich an `insert` nichts geändert hat, wurde die Funktionalität von `query` teilweise auf die Funktion `reorganize` übertragen, welche das explizite Bereinigen der SweepArea aufgrund der Information, daß in der Sortierung das Element v erreicht wurde, ermöglicht, was der in 2.5.4 beschriebenen Optimierung entspricht. Dabei wird auch übergeben, aus welcher Relation R_a das Tupel v stammt, um eine unterschiedliche Semantik von Tupeln bei asymmetrischen Joins zu ermöglichen.

Die Funktion `query` übernimmt die Anfragebearbeitung an die SweepAreas. Beim Aufruf in `earlyJoin` werden ihr das gerade zur Bearbeitung anstehende Tupel t in einer einelementigen Folge (t), die Nummer a der Relation, aus der t stammt, sowie alle SweepAreas übergeben. Sie muß nun alle Tupel im Kreuzprodukt aus $\{t\}$ und den Inhalten der SweepAreas außer S_a finden, die das Joinprädikat P_{join} erfüllen könnten. Dazu wird eine rekursive Suche durchgeführt, bei der die anfängliche Folge (t) bei jedem Aufruf von `query` um ein Element aus der angefragten SweepArea verlängert wird, bis sie die Länge r erreicht. Solche Folgen in Anfragereihenfolge werden dann zu Tupeln in Reihenfolge der Relationen umsortiert und zurückgeliefert.

Zur effektiven Durchführung rekursiver Suchen hat sich das Konzept des *Prunings* bewährt, bei dem der rekursive Aufbau von Ergebnissen an den Stellen vorzeitig abgebrochen wird, an denen bereits feststeht, daß kein gültiges Ergebnis mehr erreicht werden kann. Dieses wird auch hier eingesetzt, indem bei Aufrufen an `query` die Menge X der Elemente in der SweepArea für die Anfrage auf die Menge Y derjenigen davon eingeschränkt wird, die nicht

Datentyp 3.3 Erweiterte SweepArea

1: Predicate $P_{rm_{i,j}}, P_{prune_{k,l}},\ i,j,k,l \in \{1,\ldots,r\},\ l > 1$;
2: Let $X := \emptyset$;
3: Let z be the number of this SweepArea;
4: Procedure insert(Element v) [
5: Let $X := X \cup \{v\}$;
6:];
7: Procedure reorganize(Element v, int a) [
8: Let $X := X \setminus \{u \in X \mid P_{rm_{z,a}}(u,v)\}$;
9:];
10: Function query(Elements $(v_1,\ldots,v_p)$, int a, SweepAreas $(S_1,\ldots,S_r)$) [
11: **if** $p = r$ **then**
12: return $\{(w_1,\ldots,w_r) \mid w_a = v_1,\ w_{qo(a,k)} = v_{k+1},\ k = 1,\ldots,r-1\}$;
13: **else**
14: Let $Res := \emptyset$;
15: Let $Y := \{x \in X \mid \neg P_{prune_{a,p+1}}(v_1,\ldots,v_p,x)\}$;
16: **for all** $y \in Y$ **do**
17: Let $Res := Res\ \cup\ S_{qo(a,p+1)}.query((v_1,\ldots,v_p,y),a,(S_1,\ldots,S_r))$;
18: **end for**
19: return Res;
20: **end if**
21:];

zusammen mit den bereits auf der Suche festgelegten Elementen $v_1,\ldots,v_p$ das Pruning-Prädikat $P_{prune_{a,p+1}}$ erfüllen. Dieses erkennt in Abhängigkeit von Suchbeginnrelationsnummer a und Suchtiefe p die Elemente aus X, welche nicht mehr zum Joinen in Frage kommen.

3.3 Korrektheit

Der Korrektheitsbeweis für den mehrdimensionalen PMJ verläuft ebenso wie der zum PMJ in 2.3 aufgeteilt in Aufstellung hinreichender Bedingungen (3.3.1) und Beweis von Korrektheit (3.3.2), Vollständigkeit (3.3.3) und Einmaligkeit (3.3.4) der Ergebnisse. Zu zeigen ist wieder, daß die Ergebnismenge am Ende genau diejenigen Tupel aus dem Kreuzprodukt der Eingaberelationen $R_1,\ldots,R_r$ enthält, die das Joinprädikat P_{join} erfüllen:

$$Result = \{(t_1,\ldots,t_r) \mid t_i \in R_i,\ P_{join}(t_1,\ldots,t_r)\} \tag{3.1}$$

3.3.1 Bedingungen

Neben der Bedingung an das Entfernungsprädikat P_{rm} beim PMJ, das hier auf mehrere Prädikate für die verschiedenen SweepAreas erweitert wird (3.3.1.1), kommt nun mit den Säuberungsprädikaten P_{prune} auch eine Bedingung hinzu, die sicherstellt, daß diese für den Join geeignet sind (3.3.1.2).

3.3.1.1 Bedingung an die Entfernungsprädikate

Seien $i, j, k \in \{1, \ldots, r\},\ v_i \in R_i,\ v_j \in R_j,\ v_k \in R_k$ und P_{join} das Joinprädikat. Für die SweepArea-Leerungsprädikate $P_{rm_{a,b}}$ (wobei $P_{rm_{a,b}}(x, y)$ bedeute, daß $y \in R_b$ das Element x aus der zu R_a gehörigen SweepArea entfernt) muß folgende Bedingung erfüllt sein:

$$v_i \leq_{i,j} v_j \ \wedge\ P_{rm_{k,i}}(v_k, v_i) \quad \Rightarrow \quad \neg P_{join}(\ldots, v_k, \ldots, v_j, \ldots) \tag{3.2}$$

Die Bedingung besagt, daß ein Element v_k erst aus seiner SweepArea entfernt werden darf, wenn alle in den Ordnungen hinter dem entfernenden Element v_i stehenden Elemente v_j nicht zusammen mit v_k an der Bildung eines Tuples beteiligt sein können, das die Joinbedingung erfüllt, also wenn alle Ergebnisse, in die v_k eingeht, schon vor dem Auftreten von v_i produziert wurden.

Diese Bedingung entspricht der in 2.3.1 für den PMJ formulierten zur Verhinderung des verfrühten Löschens von Elementen aus den SweepAreas.

3.3.1.2 Bedingung an die Säuberungsprädikate

Die neue Bedingung hat die Aufgabe zu verhindern, daß die Produktion korrekter Joinergebnisse durch die Säuberungsprädikate P_{prune} unterdrückt wird. Für alle $v_1 \in R_1, \ldots, v_r \in R_r$, alle $a \in \{1, \ldots, r\}$ und alle $p \in \{1, \ldots, r-1\}$ muß gelten:

$$P_{prune_{a,p+1}}(v_a, v_{qo(a,1)}, \ldots, v_{qo(a,p)}) \quad \Rightarrow \quad \neg P_{join}(v_1, \ldots, v_r) \tag{3.3}$$

Also nur, wenn $(v_1, \ldots, v_r)$ kein Joinergebnis ist, darf eine von einem Tupel v_a angestoßene rekursive Suche nach diesem Ergebnis in einem Schritt p abgebrochen werden.

3.3.2 Korrektheit der Ergebnisse

Alle Elemente der Ergebnismenge `Result` des mehrdimensionalen PMJ werden von Aufrufen der Funktion `earlyJoin` erzeugt. Diese wieder fügt nur Tupel in die Ergebnismenge ein, die explizit das Joinprädikat P_{join} erfüllen, also

$$Result \subseteq \{(t_1, \ldots, t_r) \mid t_i \in R_i,\ P_{join}(t_1, \ldots, t_r)\}$$

3.3.3 Vollständigkeit der Ergebnismenge

Seien $v_1 \in R_1, \ldots, v_r \in R_r$ mit $P_{join}(v_1, \ldots, v_r)$. Zu zeigen ist, daß $(v_1, \ldots, v_r)$ als Joinergebnis erkannt wird. Die Funktion `earlyJoin` wird solange zum Joinen und Mergen von Runs aufgerufen, bis sie im letzten Schritt nur noch einen Run pro Relation produziert, wozu sie alle Tupel aller Relationen verarbeiten muß. Daher muß es einen ersten Aufruf geben, bei dem die Funktion alle Tupel aus $\{v_1, \ldots, v_r\}$ verarbeitet, und zwar in einer Reihenfolge $(v_{e_1}, \ldots, v_{e_r})$.

Zum Zeitpunkt Z des Aufrufs von `query` der SweepArea S_{e_r} mit dem Tupel v_{e_r} wurden also alle Tupel $v_1, \ldots, v_r$ bereits gelesen und in ihre SweepArea eingefügt. Wäre eines dieser Elemente $v_f \in \{v_1, \ldots, v_r\}$ schon wieder daraus entfernt worden, so hätte dies durch ein Element $u_g \in R_g$ geschehen müssen, für das wegen der sortierten Verarbeitung $u_g \leq_{g,e_r} v_{e_r}$ gelten würde, und das zum Entfernen $P_{rm}(v_l, u_g)$ erfüllt haben müßte. Daraus folgte mit Bedingung 3.2 aber der Widerspruch $\neg P_{join}(\ldots, v_l, \ldots, v_{e_r}, \ldots)$. Bei Z befinden sich also alle Elemente $v_1, \ldots, v_r$ in ihren SweepAreas.

Der bei Z erfolgte Aufruf baut nun bei seiner rekursiven Suche in der durch $qo(e_r, i)$, $i = 1, \ldots, r-1$ vorgegebenen Reihenfolge u. a. das Tupel $(v_{e_r}, v_{qo(e_r,1)}, \ldots, v_{qo(e_r,r-1)})$ auf. Würde dieser Aufbau vorzeitig durch Pruning abgebrochen, so müßte für ein $i \in \{1, \ldots, r-1\}$ die Bedingung $P_{prune_{e_r,i+1}}(v_{e_r}, v_{qo(e_r,1)}, \ldots, v_{qo(e_r,i)})$ gelten, was mit Bedingung 3.3 aber wieder den Widerspruch $\neg P_{join}(v_1, \ldots, v_r)$ lieferte. $(v_{e_r}, v_{qo(e_r,1)}, \ldots, v_{qo(e_r,r-1)})$ wird also aufgebaut, im Abbruchfall der Rekursion zu $(v_1, \ldots, v_r)$ umsortiert, zurückgeliefert und wegen $P_{join}(v_1, \ldots, v_r)$ der Ergebnismenge zugeordnet, für die also gilt:

$$Result \supseteq \{(t_1, \ldots, t_r) \mid t_i \in R_i,\ P_{join}(t_1, \ldots, t_r)\}$$

3.3.4 Einmaligkeit der Ergebnisse

Während eines einzelnen Aufrufs von `earlyMerge` wird ein Ergebnis maximal einmal gefunden, da die Komponente des dort gerade zur Verarbeitung anstehenden Tupels t bei den rekursiven Anfragen mit `query` auf dieses Tupel fixiert ist und kein vorher produziertes Ergebnis auftreten kann, da t zuvor ja noch nicht zur Verfügung stand. Um das mehrmalige Einfügen von Ergebnissen in die Ergebnismenge *Result* während der gesamten Verarbeitung zu verhindern, wird der Funktion `earlyMerge` im Pseudocode die globale Ergebnismenge des Algorithmus als IN/OUT-Parameter zur Verfügung gestellt, wodurch sie aus der Kandidatenmenge für neue Ergebnisse `QueryResult` zuvor produzierte Ergebnisse einfach aussortieren kann.

Diese Vorgehensweise ist für eine Implementierung natürlich ungeeignet, weil dafür die Gesamtmenge aller zuvor produzierten Ergebnisse immer wie-

der benutzt werden müßte, obwohl für jene gar kein Speicher vorhanden ist, da Ergebnisse normalerweise sofort bei Erzeugung unwiderruflich abgeliefert werden. Die Duplikateliminierung wird daher in 3.4.2 diskutiert.

3.4 Implementierungsaspekte

In diesem Abschnitt werden wichtige Aspekte für die Implementierung des mehrdimensionalen Progressive-Merge-Joins aufgezeigt und erläutert, wie diese in der Referenzimplementierung, die Teil dieser Arbeit ist, behandelt wurden. Die in 2.5.2 und 2.5.4 beschriebenen Optimierungen wurden hier schon im Pseudocode eingearbeitet und sind auch in der Implementierung umgesetzt.

3.4.1 Early Joins

`earlyJoin` ist wie im Pseudocode als eine Funktion implementiert, die sowohl initiale als auch andere Runs joinen und mergen kann. Zur Bestimmung des jeweils minimalen Elements über alle zu verarbeitenden Runs zu allen Relationen findet immer noch ein MinHeap Verwendung, in dem jeweils die aktuellen Minima der Runs liegen. Da die Runs nun nicht mehr in Paaren organisiert sind, muß die Bestimmung des Herkunftsruns eines Tupels bei seiner Entnahme aus dem Heap anders als beim PMJ geregelt werden. Dazu wird jedes Element vor dem Einfügen in die SweepArea in ein Wrapper-Objekt verpackt, welches Ursprungsrelation und Runnummer enthält.

Im Heap müssen die Tupel bezüglich der globalen Ordnung O über alle Relationen eingeordnet werden. Ein einzelnes Prädikat $\leq_O$ zur Berechnung des Vergleichs bezüglich O hätte nun das Problem, daß es erst die Art seiner beiden Parameter bestimmen müßte, um sie dann entsprechend zu vergleichen. Da Tupel, die aus verschiedenen Relationen stammen, dennoch übereinstimmen können, ihre Stellung in der Ordnung aber ggf. davon abhängt, aus welcher Relation sie kommen, könnte ein solches Prädikat sogar nicht immer angegeben werden. Daher verwendet die Implementierung ein Feld von Prädikaten $\leq_{i,j},\ i,j \in \{1,\ldots,r\}$, wobei für $a \in R_i,\ b \in R_j$ gilt: $a \leq_{i,j} b :\Leftrightarrow a \leq_O b$. Die in den Heap eingefügten Wrapper können daher anhand der enthaltenen Tupel bezüglich O eingeordnet werden, indem zum Vergleich zweier Wrapper die gekapselten Elemente mit Hilfe des passenden Prädikates $\leq_{i,j}$ verglichen werden, da die Relationsnummern i und j in den Wrappern gespeichert sind. Im minimalen Wrapper an der Wurzel des Heaps ist damit das minimale Tupel bezüglich O gekapselt und kann entnommen werden.

3.4.2 Duplikateliminierung

Ein Duplikat ist ein Ergebnis $(v_1, \ldots, v_r)$, welches während der Joinberechnung mehrmals als Ergebnis produziert wird. Es darf nur beim ersten Auftreten der Ergebnismenge `Result` zugeführt werden, bei weiterem Auftreten muß es als Duplikat erkannt werden. Diese Erkennung ist ein echter Mehraufwand des PMJ gegenüber dem Sort-Merge-Join ohne frühe Ergebnisse, bei dem jedes Ergebnis im letzten Schritt und damit nur einmal erzeugt wird.

Bei der Generierung initialer Runs können noch keine Duplikate auftreten, da jedes Tupel nur in einen derer gelangt und ein einzelner Aufruf von `earlyJoin` keine doppelten Ergebnisse produziert. Daher wird der Mechanismus zur Duplikateliminierung zur Steigerung der Effektivität in Phase 1 des mehrdimensionalen PMJ abgeschaltet.

In Phase 2 ist $(v_1, \ldots, v_r)$ nun genau dann ein Duplikat, wenn die beteiligten Tupel schon zuvor einmal alle im selben Aufruf von `earlyJoin` verarbeitet wurden. Danach befinden sie sich alle in den bei diesem Aufruf erzeugten Runs. Es muß also entschieden werden, ob dies für $v_1, \ldots, v_r$ zuvor mindestens einmal erfüllt war, weshalb diese Information immer rekonstruierbar bleiben muß. In der Referenzimplementierung muß dies für `earlyJoin` immer daraus feststellbar sein, aus welchem Run die Tupel bei diesem Aufruf stammen. Die Strategie dazu wird in 4.2.1.2.1 erläutert.

