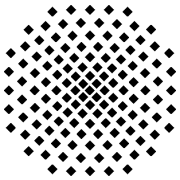


Universität Stuttgart

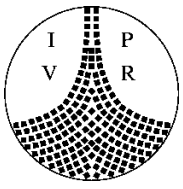
Fakultät Informatik

Studiengang:	Informatik
Prüfer:	Prof. Dr.-Ing. B. Mitschang
Betreuer:	Ralf Rantzau
Beginn am:	24.01.2001
Beendet am:	23.07.2001
CR-Nummer:	H.2.4

Diplomarbeit Nr. 1917

Analyse des Stream-Join Operators und Konzepte zu seiner Integration in ein relationales Datenbanksystem

Bernd Watzal



Institut für Parallele und Verteilte Höchstleistungsrechner (IPVR)

Abteilung Anwendersoftware (AS)

Inhaltsverzeichnis

Tabellenverzeichnis

1. Einleitung

Stream-Join ist ein unärer Operator für relationale Datenbanksysteme. Seine algorithmische Vorgehensweise ist vorgegeben und in [1] und [3] besch rieben. Aus dieser nicht formalen Beschreibung von Stream -Join lassen sich zwei algebraische Operatoren ableiten, die sich algorithmisch fast gleichen. Sie werden durch die erweiterte Bezeichnung als equivalence bzw. sequence Stream -Join voneinander unterschieden. Der Operator equivalence Stream -Join, kurz Equi -Stream-Join, bearbeitet Allquantifizierung in Anfragen. Der Operator sequence Stream -Join, kurz Sequi-Stream-Join, analysiert Sequenzen unter Einsatz von Allquantifizierung..

Grundsymbol von Stream -Join ist das Zeichen $\triangleright$. Es bringt grafisch zum Ausdruck, dass