Wird $(v_1, \ldots, v_r)$ als Ergebnis einer Anfrage an die SweepAreas zurückgeliefert, so muß also noch die Information verfügbar sein, aus welchen Runs diese eingefügt wurden. Dazu könnte man wie in der Referenzimplementierung des binären PMJ eine SweepArea pro Run einrichten. Dies zieht jedoch schon dort statt je einer Anfrage an eine SweepArea je eine pro Run nach sich. Im Falle von mehr beteiligten Relationen würde sich dieser Effekt exponentiell ausweiten, da jedes Ergebnis einer Anfrage an die SweepArea der zweiten Relation Anfragen an alle SweepAreas der dritten nach sich zöge usw.. Daher wird der auch in [Dit02] als Alternative genannte Ansatz umgesetzt, auch in der Implementierung nur eine SweepArea pro Relation zu betreiben. Wird ein Tupel aus einer SweepArea geliefert, muß nun dessen ursprünglicher Run erkennbar sein. Genau dieses Problem wurde aber schon für den Heap mit Hilfe der Wrapper gelöst, die nun weiterverwendet werden, indem die Tupel auch mitsamt Warpper in die SweepAreas eingefügt werden.

Leider kann eine SweepArea zunächst einmal nur die normalen Tupel verarbeiten. Sie muß daher dahingehend erweitert werden, daß sie alternativ auch mit in Wrappern eingepackten Tupeln betrieben werden kann. Das Verhalten muß dabei exakt dem ohne Wrapper entsprechen. Leider kann man dazu nicht einfach die Wrapper entfernen und eine gewöhnliche SweepArea benutzen, da die Wrapper beim Zurückliefern der Tupel wieder hinzugefügt

werden müßten. Dies ist zwar möglich, indem man sie in einer Indexstruktur ablegt, in der sie anhand der eingekapselten Tupel wieder auffindbar sind, effektiver ist jedoch eine Anpassung der SweepArea an den PMJ, die in jedem Fall einfach möglich sein sollte.

3.4.3 Entrekursivierung der Anfragebearbeitung

Die rekursive Verarbeitung der Anfragen mit Hilfe der Funktion `query` ist zwar relativ einfach zu implementieren, erweist sich aber leider als recht langsam. Da die Rekursionstiefe durch r begrenzt ist, kann der Aufrufstack mit Hilfe von Arrays nachgebildet werden und eine effektive Entrekursivierung ist möglich. Eine solche wurde in der Referenzimplementierung vorgenommen und der iterative Ergebnisaufbau in `earlyJoin` integriert. Die SweepAreas müssen bei Anfragen diese nun nicht mehr selber abarbeiten, sondern nur noch die in ihnen gespeicherten Tupel zurückliefern, die nicht durch das Pruning abgelehnt werden können. Bei der Implementierung von SweepAreas kann man sich daher auf deren Anpassung an den jeweiligen Join konzentrieren.

3.5 Schätzer

Auch beim Joinen von Relationen $R_1, \ldots, R_r$, $r > 2$, besteht das Problem, daß das Kreuzprodukt $\tilde{K} := \tilde{R}_1 \times \cdots \times \tilde{R}_r$ von Stichproben $\tilde{R}_i \subset R_i$ keine Stichprobe des Kreuzproduktes $K := R_1 \times \cdots \times R_r$ darstellt. Erhalten bleibt aber auch die positive Tatsache, daß jedes Tupel daraus einzeln betrachtet mit gleicher Wahrscheinlichkeit in $\tilde{K}$ auftritt. Als Schätzer für die Selektivität kann daher weiterhin der Quotient aus der Zahl schon gefundener Joinergebnisse und der Größe der Teilmenge E von R, die dabei untersucht wurde, also der Vereiningung der betrachteten Mengen $\tilde{K}$, verwendet werden: $\frac{|\{e \in E \mid P_{join}(e)\}|}{|E|}$. Eine Aktualisierung des Schätzers kann weiterhin nach jeder kompletten Ausführung eines `earlyJoin` erfolgen.

Duplikate spielen auch für den Schätzer eine Rolle. Beim Mergen von Runs bildet die Vereinigung mehrerer Runs zu einem zwar jeweils wieder eine größere Stichprobe der Ursprungsrelation, ein Teil des Kreuzproduktes dieser Stichproben wurde jedoch schon zuvor auf Ergebnisse untersucht. Wird die Größe des bisher untersuchten Bereichs E also durch fortlaufende Addition der Größen der Bereiche der Aufrufe von `earlyMerge` ermittelt, so werden die Bereiche, in denen Duplikate auftreten können, ggf. mehrmals gezählt. Dies kann man kompensieren, indem man auch die Duplikate für den Schätzer nochmals als Ergebnisse wertet. Die exakte Lösung verlangt hingegen eine

genaue Berechnung von $|E|$, welche jedoch zusätzlichen Aufwand erfordern würde.

3.6 Anwendungsgebiete

Auch der mehrdimensionale PMJ erzeugt mit $P_{rm_{i,k}} := False$, $P_{prune_{i,j}} := False$ das Kreuzprodukt der Eingaberelationen und testest jedes Element gegen P_{join}, kann also theoretisch jede Art von Join berechnen, wenn die Problematik des Auslagerns der SweepAreas gelöst ist. Dies ist jedoch ggf. sehr ineffizient.

Als wirklich geeignete Einsatzgebiete für den mehrdimensionalen PMJ lassen sich diejenigen Joins mit Joinbedingung P_{join} identifizieren, bei denen sich auf den zu joinenden Relationen $R_1, \ldots, R_r$ eine gemeinsame Ordnung O definieren läßt, so daß für $(v_1, \ldots, v_r)$ mit $P_{join}(v_1, \ldots, v_r)$ die Tupel $v_1, \ldots, v_r$ bezüglich O mit hoher Wahrscheinlichkeit alle relativ nahe beieinander stehen.

3.6.1 Verallgemeinerung binärer Joins

Für $r = 2$ wurden in 2.8 bereits Joins vorgestellt, für die der PMJ gut anwendbar ist. Equi-, Band-, Temporal- und Spatial-Join lassen sich auch für $r > 2$ verallgemeinern, indem man die jeweilige Joinbedingung auf Paare (R_i, R_j) überträgt. Für $K := \{1, \ldots, r\}$ und eine Menge $E := \{(i,j) \in K^2 |\ i \neq j\}$ kann man also definieren: $P_{join}^{mehrdim.}(v_1, \ldots, v_r) :\Leftrightarrow \forall (i,j) \in M :\ P_{join}^{binär}(v_i, v_j)$[1].

Faßt man $G := (K, E)$ als Graphen auf (für symmetrische binäre Bedingung $P_{join}^{binär}$ als ungerichteten), so machen für eine Joinverarbeitung mit gemeinsamer Sortierung der Relationen nur Fälle Sinn, in denen diese alle in Zusammenhang miteinander stehen, G also (im ungerichteten Fall stark) zusammenhängend ist. Von besonderem Interesse sind die Extremfälle, in denen G einen Baum oder vollständiger Graphen darstellt.

Beim Aufbau eines Joinergebnisses durch den MPMJ wird dieses von einer Relation R_a ausgehend durch Hinzunahme von Tupeln anhand der Anfragereihenfolge $qo(a, i)$ aufgebaut. Betrachtet man $(v_a, v_{qo(a,1)}, \ldots, v_{qo(a,j)})$, so kann man darin die Bedingungen des von $\{a, qo(a,1), \ldots, qo(a,j)\}$ aufgespannten Teilgraphen prüfen. Folgt man bei zusammenhängendem G mit qo einem aufspannenden Baum, so kann in jedem Schritt ein Pruning mit Hilfe der hinzukommenden binären Bedingungen erfolgen.

[1] Noch allgemeiner könnte man verschiedene Joinbedingungen zwischen den Relationspaaren zulassen.

Leider ergibt sich ein Problem bei den Entfernungsprädikaten $P_{rm_{i,j}}$. Diesen stehen immer nur eine SweepArea und das zur Verarbeitung anstehende Tupel zur Auswertung zur Verfügung, nicht jedoch die in den anderen SweepAreas befindlichen Tupel. Von diesen kann aber ggf. die Entscheidung abhängen, ob ein Tupel noch zu Joinergebnissen beitragen kann. Wegen Bedingung 3.2 können daher die Entfernungsprädikate nicht immer scharf genug gefaßt werden, wodurch die SweepAreas während der Joinverarbeitung zu groß würden und ein Einsatz des mehrdimensionalen PMJ nicht zu empfehlen ist.

Ist G aber ein vollständiger Graph, so wird die binäre Bedingung $P_{join}^{binär}$ auf jeden Fall paarweise zwischen den Relationen gefordert, so daß das zugehörige Entfernungsprädikat für den binären Join anwendbar bleibt und damit der MPMJ zum Einsatz kommen kann.

Ist P_{join} symmetrisch und transitiv, so folgt aus der Eigenschaft, daß G zusammenhängend ist, direkt, daß G vollständig ist.

3.6.2 Equi-Join

Da die Gleichheit ein transitives Prädikat darstellt, ist hier nur der Join von Interesse, bei dem Gleichheit aller joinenden Tupel auf einem Attribut (oder einer Menge davon) gefordert wird. Dieser kann wie zuvor erläutert sehr gut mit dem MPMJ berechnet werden. P_{rm} wird wie beim binären Equi-Join gewählt, die interne Struktur der SweepAreas kann auch bestehen bleiben. Ein Pruning ist nicht nötig, die Gleichheit braucht für die Ergebnisse auch nicht mehr mit P_{join} überprüft zu werden, da sie sich weiterhin für alle in den SweepAreas gespeicherten Tupeln daraus ergibt, daß sie nicht im Entfernungsschritt vor der Anfrage daraus gelöscht wurden.

Auch im mehrdimensionalen Fall spielt der Equi-Join eine sehr wichtige Rolle, da er auch hier u. a. zur Auflösung von Fremdschlüsselbeziehungen eingesetzt werden kann. Die Beschreibung des MJoins in [VNB02] konzentriert sich ausschließlich auf den Equi-Join.

3.6.3 Band-Join

Die Bedingung beim binären Band-Join ist, wie auch bei folgenden Joinarten, nicht transitiv. Für zusammenhängendes G kann aber ein Tupel $(v_1, \ldots, v_r)$ kein Joinergebnis sein, wenn es darin ein Paar (v_i, v_j) gibt, so daß die darin enthaltenen Punkte A_i bzw. B_j einen Abstand größer als $(r-1)\cdot\varepsilon$ haben, wobei ε der Maximalabstand beim binären Band-Join sei. Damit kann man also für hinreichend kleines ε immer noch geeignete binäre Entfernungsprädikate

angeben, die einen Einsatz des MPMJ ermöglichen. Zudem kann auch Pruning zum Einsatz kommen, indem der Maximalabstand ε auf Paaren geprüft wird.

3.6.4 Temporal-Join und Spatial-Join

Beim Temporal- und Spatial-Join kann im schlimmsten Fall ein einzelnes Intervall bzw. ein einzelnes Rechteck mit allen anderen in sämtlichen beteiligten Relationen die binäre Joinbedingung erfüllen. Fehlt für $r > 2$ in E also die Kante (i, j), so kann in den SweepAreas S_i und S_j nie ein Element entfernt werden, da dieses mit später ankommenden immer noch über eines aus einer anderen Relation $R_k,\ k \neq i, j$ verbunden sein könnte. Daher können diese Joinarten nur für vollständigen Graphen G wie in 3.6.1 beschrieben verallgemeinert werden.

Zusammenfassung

Der mehrdimensionale Progressive-Merge-Join (MPMJ) ist eine Weiterentwicklung des PMJ, die auch asymmetrische Joins unterstützt und Joins über mehr als zwei Eingaberelationen erlaubt, wenn sich darauf eine geeignete gemeinsame Ordnung definieren läßt.

Kapitel 4

Optimierungen

Übersicht

Dieses Kapitel widmet sich der Optimierung des in Kapitel 3 vorgestellten MPMJ, wobei viele Konzepte auch auf den zugrundeliegenden PMJ anwendbar sind. Zunächst werden in 4.1 einige Vorüberlegungen angestellt und Ziele der Optimierung gesetzt. Diese wird nun in 4.2 zunächst durch Wahl günstiger Parameter des MPMJ erreicht. In 4.3 werden dann diverse Veränderungen am Algorithmus vorgestellt, die dessen Effektivität noch steigern. Aspekte der Implementierung werden in 4.4 erläutert.

4.1 Vorüberlegungen

4.1.1 Struktur des Sort-Merge-Baumes

In der letzten Joinphase sowohl des Sort-Merge-Joins mit Join-During-Merge (1.5.2.4.2) als auch des PMJ (2.2) und MPMJ (3.2) werden jeweils alle noch vorhandenen Runs gelesen. Um den dabei verwendbaren Fan-In anzugeben, benötigt man Seiten- und Hauptspeichergröße. Schon bei derzeit gängigen PCs für den Privatbereich sind 512 MB Hauptspeicher durchaus üblich, bei einem Server zum Betrieb von Hochleistungsdatenbanksystemen können es durchaus auch mehrere GB sein. Auch wenn solche Server mehrere Transaktionen nebenläufig verarbeiten und dazu ihren Speicher aufteilen, kann man annehmen, daß für aufwendige Joins über vielen Relationen 128 MB Speicher für die Seiten zum Lesen der Runs zur Verfügung stehen (weiterer Speicher wird für die SweepAreas benötigt). Bei einer Seitengröße von 8 kB können damit 16384 Runs verarbeitet werden. Werden also maximal 2^{14} initiale Runs generiert, so können diese danach ohne zwischenzeitliche Merges

gejoint werden.

Werden zum Generieren der initialen Runs ebenfalls 128 MB Speicher verwendet und auf die r Relationen verteilt, so haben die je r in einem Schritt erzeugten Runs zusammen eben diese 128 MB Kapazität. Um weniger als 2^{14} initiale Runs zu erhalten, dürfen für $r = 10$ die Relationen zusammen eine Länge von 204,8 GB nicht überschreiten, für $r = 2$ kann sogar 1 TB verarbeitet werden.

Es kann also realistisch die Annahme getroffen werden, daß in der zweiten Phase des (M)PMJ nur ein Durchlauf der `while`-Schleife erfolgt, der Join also in einem Schritt fertig berechnet wird. Unter dieser Annahme ist es nicht sinnvoll möglich, die Phase 2 teilweise vorzuziehen und mit Phase 1 zu verzahnen, da erst nach deren Abschluß alle initialen Runs für den einen Merge-Schritt zur Verfügung stehen.

Für Sort-Merge-Bäume mit mehr als zwei Ebenen hingegen lohnt sich eine Modifikation des Algorithmus durchaus. Dies zeigt sich daran, daß höhere Knoten (bei realistisch hohem Fan-In) erheblich mehr Ergebnisse produzieren als alle ihre Kindknoten zusammen. Damit bietet sich eine Post-Order-Traversierung des Baumes an, die hohe Knoten so früh wie möglich behandelt. Wenn nun auf einer Ebene schon ein Knoten ausgewertet wurde und frühe Ergebnisse produzierte, so stellt sich die Frage, ob es danach noch lohnt, frühe Ergebnisse in verhältnismäßig kleinerer Zahl auf niedrigeren Ebenen zu berechnen, oder ob es günstiger ist, dort fortan nur einen Merge-Baum wie beim Sort-Merge-Join aufzubauen. Join-During-Merge Knoten gäbe es dann nur noch auf einer Flanke des Baumes. Diese Ideen sind Teil des Konzeptes, das in [DSTW02a] für den PMJ verfolgt wird. Dieses Kapitel konzentriert sich hingegen auf Sort-Merge-Bäume mit nur zwei Ebenen, da diese besonders praxisrelevant sind.

4.1.2 I/O-Aufwand

Das wesentliche Problem bei Joins, die nicht komplett im Hauptspeicher berechnet werden können, ist, daß Externspeicherzugriffe im Vergleich zu Hauptspeicherzugriffen extrem lange dauern, weshalb man sie in zeitbasierten Kostenmodellen als teuer bezeichnet. Auch wenn die Zugriffszeiten von Festplatten durch technische Weiterentwicklungen immer kürzer werden, so wurden auch Hauptspeicherzugriffe immer schneller, so daß nicht davon auszugehen ist, daß die Lücke in absehbarer Zeit geschlossen wird. Ein wesentliches Ziel bei der Entwicklung eines externen Joinverfahrens muß daher sein, die Zahl der Externspeicherzugriffe so gering wie möglich zu halten.

In der folgenden Betrachtung des I/O-Aufkommens werden nur die Externspeicherzugriffe betrachtet, welche explizit durch die Joinalgorithmen

entstehen, also nicht das ggf. ebenfalls auf dem Externspeicher erfolgende Lesen der Eingaberelationen oder Schreiben der Joinresultate.

In der Phase zur Erzeugung initialer Runs des Sort-Merge-Joins mit Join-During-Merge und des (M)PMJ werden jeweils alle Eingaberelationen komplett betrachtet und in Form der produzierten Runs auf den Externspeicher geschrieben. In einem Sort Merge-Baum mit nur zwei Ebenen (die Bedingungen dafür sind verschieden, siehe 2.7) werden diese danach im finalen Join-Merge-Schritt wieder gelesen. Allen diesen Verfahren ist dann also gemein, daß sie alle Tupel der Eingaberelationen je einmal auf den Externspeicher schreiben und einmal wieder lesen. Bei Änderungen am Verfahren sollten nun keine weiteren Zugriffe hinzukommen. Andererseits lassen sich auch nur sehr eingeschränkt Externspeicherzugriffe einsparen, da nur solche Tupel nicht für die Merge-Phase verfügbar bleiben müssen, zu denen zuvor bereits alle Ausgabetupel, in die sie eingehen, berechnet wurden (4.3.5.1).

4.1.3 Optimierungsziele

Weil die für den (M)PMJ nötigen Externspeicherzugriffe wie zuvor erläutert in ihrer Zahl nur sehr bedingt optimierbar sind, andererseits aber stark zum Aufwand beitragen, ist die Frage von Interesse, nach wie vielen der Externspeicherzugriffe wie viele Ergebnisse produziert werden. Sobald die initialen Runs generiert sind, werden in der zweiten Phase beim Einlesen und Verarbeiten der Runs alle Ergebnisse erkannt. Dies ist schon beim Sort-Merge-Join mit Join-During-Merge so. Der Vorteil beim PMJ liegt darin, daß schon in der Rungenerierungsphase Ergebnisse produziert werden. Eine wünschenswerte Verbesserung ist nun, deren Zahl möglichst groß zu gestalten. Abbildung 4.1 zeigt dies schematisch. Die Ergebnisse werden natürlich diskret und in den einzelnen Phasen nicht kontinuierlich produziert.

Werden durch eine Veränderung am Algorithmus in Phase 1 mehr frühe Ergebnisse produziert, so wird sich dadurch jedoch auch die dazu nötige Laufzeit erhöhen, schon alleine durch das Zusammensetzen der Ergebnistupel. Leider spart man diese Mehrarbeit in Phase 2 nicht wieder ein, da dort alle Ergebnisse erkannt werden. Der Aufwand zum Aussortieren der Duplikate wächst zudem bei manchen Duplikateliminierungsverfahren mit deren Zahl, also mit der der frühen Ergebnisse aus Phase 1. Die Erhöhung der Zahl früher Ergebnisse kann also eine Verlängerung der Gesamtlaufzeit zur Folge haben. Diesen Nachteil nimmt man aber generell beim (M)PMJ gegenüber dem Sort-Merge-Join mit Join-During-Merge in Kauf, wie Abbildung 4.2 schematisch zeigt.

Bei einer Erhöhung der Ergebniszahl in Phase 1 darf man daher nicht nur die Vermeidung von zusätzlichem I/O fordern, sondern muß auch den Zusatz-

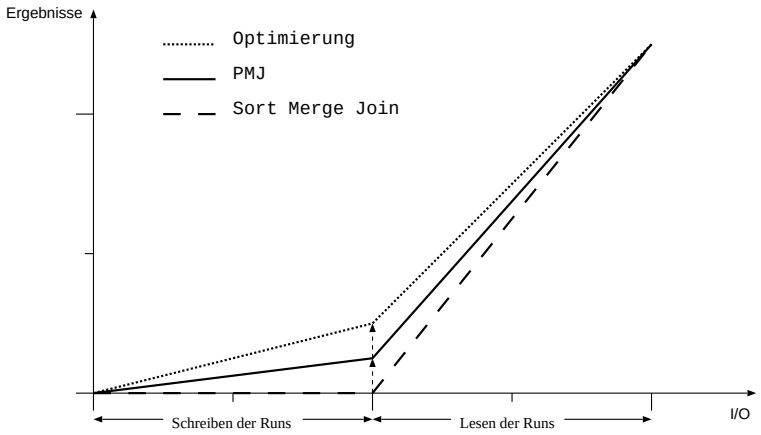


Abbildung 4.1: Optimierung der Ergebniszahl in Phase 1

aufwand speicherinterner Berechnungen in angemessenem Rahmen halten. Da die frühen Ergebnisse ja auch dem Verwendungsziel dienen, einem Benutzer im interaktiven Betrieb das vorzeitige Abbrechen des Joins nach für ihn hinreichender Berechnung von Schätzern oder Ergebnissen zu ermöglichen, ist eine moderate Erhöhung der Gesamtlaufzeit gerechtfertigt. Der Einsatz verschiedener Optimierungen kann auch von den Präferenzen des Benutzers abhängig gemacht werden, der entscheiden kann, welche Nachteile er für mehr oder schnellere frühe Ergebnisse in Kauf nehmen möchte.

4.1.4 Problemstellung

Zur Optimierung wird der MPMJ mit Joinprädikat P_{join} über r Relationen $R_1, \ldots, R_r$ betrachtet. Die Zahl der Tupel in den Relationen sei $r_i := |R_i|$, $i = 1, \ldots, r$. Der Hauptspeicher sei S Bytes groß, wobei jedes Tupel der Relation R_i zum Speichern t_i Bytes belege. Um allgemeiner auch den Join von Relationen mit Tupeln variabler Breite zu behandeln, kann man t_i auch als durchschnittliche Größe der Tupel in R_i betrachten, wenn die Tupelbreiten nicht zu sehr variieren.

4.2 Wahl der Verfahrensparameter

In Kapitel 3 wurden einige Freiheitsgrade von Algorithmus 3.1 noch nicht besprochen. Dabei handelt es sich um die Aufteilung des Speichers (4.2.1) zwischen den Stichproben bzw. den Runs der Relationen am Anfang der

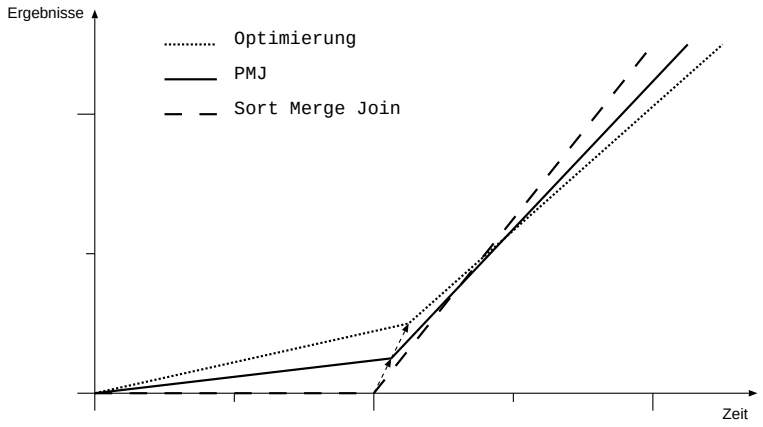


Abbildung 4.2: Erhöhte Laufzeit durch frühere Ergebnisse

`while` Schleifen von Phase 1 bzw. 2 sowie das zu verwendende Sortierverfahren (4.2.2).

4.2.1 Speicheraufteilung

4.2.1.1 Phase 1

Die Aufteilung des Hauptspeichers unter den Relationen sieht im Pseudocode folgendermaßen aus:

$$\text{Let } \hat{R}_j \subset R_j,\ j = 1, \ldots, r, \text{ where } \sum_{k=1}^{r} |\hat{R}_k| \leq M;$$

Formal stellt dies nur sicher, daß nicht mehr Speicher als verfügbar belegt wird.

4.2.1.1.1 Unterbelegung Es wird jedoch nicht gefordert, daß der gesamte Speicher genutzt wird. Da die Summe der Größen der Eingaberelationen nicht durch die Speichergröße teilbar sein muß, kann dort statt $\leq$ nicht $=$ verwendet werden, da sonst ggf. die Reste der Relationen am Ende nicht verarbeitet werden könnten. Es stellt sich die Frage, ob es Sinn macht, auch bei Verfügbarkeit hinreichend vieler Tupel nicht den gesamten Speicher auszunutzen. Da mit weniger Speicher auch nur weniger Tupel verarbeitet werden können, würde man dabei frühe Joinergebnisse verlieren. Der Vorteil wäre jedoch, daß die einzelnen Schritte schneller verarbeitet würden, wodurch dem Schätzer eine höhere Updaterate zukäme, allerdings wegen des Verlusts an Ergebnissen auch dessen Aussagekraft geschwächt würde.

Betrachtet man die Zeit zur Produktion des allerersten Joinergebnisses, so kann dies frühestens gefunden werden, wenn die Minima aller gewählten

$\hat{R}_i$ gefunden, diese also komplett betrachtet wurden. Wird viel Speicher verwendet, kann dies eine beträchtliche Anlaufzeit bedeuten. Zur Behandlung dieser Problematik wird in [DSTW02b] u. a. eine schrittweise Erhöhung des verwendeten Speichers in den ersten Schritten von Phase 1 vorgeschlagen. Dabei vergibt man aber unwiderruflich einen Teil der in Phase 1 produzierbaren Ergebnisse. Die Anlaufschritte würden dabei mehr I/O pro Ergebnis benötigen als bei vollständiger Speicherausnutzung von Anfang an. Daher bietet sich eine Unterbelegung das Speichers nur im ersten Schritt an, um die allerersten Ergebnisse schneller zu erhalten.

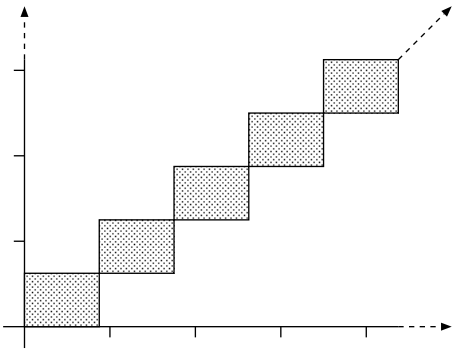
4.2.1.1.2 Verteilung Wird der gesamte Speicher verwendet, so stellt sich die Frage nach der Aufteilung unter den Relationen. Es sind Gewichte $w_i > 0,\ i = 1, \ldots, r$ zu wählen, die den Anteil der Mengen $\hat{R}_i$ an S festlegen, also $|\hat{R}_i| = \lfloor \frac{w_i \cdot S}{t_i} \rfloor \approx \frac{w_i \cdot S}{t_i}$. Da hier ein externes Joinverfahren Gegenstand der Untersuchung ist (S, $|R_i|$ etc. also sehr große Werte annehmen), werden in diesem Kapitel zum Erhalt der Übersichtlichkeit Rundungsfehler in Kauf genommen, auch wenn es natürlich keine halben Tupel oder Ergebnisse gibt. Offensichtlich muß $\sum_{i=1}^{r} w_i \leq 1$ gelten, zur vollen Ausnutzung des Speichers $\sum_{i=1}^{r} w_i = 1$.

Das generell optimale Verfahren gibt es auch hier nicht, vielmehr können nur verschiedene Möglichkeiten bezüglich der verschiedenen Ziele (geringe Gesamtlaufzeit, hohe Zahl früher Ergebnisse, schnelle Produktion der frühen Ergebnisse etc.) analysiert werden. Genauer gesagt kann man nur die Chance auf eine hohe Zahl früher Ergebnisse erhöhen. Dazu muß ein möglichst großer Teil von $R_1 \times \cdots \times R_r$ untersucht werden. Die Zahl der in einem Schritt mit Verteilung $w_1, \ldots, w_r$ untersuchten Tupel entspricht

$$|\hat{R}_1 \times \cdots \times \hat{R}_r| = \prod_{i=1}^{r} |\hat{R}_i| = \prod_{i=1}^{r} \frac{w_i \cdot S}{t_i} = \underbrace{S^r \prod_{i=1}^{r} \frac{1}{t_i}}_{const} \prod_{i=1}^{r} w_i \qquad (4.1)$$

Bei einer Selektivität des Joins von σ ergibt sich ein Erwartungswert von $\sigma \cdot S^r \prod_{i=1}^{r} \frac{w_i}{t_i}$ Ergebnissen. Da die Zahl der Ergebnisse proportional zur Suchraumgröße ist, werden im Folgenden Formulierungen wie *mehr Ergebnisse* verwendet, wenn eigentlich ein größerer Suchraum gemeint ist.

Insbesondere für $r \in \{2, 3\}$ kann man die Auswirkung der Verteilung gut an einem Diagramm veranschaulichen. Dazu werden auf den Achsen die Relationen entsprechend ihrer Größe $r_i t_i$ abgetragen. Die Speicheraufteilung entspricht dann dem Abtragen eines Runs auf jeder Achse, so daß die Gesamtlänge die Speichergröße nicht überschreiten darf. Die Fläche bzw. das Volumen des aus diesen Kanten gebildeten Rechtecks bzw. Quaders entspricht



$w_1 = \frac{7}{12},\ w_2 = \frac{5}{12}$

Abbildung 4.3: Beispiel für Speicheraufteilung in Phase 1

der Größe des untersuchten Bereichs des Kreuzproduktes der Relationen. Abbildung 4.3 zeigt dies beispielhaft für $r = 2$.

Gleichverteilung Will man die Größe des Suchraumes in einem Schritt optimieren, so muß wegen 4.1 der Term $\prod_{i=1}^{r} w_i$ maximiert werden, was unter den Nebenbedingungen so geschieht[1]:

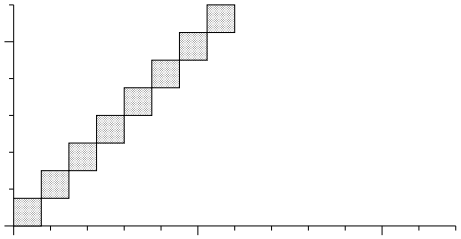
$$w_i = \frac{1}{r}\ ,\ \ i = 1, \ldots, r$$

Diese Wahl produziert die meisten Ergebnisse pro Schritt und I/O, ist also insbesondere zur Produktion vieler erster Ergebnisse geeignet. Setzt man sie in jedem Schritt ein, so tritt jedoch i. a. das Problem auf, daß irgendwann eine der Eingaberelationen R_a aufgebraucht ist, man also fortan in jedem Schritt $\hat{R}_a = \emptyset$ wählen muß, wie auch Abbildung 4.4 demonstriert. Dadurch werden dann aber gar keine frühen Ergebnisse mehr produziert, da $X \times \emptyset = \emptyset$ (siehe auch 4.3.1.1). Leider kann dies nicht kompensiert werden, indem man die aufgebrauchten Eingaben oder auch die daraus generierten Runs nochmals liest, da dies zusätzliches I/O verursachen würde, was aber gerade vermieden werden soll.

Die Gleichverteilung ist natürlich auch in Fällen der Unterbelegung günstig, da sie die optimale Ausbeute aus dem reduzierten Speicheranteil holt.

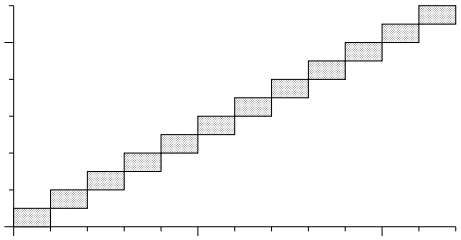
Anteilige Verteilung Um gleichmäßig und bis zum letzten Schritt frühe Ergebnisse produzieren zu können, müssen alle Relationen den An-

[1] wegen $(x + a)(x - a) < x^2$ für $a > 0$



$w_1 = \frac{1}{2}$, $w_2 = \frac{1}{2}$

Abbildung 4.4: Beispiel für Gleichverteilung



$w_1 = \frac{2}{3}$, $w_2 = \frac{1}{3}$

Abbildung 4.5: Beispiel für anteilige Verteilung

teil am Speicher bekommen, den sie an der gesamten Eingabe haben, also in jedem Schritt

$$w_i = \frac{r_i t_i}{\sum_{k=1}^{r}(r_k t_k)} \ , \ i = 1, \ldots, r$$

Haben die Eingaberelationen sehr unterschiedliche Größen, so liegt dies jedoch ggf. weit entfernt von der Gleichverteilung, so daß erheblich weniger Ergebnisse pro Schritt produziert werden. Auch für diese Verteilung ein Beispiel in Abbildung 4.5.

Ergebniszahloptimale Verteilung Rückblickend auf Abbildung 4.1 ist eine Optimierung der Gesamtzahl der frühen Ergebnisse in Phase 1 wünschenswert. Bei Beibehaltung einer gleichen Verteilung in allen Schritten können $min_{i=1}^{r} \frac{r_i t_i}{w_i S}$ Schritte bis zur vollständigen Konsumierung einer Eingaberelation[2] durchgeführt werden. Danach werden keine Ergebnisse mehr

[2]nämlich der Relation R_i, für die $\frac{r_i t_i}{w_i S}$ minimal ist

Verteilung	$w_1 = w_2$	w_3	$\frac{r_1\ t_1}{w_1} = \frac{r_2\ t_2}{w_1}$	$\frac{r_3\ t_3}{w_3}$	$\prod_{i=1}^{3} w_i$	Erwartungswert Ergebnisse
gleich	$\frac{1}{3}$	$\frac{1}{3}$	$300 \cdot 6^5$	$1200 \cdot 6^5$	$\frac{1}{27}$	86400
anteilig	$\frac{1}{6}$	$\frac{2}{3}$	$600 \cdot 6^5$	$600 \cdot 6^5$	$\frac{1}{54}$	86400
optimal	$\frac{1}{4}$	$\frac{1}{2}$	$400 \cdot 6^5$	$800 \cdot 6^5$	$\frac{1}{32}$	97200

$$r := 3,\ \sigma := \tfrac{1000}{6^{10}},\ S := 7776 = 6^5$$
$$t_1 := t_2 := t_3 := 10$$
$$r_1 := r_2 := 77760,\ r_3 := 311040 = 4\ r_1$$
$$\sigma\ S^{r-1} \prod_{i=1}^{r} \tfrac{1}{t_i} = 1$$

Tabelle 4.1: Anteilige Verteilung nicht optimal bei $r = 3$

erzeugt, insgesamt erhält man also als Erwartungswert für die Ergebniszahl:

$$\underbrace{min_{i=1}^{r} \frac{r_i\ t_i}{w_i\ S}}_{Zahl\ der\ Schritte} \cdot \underbrace{\sigma\ S^r \prod_{i=1}^{r} \frac{w_i}{t_i}}_{Ergebnisse\ pro\ Schritt} = \underbrace{\sigma\ S^{r-1} \prod_{i=1}^{r} \frac{1}{t_i}}_{const} \cdot min_{i=1}^{r} \frac{r_i\ t_i}{w_i} \cdot \prod_{i=1}^{r} w_i \tag{4.2}$$

Betrachtet man das Problem für den binären PMJ, also $r = 2$, und setzt die Nebenbedingung ein, so bleibt folgender Term zu maximieren:

$$min\left\{\frac{r_1 t_1}{w_1}, \frac{r_2 t_2}{1-w_1}\right\} \cdot w_1\ (1-w_1) = \begin{cases} r_1 t_1\ (1-w_1) & , \ \frac{r_1 t_1}{w_1} \le \frac{r_2 t_2}{1-w_1} \\ r_2 t_2\ w_1 & , \ \frac{r_1 t_1}{w_1} > \frac{r_2 t_2}{1-w_1} \end{cases}$$

Das Maximum wird am Schnittpunkt der Abschnitte für $\frac{r_1 t_1}{w_1} = \frac{r_2 t_2}{1-w_1}$ erreicht, woraus $w_1 = \frac{r_1 t_1}{r_1 t_1 + r_2 t_2}$ folgt. Dies entspricht der anteiligen Verteilung, die also beim PMJ die Maximalzahl früher Ergebnisse in Phase 1 liefert. Wie Tabelle 4.1 zeigt, läßt sich dieses Ergebnis allerdings schon für $r = 3$ nicht mehr auf den MPMJ übertragen, und das schon bei einem nicht ungewöhnlichen Verhältnis der Relationsgrößen zueinander von 1:1:4.

Während in Tabelle 4.1 noch Gleichverteilung und anteilige Verteilung gleich viele frühe Ergebnisse erwarten lassen, zeigt Tabelle 4.2, daß ebenfalls schon für $r = 3$ die anteilige Strategie auch erheblich weniger Ergebnisse erwarten lassen kann als die Gleichverteilung, hier bei einem Verhältnis der Relationsgrößen zueinander von 1:1:18.

Natürlich kann auch im Falle $r = 3$ die anteilige Verteilung der Gleichverteilung klar überlegen sein, z. B. bei einem Verhältnis der Relationsgrößen zueinander von 1:9:10.

Verteilung	$w_1 = w_2$	w_3	$\frac{r_1\ t_1}{w_1} = \frac{r_2\ t_2}{w_1}$	$\frac{r_3\ t_3}{w_3}$	$\prod_{i=1}^{3} w_i$	Erwartungswert Ergebnisse
gleich	$\frac{1}{3}$	$\frac{1}{3}$	$300 \cdot 60^3$	$5400 \cdot 60^3$	$\frac{1}{27}$	2400000
anteilig	$\frac{1}{20}$	$\frac{18}{20}$	$2000 \cdot 60^3$	$2000 \cdot 60^3$	$\frac{9}{4000}$	972000
optimal	$\frac{1}{4}$	$\frac{1}{2}$	$400 \cdot 60^3$	$3600 \cdot 60^3$	$\frac{1}{32}$	2700000

$$r := 3,\ \sigma := \tfrac{1000}{60^6},\ S := 216000 = 60^3$$
$$t_1 := t_2 := t_3 := 10$$
$$r_1 := r_2 := 2160000,\ r_3 := 38880000 = 18\ r_1$$
$$\sigma\ S^{r-1} \prod_{i=1}^{r} \tfrac{1}{t_i} = 1$$

Tabelle 4.2: Anteilige nicht immer besser als Gleichverteilung bei $r = 3$

Algorithmus 4.1 Berechnung der Verteilung mit größtem Suchraum

1: Let $M := \{r_i t_i \mid i \in \{1, \ldots, r\}\}$, $w := 1$;
2: **repeat**
3: Let $r_a t_a := max(M)$;
4: Let $w_a := min(\{\frac{r_a t_a}{\sum_{x \in M} x} \cdot w, \frac{1}{r-1}\})$;
5: Let $M := M \setminus \{r_a t_a\}$, $w := w - w_a$;
6: **until** $M = \emptyset$

Wie Gleichung 4.2 zeigt, ist zum Erreichen der optimalen Ergebniszahl der Term $min_{i=1}^{r} \frac{r_i\ t_i}{w_i} \cdot \prod_{i=1}^{r} w_i$ zu maximieren. Dessen Wert hängt vom Verhältnis der Produkte $r_i t_i$ zueinander ab. Umfangreiche computergestützte Tests[3] auch für größere Werte von r und verschiedenste Verhältnisse der Relationsgrößen haben ergeben, daß die anteilige Verteilung nur dann nicht den größten Suchraum liefert, wenn sie zur Folge hat, daß mindestens eines der w_i größer als $\frac{1}{r-1}$ ist, was im Falle $r = 2$ nicht auftreten kann.

Als optimale Verteilung bezüglich der zu erwartenden Ergebniszahl in Phase 1 erweist sich eine modifizierte anteilige Verteilung, bei der kein Anteil über $\frac{1}{r-1}$ liegt. Diese kann durch Algorithmus 4.1 mit Aufwand $O(r\ log\ r)$ berechnet werden.

Daß es sich dabei wirklich um die Verteilung handelt, die den größten Suchraum für Phase 1 liefert, ist leider nicht leicht zu zeigen. Der Beweis für $r = 3$ ist zwar erbracht, wird hier aber nicht ausgeführt, da er eher technisch und insbesondere lang ist.

[3]Programm und Ergebnisse siehe beiliegende CD

4.2.1.2 Phase 2

Die Aufteilung des Fan-Ins unter den Relationen sieht im Pseudocode so aus:

$$\text{Let } \hat{Q}_j \subset Q_j,\ j = 1, \ldots, r, \text{ where } \sum_{k=1}^{r} |\hat{Q}_k| \leq F;$$

Auch hier wird nur sichergestellt, daß nicht mehr Seiten als verfügbar benutzt werden, also nicht mehr Speicher als verfügbar verwendet wird.

Leider gibt es auch hier nicht die allgemein beste Strategie, wieder hängt deren Wahl von den Präferenzen des Benutzers ab. Ein Schritt in Phase 2 macht aus F Runs r, wenn von jeder Relation mindestens ein Run teilnimmt, was aber generell der Fall sein sollte, da es notwendig ist, um frühe Ergebnisse produzieren zu können. Werden lange Runs gewählt, so können mehr frühe Ergebnisse produziert werden, bei kürzeren entsteht dafür weniger I/O.

Ein Extrem stellt es dar, in jedem Schritt die im vorherigen produzierten Runs wieder zu verwenden. Dies entspricht jedoch einer Entartung des Algorithmus hin zu einem Block-Ripple-Join. Das andere Extrem ist, immer die kürzesten verfügbaren Runs auszuwählen, und versucht, dabei den I/O-Aufwand zu minimieren. Ähnlich dazu verarbeitet die Referenzimplementierung des binären PMJ die Runs mittels einer Warteschlange in Erzeugungsreihenfolge. Dies entspricht wiederum in etwa einer ebenenweisen Traversierung des Sort-Merge-Baumes.

Um die Ergebnisproduktion für den Fall, daß nicht alle initialen Runs in einem Schritt von Phase 2 verarbeitet werden können, zu optimieren, ist jedoch eine umfangreichere Umstellung der Verarbeitungsstruktur nötig, hierzu sei wieder auf [DSTW02a] verwiesen. An dieser Stelle muß jedoch noch die Auswirkung der Runverarbeitungsreihenfolge auf die Duplikateliminierung betrachtet werden.

4.2.1.2.1 Die Überdeckungsregel Werden mehrere Runs zu einem vereinigt, so ist danach nicht mehr nachvollziehbar, welches Tupel ursprünglich aus welchem Run kam. Wurden die Runs $R_1^*, \ldots, R_r^*,\ R_i^* \subset R_i$ einmal gemeinsam in `earlyJoin` verarbeitet, so wurden die Ergebnisse aus deren Kreuzprodukt schon produziert. Diese Information darf nicht verloren gehen, damit die Duplikateliminierung funktioniert. Da für jene nur zu erkennen ist, aus welchem Run die Tupel unmittelbar gelesen werden, muß daraus ein Rückschluß auf die gesamte Historie möglich sein. Dies kann man induktiv erreichen, indem man sicherstellt, daß Runs, die einmal gejoint wurden, auch fortan jeweils wieder gemeinsam verarbeitet werden. Dann treten die Ergebnisse nämlich jedesmal auf und werden als Duplikate erkannt. Im nächsten Schritt ist dann wieder entscheidbar, daß die Ergebnisse im vorherigen auftraten, also Duplikate sind usw.

Veranschaulicht man den Join der ausgewählten Runs wie in Abbildung

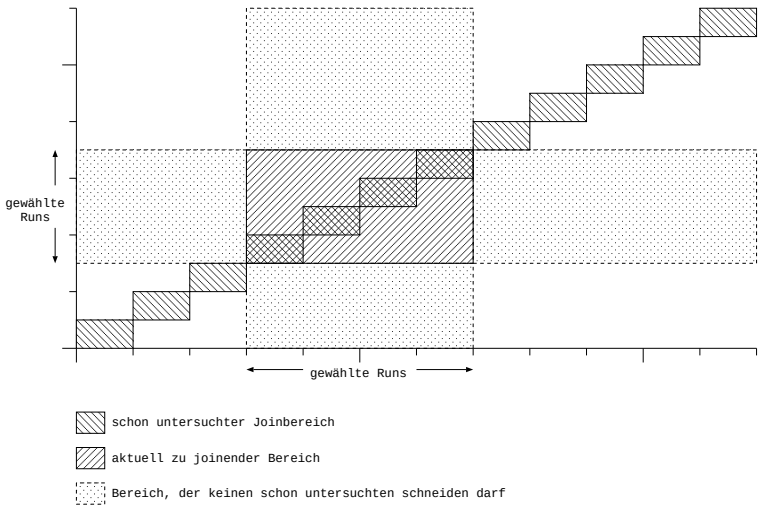


Abbildung 4.6: Überdeckungsregel

4.6, so muß der dabei betrachtete Teil des Kreuzproduktes die Bereiche, in denen früher mit diesen Runs Ergebnisse gesucht wurden, vollständig überdecken, daher die Bezeichnung *Überdeckungsregel*.

Das Diagramm zeigt auch, daß die Überdeckungsregel zur Folge hat, daß die Verteilung des Fan-Ins unter den Relationen i. w. der des Speichers in Phase 1 entsprechen muß.

Für den hier vorwiegend betrachteten Fall, daß alle initialen Runs in einem Merge-Schritt verarbeitet werden können, ist die Überdeckungsregel trivialerweise erfüllt, da dann in diesem Schritt $R_1 \times \cdots \times R_r$ komplett überdeckt wird.

4.2.1.2.2 Historie Um eine korrekte Duplikateliminierung auch unter Verletzung der Überdeckungsregel zu realisieren, muß über mehrere Merge-Schritte hinweg rekonstruierbar bleiben, in welchen Runs sich ein Tupel jeweils befunden hat. Dazu kann man jedem Tupel diesbezüglich eine Historie hinzufügen. Diese muß dann aber jeweils mit dem Tupel auf den Externspeicher geschrieben und davon gelesen werden, so daß zusätzlicher I/O-Aufwand entsteht.

4.2.2 Sortieralgorithmus

Während das globale Sortierverfahren beim (M)PMJ natürlich ein Sort-Merge-Join ist, ist nicht weiter festgelegt, mit welchem Verfahren die in-

itialen Runs im Hauptspeicher sortiert werden. Natürlich sollte ein möglichst schneller Sortieralgorithmus Verwendung finden. In diesem Kontext ist eine Unterscheidung von Verfahren wichtig, die die Sortierung erst komplett abschließen müssen, bis sie Ergebnisse liefern (z. B. der Insertion-Sort), und Algorithmen, die Ergebnisse schon vor Fertigstellung der Sortierung liefern können (z. B. der Heap-Sort). Einige Sortierprinzipien fallen je nach Implementierung in die eine oder andere Kategorie.

Produziert das eingesetzte Verfahren Ergebnisse noch während des Sortiervorgangs, so kann auch das Joinen beginnen, bevor die Sortierung abgeschlossen ist. Dies beschleunigt daher die Produktion erster Ergebnisse, weshalb dieses Vorgehen für den oder die ersten Schritte der Rungenerierung wünschenswert ist. Andererseits können Verfahren, die zunächst die Sortierung komplett abschließen, Gesamtlaufzeit einsparen, was in jedem Schritt greift, während die Wartezeit zwischen Auffinden des Minimums und kompletter Sortierung dafür nur einmal zu investieren ist. Daher empfiehlt sich ein frühzeitiger Übergang zum Verfahren mit kürzester Gesamtlaufzeit, wenn viele Runs zu generieren sind.

4.3 Modifikationen des MPMJ

4.3.1 Verringerung der Zahl initialer Runs

Wie schon in 2.7 aufgezeigt, hat der PMJ gegenüber dem Sort-Merge-Join mit Join-During-Merge insbesondere dann einen Nachteil, wenn sein I/O-Aufkommen jenem gegenüber dadurch größer ist, daß sich die Höhe des Sort-Merge-Baumes dadurch erhöht, daß der PMJ mehr initiale Runs erzeugt. Für diese Fälle sind nun Modifikationen wünschenswert, die die Zahl initialer Runs gerade so weit reduzieren, daß diese Erhöhung nicht auftritt.

4.3.1.1 Fallback

Bringt eine Veränderung an einem Verfahren Vor- und Nachteile mit sich, so erweist es sich oft als günstig, zunächst die Vorteile soweit wie möglich auszunutzen und dann vor dem Überwiegen der Nachteile auf das alte Verfahren umzuschwenken. Dies ist natürlich nicht immer möglich, der MPMJ eignet sich jedoch ausgezeichnet dazu. Hat man bei der Generierung initialer Runs in Phase 1 zunächst durch Speicheraufteilung und `earlyJoin` frühe Ergebnisse produziert, so kann man zu einem beliebigen Zeitpunkt die Produktion früher Ergebnisse einstellen und initiale Runs fortan wie beim Sort-Merge-Join erzeugen. Damit kann Replacement Selection wieder eingesetzt werden, und pro Hauptspeicherinhalt wird statt r nur noch ein Run erzeugt. Dadurch

reduziert sich die Zahl initialer Runs, und zugleich wird Berechnungaufwand eingespart. Allerdings werden nun keine frühen Ergebnisse mehr produziert, und der Schätzer kann nicht weiter verbessert werden.

Den richtigen Zeitpunkt für diesen Rückfall kann man nun, wenn man die Länge der Eingaberelationen kennt, so vorausberechnen, daß die Zahl der initialen Runs gerade keine Erhöhung des Merge-Baumes gegenüber dem Sort-Merge-Join mit Join-During-Merge zur Folge hat.

Ein natürliches Einsatzgebiet des Fallbacks ergibt sich bei anderen als der anteiligen Speicherverteilungsstrategie (4.2.1.1.2) in Phase 1. Ist irgendwann eine der Eingaberelationen komplett aufgebraucht und werden daher sowieso keine frühen Ergebnisse mehr produziert, so bringt ein Fallback nur Vorteile und sollte auf jeden Fall erfolgen.

Ein zusätzliches Einsatzgebiet des Fallbacks findet sich in der Interaktion mit dem Benutzer. Einer der Vorteile des PMJ ist ja, daß er Schätzer und frühe Ergebnisse liefert, ohne die Gesamtlaufzeit zur Berechnung aller Ergebnisse zu sehr zu erhöhen. Entscheidet sich der Benutzer nun noch in Phase 1, den Join komplett durchrechnen zu lassen, und wünscht dies nun so schnell wie möglich, so kann ein Fallback ausgelöst werden.

Die nach dem Fallback erzeugten initialen Runs haben neben ihrer Länge auch den Vorteil, daß die darin enthaltenen Tupel noch an keinem frühen Join beteiligt gewesen sind. In Phase 2 können daher Ergebnisse, bei denen mindestens eine Komponente aus einem solchen Run stammt, keine Duplikate sein. Aus diesem Grund fallen die Runs auch nicht unter die Überdeckungsregel und können in Phase 2 wann immer gewünscht der Weiterverarbeitung zugeführt werden. Um diese Vorteile ausnutzen zu können, werden die nach dem Fallback generierten Runs in der Referenzimplementierung des MPMJ getrennt verwaltet.

4.3.1.2 Replacement Selection Revival

Wie in 2.6.1 erläutert, führt der Einsatz von Replacement Selection (1.5.2.4.2) zu einer Verfälschung des Schätzers und kann daher nicht allgemein beim (M)PMJ eingesetzt werden. Ist der Benutzer aber nur an den frühen Ergebnissen und nicht am Schätzer interessiert, so kann Replacement Selection zur Generierung der initialen Runs in vollem Umfang eingesetzt werden. Die Funktion `earlyJoin` benötigt jeweils nur die Minima der Runs, also genau die Elemente, die bei Replacement Selection zum Herausschreiben ausgewählt werden.

Weiterhin eröffnet sich eine abgeschwächte Version des Fallbacks, bei deren Auslösung nur der Schätzer deaktiviert und dafür Replacement Selection aktiviert wird. Dies erhöht dann sogar die Zahl früher Ergebnisse in Phase 1.

4.3.2 Erhöhung der Zahl früher Ergebnisse in Phase 1

Wie vorstehend beschrieben kann die Zahl früher Ergebnisse in Phase 1 auf Kosten des Schätzers durch den Einsatz von Replacement Selection erhöht werden.

Das Grundprinzip des PMJ und des mehrdimensionalen PMJ ist es immer, ohne zusätzlichen I/O-Aufwand aber unter Investition von Hauptspeicheroperationen frühe Ergebnisse und Schätzer zu gewinnen. Bei einem sortierbasierten Verfahren entfällt ein wesentlicher Teil des Aufwandes natürlich auf die Sortierung. Bei Verwendung eines geeigneten Sortierverfahrens liegt ein initialer Run nach seiner Sortierung derart im Hauptspeicher, daß er danach beliebig oft mit linearem Aufwand sequentiell der Ordnung folgend gelesen werden kann. Diese Erkenntnis liefert eine Vielzahl neuer Optimierungsmöglichkeiten.

4.3.2.1 Mehrmalige Verwendung von Runs im Hauptspeicher

Beim (M)PMJ wird in Phase 1 jeweils ein Run jeder Relation gelesen, sortiert und dann der Join der Runs berechnet. Danach (bzw. dabei) werden alle Runs auf den Externspeicher geschrieben und tragen erst in Phase 2 wieder zur Produktion von Ergebnissen bei. Um jedoch nochmals einen ebenso großen Teil des Kreuzproduktes der Relationen nach Joinergebnissen zu durchsuchen, genügt es bereits, einen der Runs herauszuschreiben und durch neue Tupel der entsprechenden Relation zu ersetzen, diese zu sortieren und wieder den Join aller Runs im Hauptspeicher zu berechnen. Dabei können keine Duplikate entstehen, da sich die Ergebnisse des neuen Joins mindestens in der Komponente des ausgetauschten Runs von allen vorherigen unterscheiden.

4.3.2.1.1 Round Robin Der wohl naheliegendste Einsatz dieser Optimierung ist nun, jeweils reihum alle Runs im Hauptspeicher einmal auszutauschen, so daß man nach je r Schritten die gleichen Runs im Speicher hat, als wenn man einen Schritt des (M)PMJ durchgeführt hätte. Da jeder Schritt einen gleich großen Bereich B des Kreuzproduktes der Eingaberelationen betrachtet, erhöht sich so bei S Schritten des (M)PMJ die Suchraumgröße von SB auf $((S-1)r+1)B = rSB-(r-1)B$, i. w. also um den Faktor r. Anwendbar ist dieses Vorgehen auf beliebige Verteilungsverfahren (4.2.1.1.2) des Hauptspeichers unter den Relationen. Abbildung 4.7 zeigt den Einsatz am Beispiel der anteiligen Verteilung für $r = 3$.

4.3.2.1.2 Allgemeinere Austauschstrategien Verallgemeinert kann man in jedem Schritt eine Menge A mit $\emptyset \neq A \subseteq \{1,\ldots,r\}$ auswählen,

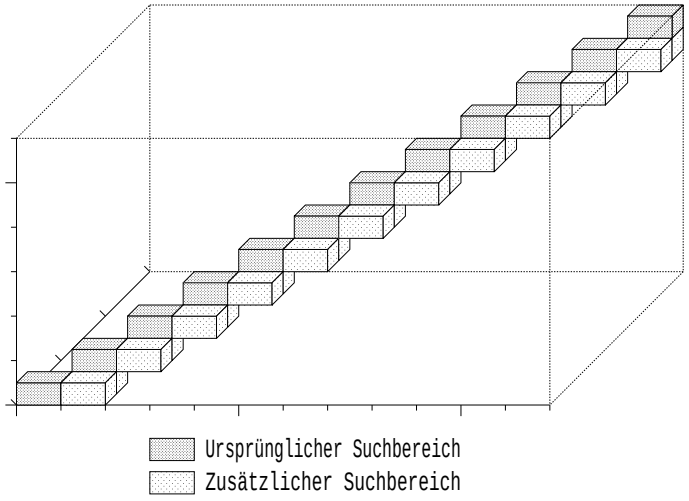


Abbildung 4.7: Round Robin Austausch von Runs

dann die Runs zu allen Relationen $R_i,\ i \in A$ im Speicher ersetzen und neu sortieren. Der Phantasie sind dabei auf den ersten Blick keine Grenzen gesetzt. Letztendlich stellt die Überdeckungsregel (4.2.1.2.1) allerdings eine Einschränkung dar, der aber leicht genüge getan werden kann, indem mindestens alle $F-r$ Schritte einmal $A = \{1, \ldots, r\}$ gewählt wird. Tauscht man die Runs zu einzelnen Relationen zu selten aus, so hat dies zudem die Folge, daß die dadurch lange im Speicher verbleibenden Stichproben jener Relationen überproportional in den Schätzer eingehen, dessen Qualität darunter leiden kann.

Auch die Aufteilung des Speichers unter den Relationen kann frei gewählt werden. In 4.2.1.1.2 werden die Aufteilungen unter der Voraussetzung analysiert, daß diese ursächlich für die Konsumierungsrate der Eingaberelationen sind. Die kann man nun aber auch steuern, indem man die Häufigkeit, in der die Relationen in A berücksichtigt werden, relativ zu ihrem Anteil am Speicher und ihrer Größe wählt. Abbildung 4.8 zeigt, wie dabei die Umkehrung der anteiligen Verteilung sogar mehr frühe Ergebnisse liefert.

4.3.2.1.3 Größe des Suchraumes Natürlich stellt sich die Frage, wie der Suchraum bei optimaler Strategie gewählt werden sollte. Dazu muß wieder bestimmt werden, wie viele Ergebnisse bei einer Aufteilung $w_1, \ldots, w_r$ des Speichers zu erwarten sind. Die Austauschstrategie sollte natürlich möglichst viele Schritte machen, da jeder Schritt Suchraum bringt. Also wird A immer einelementig gewählt. An Diagrammen wie beispielhaft in Abbildung 4.9 de-

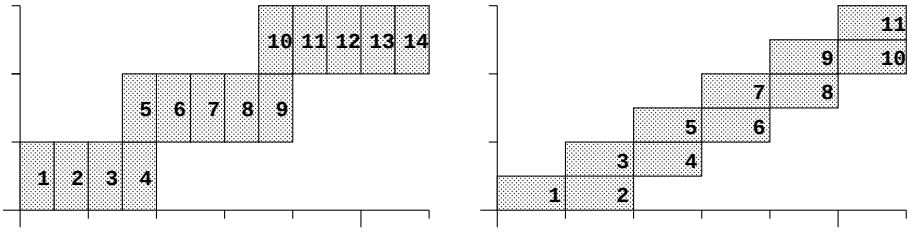


Abbildung 4.8: Auswirkung der Austauschstrategien

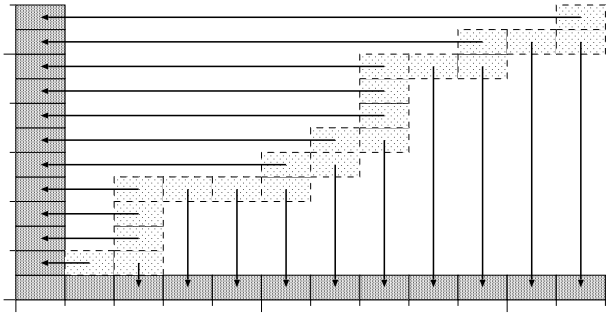


Abbildung 4.9: Ermittlung der Suchraumgröße

monstriert, kann man sich nun verdeutlichen, wie groß der Suchraum wird, indem man alle Rechtecke/Quader etc. zu den Achsen hin verschiebt.

Die Ergebniszahl pro Schritt wird also pro Relation so oft erreicht, wie der Run der Relation ausgetauscht werden kann. Hinzu kommt das Ergebnis des ersten Joins am Ursprung.

$$\underbrace{\sum_{i=1}^{r}(\frac{r_i\ t_i}{w_i\ S}-1)+1}_{Zahl\ der\ Schritte}\cdot\quad\underbrace{\sigma\ S^r\ \prod_{i=1}^{r}\frac{w_i}{t_i}}_{Ergebnisse\ pro\ Schritt} \tag{4.3}$$

Leider kann die Maximierung von Term 4.3 schon für $r = 2$ zu einem Schritt pro Tupel der größten Relation entarten. Da ein Run pro Schritt produziert wird, ist diese Wahl jedoch i. a. nicht akzeptabel. Zudem macht sich bei zu vielen Schritten auch der Mehraufwand durch das mehrmalige Verarbeiten bereits sortierter Runs im Hauptspeicher deutlich bemerkbar.

4.3.2.1.4 Gleichverteilung Als vernünftiger Kompromiß bietet sich daher an, die Ergebniszahl pro Schritt maximal zu halten, und die Austauschstrategie nur zur Anpassung an die unterschiedlichen Relationsgrößen zu ver-

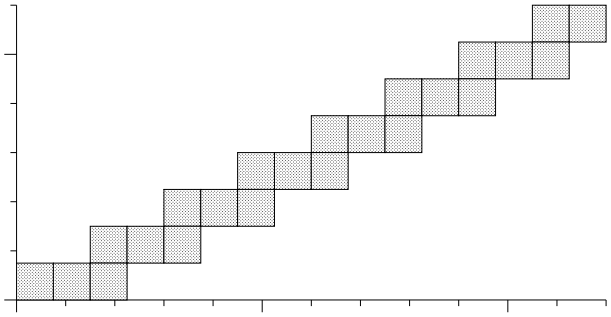


Abbildung 4.10: Gleichverteilung mit anteiliger Austauschstrategie

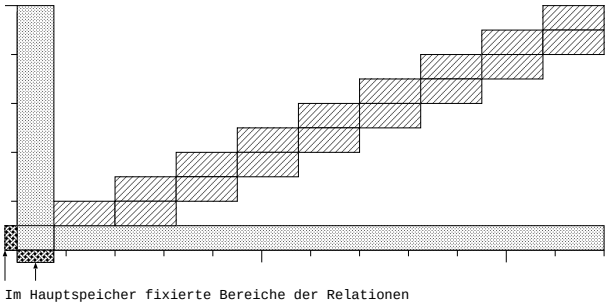


Abbildung 4.11: Teilfixierter Hauptspeicherbereich

wenden. Ein Beispiel dafür zeigt Abbildung 4.10.

4.3.2.2 Teilfixierung des Hauptspeichers

Unter der Voraussetzung, daß für Phase 1 des (M)PMJ eine Strategie anwendbar ist, die hinreichend wenige Runs generiert, ohne den verfügbaren Hauptspeicher komplett auszunutzen, ist eine geeignete Verwendung des freibleibenden Speichers gefragt. Es liegt nahe, dort ebenfalls Tupel zu deponieren und diese zur Produktion früher Ergebnisse zu verwenden. Also wird ein Anteil f des Hauptspeichers reserviert; für die Speicherverteilung muß nun also $\sum_{i=1}^{r} w_i \leq 1 - f$ gelten. Der Teil f wird nun in Anteilen $f_i,\ i = 1, \ldots, r$ unter den Relationen verteilt. Vor Phase 1 werden die zu fixierenden Speicherbereiche dann mit Tupeln aus den entsprechenden Relationen geladen und sortiert. Das Ergebnis sind zusätzliche Runs, die nie ausgetauscht werden. Daß nun ggf. zwei Runs pro Relation im Speicher sind, ist beim MPMJ kein grundlegendes Problem, da `earlyJoin` beliebig viele Runs pro Relation verarbeiten kann. Abbildung 4.11 veranschaulicht das Prinzip.

Wird für jede Relation ein Speicherbereich reserviert, so ergibt sich das Problem, daß Joinergebnisse entstehen, deren Komponenten sämtlich aus den fixierten Bereichen stammen. Diese werden dann in jedem Schritt von Phase 1 produziert, es treten also schon dort Duplikate auf. Diese können zwar gehandhabt werden, indem die Duplikateliminierung von `earlyJoin` auch in Phase 1 benutzt wird, was aus Effizienzgründen aber nicht zu empfehlen ist. Daher sollte stets $\exists i \in \{1, \ldots, r\} : \ f_i = 0$ gelten.

Frühe Ergebnisse in Phase 1 ziehen nun aber (fast) zwangsläufig Duplikate in Phase 2 nach sich. Für den Spezialfall $r = 2$ allerdings ergeben sich besondere Vorteile. Die Tupel in den fixierten Runs R_1^f und R_2^f befinden sich während der gesamten Abarbeitung der jeweils anderen Relation im Hauptspeicher und können damit auch mit allen diesen joinen. Nach Phase 1 sind also schon $R_1^f \bowtie_{P_{join}} R_2$ und $R_1 \bowtie_{P_{join}} R_2^f$ vollständig berechnet. R_1^f und R_2^f würden also in Phase 2 nur noch zur Produktion von Duplikaten beitragen und können stattdessen schon nach Phase 1 entfernt werden. Da sie dann nicht als Runs auf den Externspeicher geschrieben werden müssen, spart dies sogar I/O gegenüber dem (M)PMJ, allerdings nur maximal eine Hauptspeichergröße.

Für $r = 2$ stellen $R_1^f \bowtie_{P_{join}} R_2$ und $R_1 \bowtie_{P_{join}} R_2^f$ zudem den Join einer Stichprobe der einen Relation mit der jeweils kompletten anderen dar. Daraus kann am Ende von Phase 1 nochmals separat ein qualitativ guter Schätzer berechnet werden.

Für $r > 2$ gilt zwar auch, daß die Tupel in den Fixbereichen $R_1^f, \ldots, R_r^f$ schon in Phase 1 zusammen mit allen Tupeln der anderen Relationen gejoint werden, aber leider nicht mit allen Kombinationen davon, für $r = 3$ werden z. B. in Phase 1 $\bowtie_{P_{join}} (R_1^f, R_2^1, R_3^1)$ und $\bowtie_{P_{join}} (R_1^f, R_2^2, R_3^2)$ berechnet, aber eben nicht $\bowtie_{P_{join}} (R_1^f, R_2^1, R_3^2)$. Daher müssen auch die Runs R_i^f am Ende von Phase 1 auf den Externspeicher geschrieben und in Phase 2 weiterverarbeitet werden, wo sie die Duplikateliminierung zusätzlich verkomplizieren. Insbesondere wird bei mehr als einem Merge in Phase 2 die Überdeckungsregel verletzt.

Interessant ist ein Einsatz der Teilfixierung also insbesondere für $r = 2$. Unter Verwendung der Gleichverteilung wie in 4.3.2.1.4 beschrieben und Vergabe eines Anteils f des Hauptspeichers für den fixierten Bereich, stellt sich nun die Frage, ob diese Strategie mehr frühe Ergebnisse bringt, und wie dazu f optimalerweise unter den beiden Relationen zu verteilen ist. Sei dazu $z := f_1 - \frac{f}{2}$ und o.B.d.A $t_1 := t_2 := 1$. Dann errechnet sich ein Zugewinn von Suchraum in Phase 1 von $zS\ (r_2 - r_1) + (zS)^2$. Dieser lohnt sich nur bei Relationen mit merklich unterschiedlicher Größe, wobei der fixierte Speicherbereich dann komplett der kleineren Relation zugeteilt werden sollte. Dies

paßt gut zu der Forderung, daß zur Vermeidung von Duplikaten eines der f_i stets 0 sein sollte.

4.3.2.2.1 Ausnutzung von Skew Der eigentlich negativen Effekt des Skew kann auch Vorteile haben, da es dann Tupel gibt, die an überdurchschnittlich vielen Ergebnissen beteiligt sind. Hat ein Tupel nämlich global diese Eigenschaft, so wird es diese auch auf Stichproben zeigen. Daher kann man einen Teil des Hauptspeichers zunächst ungenutzt lassen und darin dann Runs der Tupel aufbauen, die in den einzelnen Schritten an besonders vielen Ergebnissen beteiligt waren. Dies erhöht die Zahl der frühen Ergebnisse, erschwert allerdings die Berechnung des Schätzers und bringt erheblichen Mehraufwand für die Duplikateliminierung in Phase 2, da genau ausgewertet werden muß, ab welchem Run welche hochselektiven Tupel in Phase 1 verwendet wurden.

4.3.3 Überschreitung des Fan-Ins

Die Zielsetzung dieser Optimierung ist die gleiche wie bei der Reduktion der Zahl initialer Runs, nämlich die Erhöhung des Merge-Baumes gegenüber dem Sort-Merge-Join mit Join-During-Merge zu verhindern. Nur setzt sie nicht in Phase 1, sondern erst in Phase 2 an, die mehr Runs verarbeiten soll, als der Fan-In erlaubt. Dazu wird ein Teil der Seiten halbiert und unter je zwei Runs verteilt. Beim Nachladen dieser Runs vom Externspeicher wird zwangsweise weiterhin eine ganze Seite gelesen, aber nur eine Hälfte im Hauptspeicher hinterlegt, beim nächsten Nachladen wird dann nochmal dieselbe Seite geholt und diesmal die andere Hälfte benutzt. Dies erhöht die Zahl der lesenden Externspeicherzugriffe für die betroffenen Runs um den Faktor 2. Maximal werden jetzt also alle Tupel einmal mehr gelesen. Dem gegenüber steht aber ggf. einmal mehr Lesen und Schreiben fast aller Tupel, wenn sich der Merge-Baum erhöht.

Da diese Optimierung den I/O-Mehraufwand nicht verhindert, sondern nur reduziert, sollte sie nur eingesetzt werden, wenn die Zahl der Runs am Ende von Phase 1 zwischen Fan-In und doppeltem Fan-In liegt, weil sie durch Benutzerwünsche oder vorher unbekannte Größe der Eingaberelationen nicht schon vorher geeignet verringert werden konnte.

4.3.4 Kombination von Optimierungen

Die meisten der in diesem Kapitel besprochenen Optimierungen lassen sich miteinander kombinieren. Dabei muß jedoch beachtet werden, daß einige Unverträglichkeiten auftreten können. So kann ein Fallback (4.3.1.1) auch bei

Einsatz einer Teilfixierung des Hauptspeichers (4.3.2.2) erfolgen, allerdings müssen dann die fixierten Bereiche auf jeden Fall als Runs herausgeschrieben und in Phase 2 verarbeitet werden, mit allen damit verbundenen Nachteilen.

4.3.4.1 Kurzzeitiges Replacement Selection

Die Idee der Wiedereinführung von Replacement Selection kann mit der des mehrmaligen Lesens sortierter Runs im Hauptspeicher kombiniert werden. Dabei wird zunächst wie gehabt der Hauptspeicher zunächst mit Runs gefüllt, diese werden sortiert und ihr Join berechnet, das Joinergebnis kann zur Berechnung des Schätzers benutzt werden. Statt nun einen oder mehrere Runs auszutauschen, wird jetzt Replacement Selection aktiviert, möglichst unter Ausnutzung der bestehenden Sortierungen. Beim Herausschreiben der auszutauschenden Runs wird nun nochmal der Join der Runs berechnet. Dabei muß eine Duplikateliminierung stattfinden, die nur noch Ergebnisse berücksichtigt, bei denen mindestens eine Komponente aus einem von Replacement Selection nachgelesenen Tupel besteht. Dieser Teil des Joins kann nicht in den Schätzer einfließen. Nun wird der Join der im Hauptspeicher befindlichen sortierten Runs berechnet (ebenfalls nicht für den Schätzer verwendbar), und diese werden ohne Replacement Selection herausgeschrieben. Ergebnis ist ein leerer Hauptspeicher, mit dem das Verfahren von vorne beginnen kann. Insgesamt liefert es mehr frühe Ergebnisse bei weniger initialen Runs, opfert aber einen Teil der Information für den Schätzer. Dieser Nachteil kann kompensiert werden, wenn Replacement Selection nur zum Herausschreiben eines der Runs benutzt wird. Dann ist nämlich jeweils bei Erreichen des leeren Hauptspeichers in den drei Joinschritten zusammen der Join von $r-1$ normalen und einer dank Replacement Selection längeren Stichprobe berechnet worden.

4.3.5 Ausnutzung von Metainformationen

Neben den Eingaberelationen und dem Joinprädikat sind dem Datenbanksystem oft noch weitere Informationen über den zu berechnenden Join bekannt. Diese können teilweise eingesetzt werden, um den Join wesentlich effektiver berechnen zu können.

4.3.5.1 Frühes Verwerfen von Tupeln

Gilt für ein Tupel $v_i \in R_i$, daß es maximal an der Produktion eines Joinergebnisses $(v_1, \ldots, v_r)$ beteiligt sein kann, so wird v_i nach dem Auftreten des Ergebnisses $(v_1, \ldots, v_r)$ nur noch zur Produktion von Duplikaten beitragen

können. Daher kann v_i sofort nach seiner Verwendung zur Produktion eines Ergebnisses verworfen werden und muß nicht mehr seinen Weg durch die Runs nehmen, wodurch ggf. sogar I/O eingespart wird.

Ist weitergehend sogar sicher, daß an der Produktion jedes Ergebnisses $(w_1, \ldots, w_r)$ ein Tupel mit der zuvor beschriebenen Eigenschaft beteiligt sein muß, und werden diese Tupel auch entsprechend entfernt, so kann die Duplikateliminierung völlig entfallen.

Diese zunächst ungewöhnlich erscheinenden Bedingungen sind z. B. bei Joins erfüllt, die der Auflösung einer Fremdschlüsselbeziehung einer relationalen Datenbank dienen. Dann werden sie nämlich von allen Tupeln der Relation erfüllt, in der der Schlüssel als Fremdschlüssel auftritt.

4.3.5.2 Self-Joins

Bei einem binären *Self-Join* wird der Join einer Relation mit sich selbst berechnet, also $\bowtie^{self}_{P_{join}}(R) := \{(t,t) \in R \times R \mid P_{join}(t,t)\}$. Für die Optimierung des MPMJ sind generell Joins $\bowtie_{P_{join}}(R_1, \ldots, R_r)$ von Interesse, bei denen mindestens eine Relation mehrmals vorkommt, für die also die Bedingung $\exists i, j \in \{1, \ldots, r\} : \; i \neq j \wedge R_i = R_j$ erfüllt ist. Die Idee ist nun, bei doppelt vorkommenden Eingaberelationen doppelte Arbeit einzusparen. Leider ist dies nicht allgemein möglich, da sich schon die Sortierung der Relation in den Vorkommen unterscheiden kann, wodurch nicht einmal $t \leq_{i,j} t$ erfüllt sein muß. Gilt allerdings unter den Ordnungen $\leq_{i,i} = \leq_{i,j} = \leq_{j,i} = \leq_{j,j}$, so muß die Relation nur einmal sortiert werden, z. B. als R_i. Der dadurch freiwerdende Hauptspeicher für die Runerzeugung und der Fan-In beim Mergen sollte nun neu verteilt werden. Dazu kann man einfach die ursprünglichen Verteilungsstrategien mit halbierten t_i und t_j verwenden.

Entnimmt nun die Funktion `earlyJoin` als zu verarbeitendes Tupel t eines aus R_i, so muß sie danach, statt erneut das Minimum über alle Folgen zu bestimmen, simulieren, sie hätte t jetzt als Minimum von R_j entnommen, wobei dabei allerdings das Herausschreiben in den Run unterdrückt wird.

Nach dem gleichen Prinzip kann man nun auch erwägen, die SweepArea für R_j einzusparen. Dazu muß der Join allerdings bezüglich R_i und R_j symmetrisch sein. Zudem darf ein Tupel erst dann in eine SweepArea mit Doppelfunktion eingefügt werden, nachdem es die SweepAreas nach Ergebnissen abgefragt hat, da sonst Duplikate auftreten könnten, indem ein Tupel mehrmals mit sich selber joint.

4.3.6 Blockweises Löschen in SweepAreas

Die zum Beweis der Korrektheit des PMJ in 2.3 nicht mehr benutzte Bedingung 2.4 aus [DSTW02b] motiviert eine Optimierung, wie sich zeigt, wenn man sie umschreibt:

$$\forall v \geq w : P_{rm}(u, w) \Rightarrow P_{rm}(u, v) \tag{4.4}$$

Dies sagt aus, daß ein Tupel u, das beim Auftreten von w aus seiner SweepArea gelöscht wird, auch von jedem in der Ordnung hinter w stehenden Tupel v gelöscht werden darf. Ist ein Löschen von Elementen einer SweepArea besonders aufwendig, etwa weil dafür alle deren Elemente betrachtet werden müssen, so kann man das Löschen auch zwischenzeitlich einstellen und nur gelegentlich ausführen lassen. Dies spart Löschaufwand, erhöht aber ggf. die Größe der SweepAreas und damit auch die Zeit für Anfragen daran.

Bedingung 4.4 ist auch für die Korrektheit dieser Optimierung nicht nötig, da alle Ergebniskandidaten gegen P_{join} getestet werden und nur sichergestellt werden muß, daß P_{rm} nicht zu viele Tupel löscht. Sie ist lediglich sehr empfehlenswert, um die Effizienz sicherzustellen. Allerdings ist zu beachten, daß diverse Optimierungen bei der Implementierung der SweepAreas nicht mehr anwendbar sind, wenn das Entfernungsprädikat P_{rm} nicht so scharf wie möglich gewählt ist, oder eben auch, wenn es durch blockweises Löschen nicht immer benutzt wird.

4.4 Implementierungsaspekte

Um die Effektivität der Optimierungen und ihr Zusammenwirken testen zu können, wurden viele davon in die Implementierung des MPMJ aufgenommen. Da diese aber i. a. nicht nur Vor-, sondern auch Nachteile haben, muß es auch möglich sein, sie zu deaktivieren. Zudem soll die Wahl der Verfahrensparameter flexibel möglich sein.

In der Implementierung wurden daher nur die nötigen Strukturen geschaffen. Die Steuerung des Verhaltens des MPMJ wird zu einem seiner Parameter. Beim Einsatz des MPMJ muß also eine Ablaufstrategie mit übergeben werden, die während der Laufzeit dessen Verhalten steuert. Dabei bekommt sie Rückmeldungen vom MPMJ, so daß auch flexible Strategien in Abhängigkeit von abgeschlossenen Teilberechnungen möglich sind. Zudem werden auch Strategien unterstützt, die zur Laufzeit vom Benutzer beeinflußt werden, der z. B. einen Fallback auslösen kann. Damit ein Einsatz des MPMJ auch ohne Neuimplementierung einer Strategie möglich ist, werden Referenzimplementierungen wichtiger Varianten zur Verfügung gestellt. Einige davon werden in Kapitel 5 für Experimente benutzt.

Zusammenfassung

PMJ und MPMJ lassen sich nach verschiedenen Kriterien optimieren, u. a. bezüglich der Zahl früher Ergebnisse, indem man die Verfahrensparameter, insbesondere die Speicherverteilung, geeignet wählt. Zusätzliche Veränderungen am Algorithmus bewirken einen weiteren Zugewinn an frühen Ergebnissen, wodurch der Einsatz noch attraktiver wird.

Kapitel 5

Experimente

Übersicht

Nach einigen Vorüberlegungen (5.1) und dem Hinweis auf XXL (5.2) werden in diesem Kapitel die Ergebnisse einiger Messungen (5.3) vorgestellt.

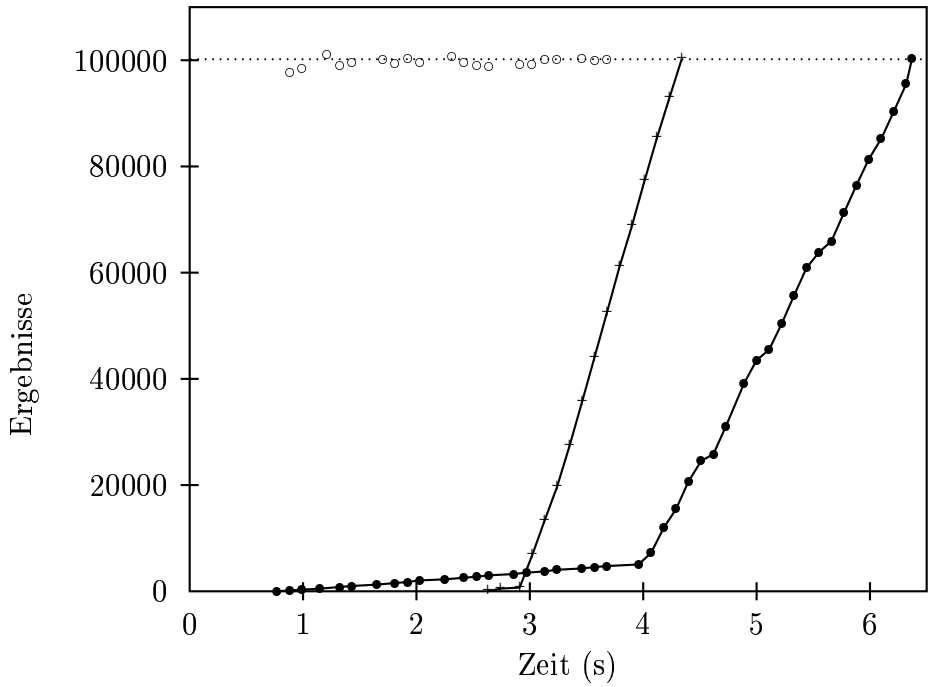
5.1 Vorüberlegungen

Kennt man die Eingaberelationen und die Strategie, welche den Ablauf des MPMJ steuert, also dessen Ablaufverhalten und insbesondere die Struktur des Sort-Merge-Baumes, so kann man für diesen vorausberechnen, bei welchem Knoten welcher Teil des Suchraumes untersucht wurde. Auch der I/O-Bedarf dafür kann bestimmt werden, insbesondere ist er bezüglich seiner Summe in Phase 1 in fast allen Fällen eine Invariante. Aus diesem Grunde werden hier Experimente vorgestellt, welche die Ergebnisproduktion des MPMJ in Abhängigkeit von der Laufzeit untersuchen, und die Ergebnisse in entsprechenden Graphen dargestellt.

5.2 XXL

Die Referenzimplementierung des MPMJ wurde mit Hilfe der komplett in Java programmierten XXL[1]-Bibliothek erstellt, die in [vdBBD+01] vorgestellt wird und insbesondere Implementierung und Vergleich verschiedener Verfahren in Datenbanksystemen erleichtern soll. Zur Implementierung des physischen Joinoperators MPMJ wird dabei i. w. auf Datentypen und die Cursoralgebra von XXL zurückgegriffen.

[1] eXtensible and fleXible Library



Sort-Merge-Join mit Join-During-Merge (+)
und MPMJ ($\cdot$) mit Schätzer (o)

Abbildung 5.1: Vergleich Sort-Merge-Join vs. MPMJ

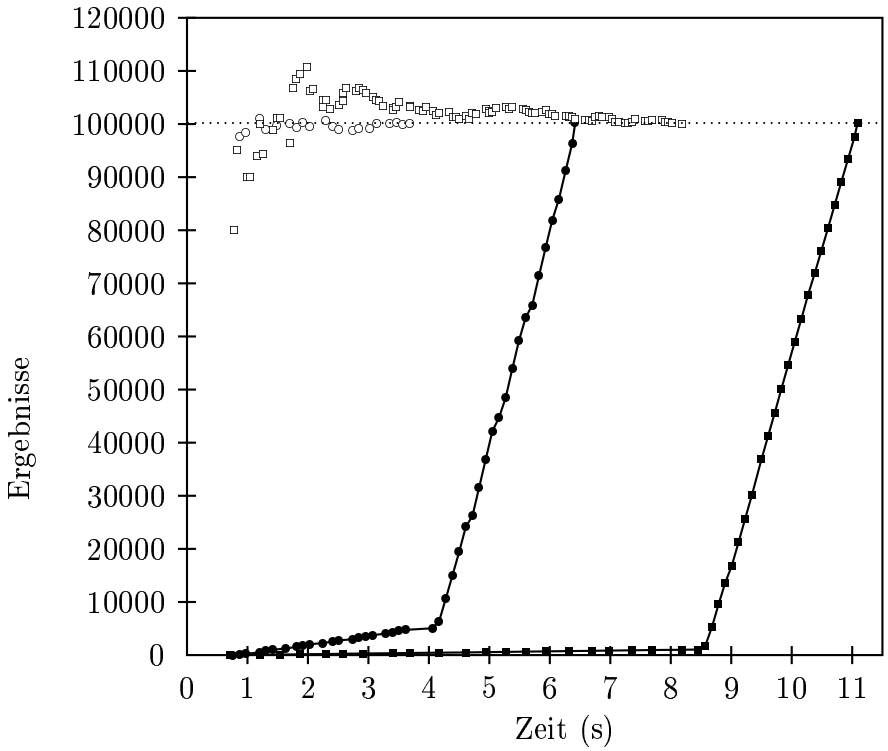
5.3 Messungen

5.3.1 Vergleich des MPMJ mit dem Sort-Merge-Join

Mit Hilfe einer Strategie, die gleich zu Beginn der Verarbeitung einen Fallback auslöst, kann die Implementierung des MPMJ dazu gebracht werden, sich wie der Sort-Merge-Join mit Join-During-Merge zu verhalten, welcher die Grundlage der Entwicklung des PMJ bildet. Beibehalten wird aber das Verfahren zur Ergebnisermittlung, also insbesondere die Verwendung von SweepAreas. Für $r > 2$ entsteht ein mehrdimensionaler Sort-Merge-Join mit den in 3.6 beschriebenen Anwendungsgebieten.

Abbildung 5.1 zeigt eine Vergleichsmessung mittels eines Equi-Joins, angewendet auf zwei je 100000 Elemente lange Folgen von Zufallszahlen[2] zwischen 1 und 100000, was eine erwartete Selektivität von $\sigma \approx \frac{1}{100000}$ ergibt.

[2]Durch festen Anfangszustand des Zufallsgenerators erfolgten alle Messungen jeweils mit identischen Folgen.



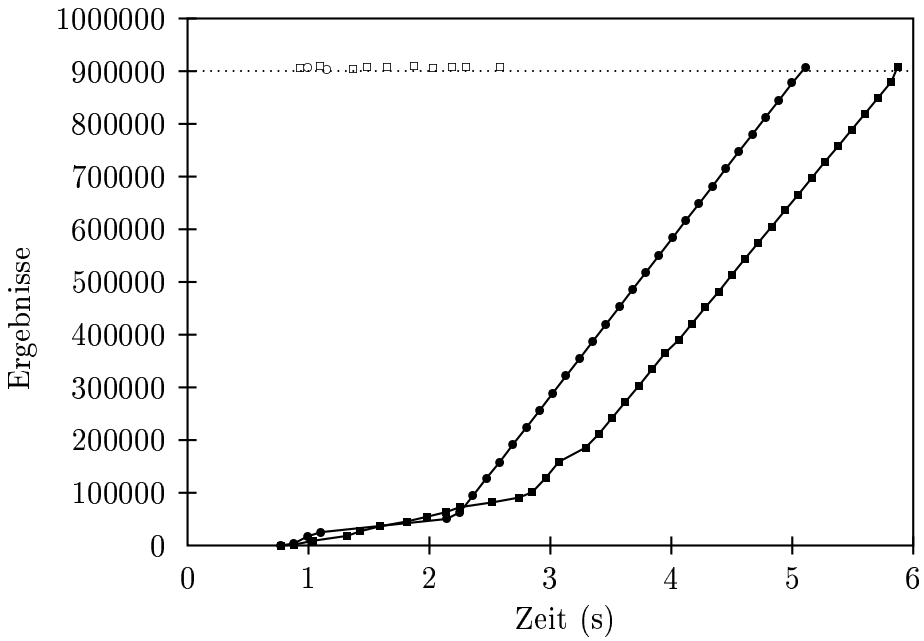
Ergebnisproduktion und Schätzer für Hauptspeicherkapazität
von 1000 (Quadrate) und 10000 (Kreise) Tupeln

Abbildung 5.2: Auswirkung der Speichergröße

Es wurde Speicher für 10000 Tupel bereitgestellt. Während der Sort-Merge-Join fast drei Sekunden bis zur Produktion von Ergebnissen benötigt, liefert der PMJ schon innerhalb der ersten Sekunde Ergebnisse und einen guten Schätzer. Allerdings verliert er den Vergleich bei der Gesamtlaufzeit klar.

5.3.2 Auswirkung der Speichergröße

Der Join aus 5.3.1 wurde hier noch einmal für den MPMJ wiederholt, diesmal unter Variation der Speichergröße zur Bestimmung ihrer Auswirkung. Wie Abbildung 5.2 zeigt, hat weniger Hauptspeicher gleich mehrere negative Folgen. Zunächst wird der Suchraum pro Schritt in Phase 1 kleiner, weshalb der Schätzer schlechter und die Gesamtzahl der frühen Ergebnisse kleiner wird. Zudem benötigt die Vielzahl der Schritte auch erheblich mehr Zeit. Dieser Versuch zeigt auch, warum eine absichtliche Unterbelegung des Hauptspeichers über viele Schritte wenig Sinn macht.



Ergebnisproduktion und Schätzer für
anteilige Verteilung (Quadrate) und Gleichverteilung (Kreise)

Abbildung 5.3: Gleichverteilung vs. anteilige Verteilung

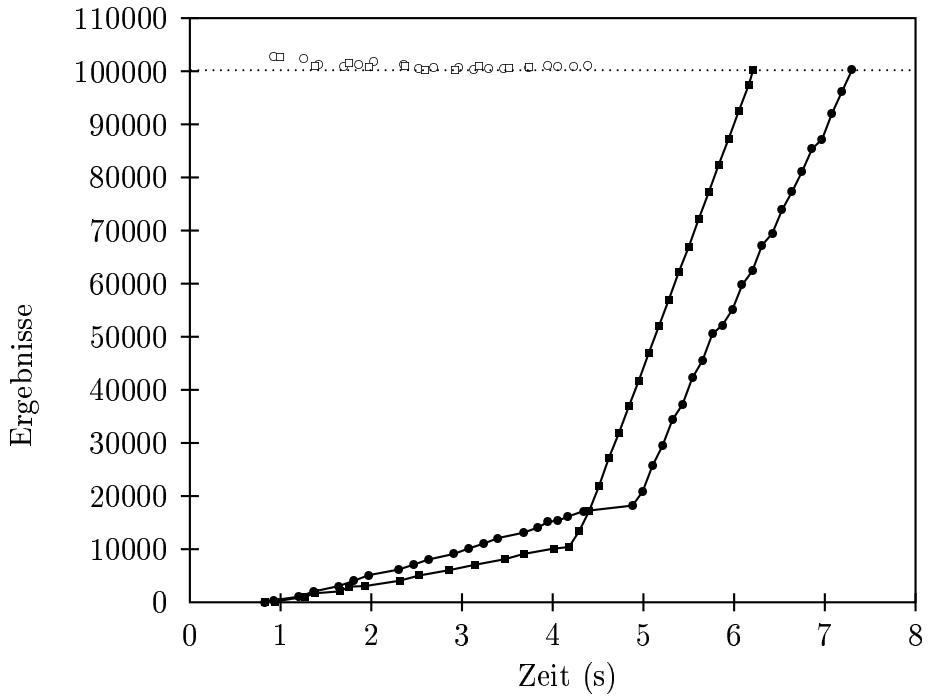
5.3.3 Auswirkung der Speicherverteilung

Abbildung 5.3 zeigt einen Vergleich zwischen der Gleichverteilung (4.2.1.1.2) und der für $r = 2$ bezüglich der Ergebniszahl in Phase 1 optimalen anteiligen Verteilung (4.2.1.1.2) des Hauptspeichers in Phase 1 des MPMJ. Die eine Relation wurde dabei mit 10000, die andere mit 90000 Zufallszahlen gewählt, da sich ein Unterschied zur Gleichverteilung nur bei verschieden großen Eingaberelationen ergeben kann; Hauptspeicher wurde für 10000 Tupel reserviert.

Es zeigt sich, daß zwar mehr Ergebnisse in Phase 1 produziert werden, dieser Vorteil jedoch fast durch die Mehrzahl an Schritten wieder verloren geht, die Ergebnisproduktionsrate in Phase 1 also nicht erheblich ansteigt.

5.3.4 Mehrmalige Verwendung von Runs im Hauptspeicher

Der Zugewinn an frühen Ergebnissen durch mehrmalige Verwendung von Runs im Hauptspeicher (4.3.2.1) zeigt sich in Abbildung 5.4. Betrachtet wird auch hier ein Equi-Join auf Zufallszahlenfolgen, diesmal beide mit Länge



Ergebnisproduktion und Schätzer ohne (Quadrate)
und mit (Kreise) mehrmaliger Verwendung von Runs im Hauptspeicher

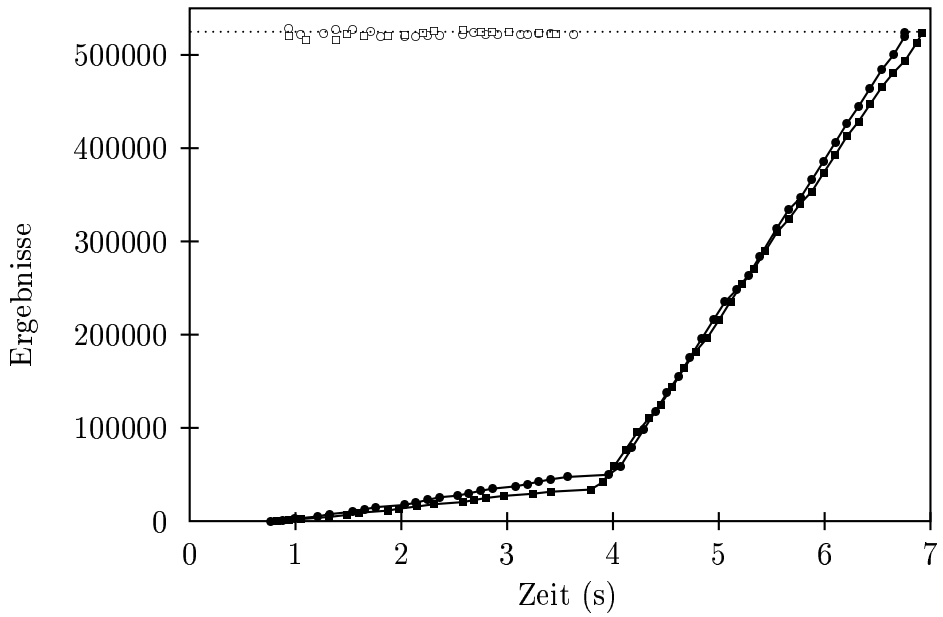
Abbildung 5.4: Mehrmalige Verwendung von Runs im Hauptspeicher

100000 bei Hauptspeicher für 20000 Tupel.

Auch hier bewahrheitet sich die Erwartung, daß der Zugewinn an frühen Ergebnissen durch eine längere Laufzeit von Phase 1 erkauft wird.

5.3.5 Teilfixierung des Hauptspeichers

Die in 4.3.2.2 vorgeschlagene Teilfixierung des Hauptspeichers wird in Abbildung 5.5 dargestellt. Die Eingaben wurden dabei mit 52500 bzw. 100000 Tupeln gewählt. Die 10000 Tupel Kapazität des Hauptspeichers wurden in der ersten Messung anteilig verteilt, bei aktivierter Teilfixierung wurden dann 2500 Tupel der kleineren Relation fest im Hauptspeicher positioniert. Die dadurch erreichte Gleichverteilung bei jedem Schritt von Phase 1 liefert den erwarteten Zugewinn an frühen Ergebnissen und hat sogar die kürzere Gesamtlaufzeit.



Ergebnisproduktion und Schätzer ohne (Quadrate)
und mit (Kreise) Teilfixierung des Hauptspeichers

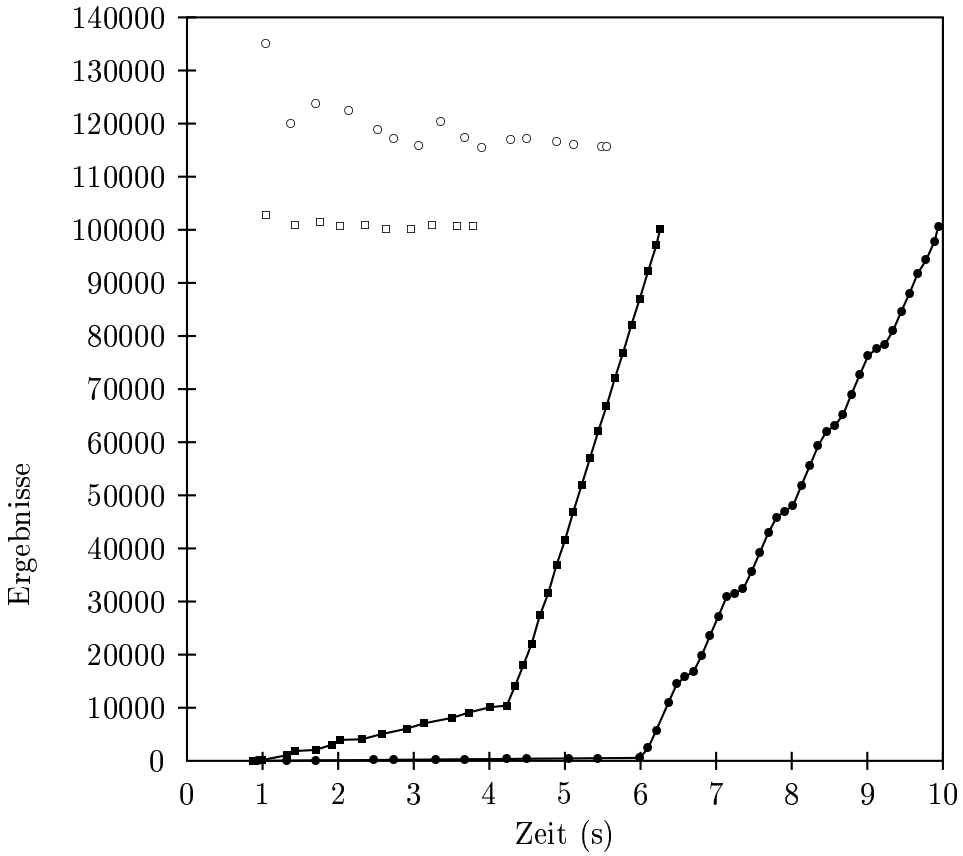
Abbildung 5.5: Teilfixierung des Hauptspeichers

5.3.6 Auswirkungen der Mehrdimensionalität

Abbildung 5.6 verdeutlicht eine der Schattenseiten der Mehrdimensionalität des MPMJ. In den beiden dort verglichenen Messungen werden ungefähr gleich viele Ergebnisse produziert. Der Unterschied liegt in der Zahl der aus jeweils 100000 Tupeln bestehenden Eingaberelationen, die einmal zwei und einmal drei beträgt. Im Falle $r = 3$ muß die Selektivität jedoch viel kleiner gewählt werden, um nur auf die gleiche Ergebniszahl zu kommen. Zusätzlich müssen sich dann drei statt zwei Runs den Hauptspeicher von 10000 Tupeln in Phase 1 teilen. Diese Faktoren führen dazu, daß die Zahl früher Ergebnisse drastisch geringer ist und die Qualität des Schätzers erheblich darunter leidet.

Zusammenfassung

Praktische Messungen belegen die gewünschte Eigenschaft von PMJ und MPMJ, frühe Ergebnisse zu liefern. Die vorgestellten Optimierungen erhöhen deren Anzahl wie erwartet, allerdings auf Kosten einer längeren Gesamtlaufzeit. Dennoch steigern sie die Produktionsrate früher Ergebnisse.



Ergebnisproduktion und Schätzer
für $r = 2$ (Quadrate) und $r = 3$ (Kreise)

Abbildung 5.6: Auswirkungen der Mehrdimensionalität

Kapitel 6

Fazit und Ausblick

Übersicht

Zum Abschluß der Arbeit wird ein Fazit (6.1) gezogen, und ein Ausblick (6.2) zeigt Themengebiete auf, die Gegenstand zukünftiger Untersuchungen werden könnten.

6.1 Fazit

In Zeiten der Informationsgesellschaft halten Datenverarbeitungssysteme in immer weitere Bereiche des Lebens Einzug. Dabei spielen Datenbanksysteme eine wichtige Rolle, um die zunehmenden Datenmengen effektiv verarbeiten zu können. Eine der wichtigsten und komplexesten Berechnungen ist dabei die von Joins, und obwohl Computer immer schneller werden und ihre Hauptspeicherkapazität wächst, wird die Bereitstellung effizienter externer Joinverfahren auf absehbare Zeit nötig sein.

Die Erzeugung früher Ergebnisse und Schätzer wurde zuerst für speicherinterne Joinverfahren entwickelt. Effektive Erweiterungen auf externe Joins erfolgten zunächst auf Basis von Hashing, wodurch deren Einsatz auf ein schmales Spektrum von Joinarten begrenzt ist. Dieses wurde durch die Entwicklung des PMJ [DSTW02b] auf sortierbasierte binäre Joins erheblich erweitert.

Die Entwicklung des MPMJ versucht, diesen Weg weiter zu verfolgen und weitere Joinarten für die Produktion früher Ergebnisse bei einem externen Joinverfahren zu erschließen. Das Konzept des PMJ wurde dazu auf mehrdimensionale sortierbasierte Joins erweitert und die gegenüber dem binären Fall hinzukommenden Probleme erläutert und eine Lösung vorgestellt.

Durch diverse Optimierungen der Produktion früher Ergebnisse und damit auch Schätzer wurde die Attraktivität des MPMJ (und damit auch des PMJ) für die Anwendung noch gesteigert. Leider muß beim Einsatz der Optimierungen zumeist eine moderate Erhöhung der Gesamtlaufzeit in Kauf genommen werden.

Obwohl der MPMJ die effektive Unterstützung möglichst vieler Joinarten verfolgt, bleibt doch die Einschränkung, daß er nur sinnvoll auf Joins anwendbar ist, bei denen eine gemeinsame Ordnung auf den Eingaberelationen definiert werden kann, bezüglich derer joinende Tupel nahe beieinander stehen. Da dies nicht immer der Fall ist, ist eine diesbezügliche Weiterentwicklung geboten (6.2.2).

6.2 Ausblick

6.2.1 Weitere Erhöhung der Effizienz

Beim MPMJ wird per Parameter die Anfragereihenfolge *qo* der SweepAreas für jede Relation festgelegt. Prinzipiell wäre es aber möglich, diese während der Laufzeit dynamisch zu ändern (siehe dazu auch [VNB02]). Dazu könnte man in den ersten Schritten von Phase 1 verschiedene Reihenfolgen testen und deren Effektivität messen, etwa anhand der Laufzeit oder der Selektivität der Pruningschritte. Allerdings müssen dazu dem MPMJ auch geeignete Pruningprädikate für die alternativen Reihenfolgen zur Verfügung gestellt werden. Das daraus resultierende Optimierungspotential wäre eine Betrachtung wert.

6.2.2 Erweiterung des Einsatzgebietes

Mit Hilfe eines sortierbasierten Verfahrens zur Berechnung mehrdimensionaler Joins sollten alle Kombinationen aus binären Joins effizient berechenbar sein, bei denen diese von sortierbasierten binären Joinverfahren unterstützt werden. Bei der Verkettung zweier binärer Joins wird die Ergebnismenge des ersten aber im zweiten ggf. umsortiert. Dieses Verhalten kann man nun auch verzahnen, also zunächst den Join zweier Runs berechnen, die Ergebnismenge umsortieren und mit einem dritten Run joinen. Leider muß die Ergebnismenge nicht in den Hauptspeicher passen, wenn es die zugrundeliegenden Runs tun. Die Wahl der Speicherverteilung unter den Runs sollte daher von der Selektivität des ersten Joins anhängen. Diese ist aber zunächst unbekannt; sie kann zur Anpassung der Verteilung aber ggf. während der Joinberechnung geschätzt werden, was nochmals die Vorteile eines Selektivitätsschätzers un-

terstreicht. Die Ausarbeitung dieses Prinzips sollte Gegenstand weiterer Untersuchungen sein.

Literaturverzeichnis

[AGPR99] Acharya, Swarup, Phillip B. Gibbons, Viswanath Poosala und Sridhar Ramaswamy: *Join Synopses for Approximate Query Answering.* In: *Proc. ACM SIGMOD Conf.*, 1999.

[BE77] Blasgen, Mike W. und Kapali P. Eswaran: *Storage and Access in Relational Data Bases.* IBM Systems Journal, 16(4):362–277, 1977.

[Ber02] Beringer, Jürgen: *Anfrageoptimierung in XXL.* Diplomarbeit, Philipps Universität Marburg, 2002.

[BSS98] Brinkhoff, Thomas, Ralf Schneider und Bernhard Seeger: *Skript zur Vorlesung GEO-Datenbanken*, 1998.

[Dit02] Dittrich, Jens-Peter: *Generische Joinverarbeitung am Beispiel des Similarity Join.* Doktorarbeit, Universität Marburg, 2002.

[DSTW02a] Dittrich, Jens-Peter, Bernhard Seeger, David Scot Taylor und Peter Widmayer: *On the efficiency of producing join results early.* 2002.

[DSTW02b] Dittrich, Jens-Peter, Bernhard Seeger, David Scot Taylor und Peter Widmayer: *Progressive Merge Join: A Generic and Non-Blocking Sort-Based Join Algorithm.* In: *VLDB*, 2002.

[HAR99] Hellerstein, Joseph M., R. Avnur und V. Raman: *Informix under CONTROL.* Submitted for publication, 1999. IBM Research Report RJ 10126.

[HH99] Haas, Peter J. und Joseph M. Hellerstein: *Ripple Joins for Online Aggregation.* In: *Proc. ACM SIGMOD Conf.*, Seiten 287–298, 1999.

[KE96] KEMPER, A. und A. EIKLER: *Datenbanksysteme*. Oldenbourg, 1996.

[Knu98] KNUTH, DONALD ERVIN: *The Art of Computer Programming*, Band 3. Addison Wesley, 2. Auflage, 1998.

[KS00] KOSSMANN, DONALD und KONRAD STOCKER: *Iterative Dynamic Programming: A New Class of Query Optimization Algorithms*. In: *ACM Transactions on Database Systems*, Band 25, Seiten 43–82, 2000.

[Lar02] LARSON, PER-ÅKE: *External Sorting: Run Formation Revisited*. Microsoft Research, 2002.

[LEHN02] LUO, GANG, CURT J. ELLMANN, PETER J. HAAS und JEFFREY F. NAUGHTON: *A Scalable Hash Ripple Join Algorithm*. In: *SIGMOD*, 2002.

[LGS02] LI, WEI, DENGFENG GAO und RICHARD T. SNODGRASS: *Skew Handling Techniques in Sort-Merge Joins*. In: *ACM SIGMOD*, 2002.

[MP01] MAMOULIS, NIKOS und DIMITRIS PAPADIAS: *Multiway Spatial Joins*. In: *ACM Transactions on Database Systems*, Band 26, Seiten 424–475, 2001.

[NBC+94] NYBERG, C., T. BARCLAY, Z. CVETANOVIC, J. GRAY und D.B. LOMET: *AlphaSort: A RISC Machine Sort*. In: *SIGMOD*, Seiten 233–242, 1994.

[NP85] NEGRI, MAURO und GIUSEPPE PELAGATTI: *Join during merge: An improved sort based algorithm*. Information Processing Letters, 21(1):11–16, 1985.

[Olk93] OLKEN, F.: *Random Sampling from Databases*. Doktorarbeit, University of California, Berkeley, CA, 1993. Available as Tech. Report LBL-32883, Lawrence Berkeley Laboraties, Berkeley, CA.

[SD89] SCHNEIDER, D.A. und D.J. DEWITT: *A Performance Evaluation of Four Parallel Join Algorithms in a Shared-Nothing Multiprocessor Environment*. In: *SIGMOD*, Seiten 110–121, 1989.

[See97] SEEGER, BERNHARD: *Skript zur Vorlesung Datenbanksysteme I*, 1997.

[See98] SEEGER, BERNHARD: *Skript zur Vorlesung Datenbanksysteme II*, 1998.

[UF99] URHAN, TOLGA und MICHAEL J. FRANKLIN: *XJoin: Getting Fast Answers From Slow and Busty Networks.* Technischer Bericht CS-TR-3994, UMIACS-TR-99-13, 1999.

[UF00] URHAN, TOLGA und MICHAEL J. FRANKLIN: *XJoin: A Reactively-Scheduled Pipelined Join Operator.* IEEE Data Engeneering Bulletin, 23(2):27–33, 2000.

[vdBBD+01] BERCKEN, JOCHEN VAN DEN, BJÖRN BLOHSFELD, JENS-PETER DITTRICH, JÜRGEN KRÄMER, TOBIAS SCHÄFER, MARTIN SCHNEIDER und BERNHARD SEEGER: *XXL - A Library Approach to Supporting Efficient Implementations of Advanced Database Queries.* In: *VLDB*, Seiten 39–48, 2001.

[VNB02] VIGLAS, STRATIS D., JEFFREY F. NAUGHTON und JOSEF BURGER: *Maximizing the Output of Multi-Join Queries over Streaming Information Sources.* In: *Proceedings of the 28th VLDB Conference*, Hong Kong, China, 2002.

Erklärung

Hiermit versichere ich, daß ich diese Arbeit selbständig verfaßt und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Marburg, im Dezember 2002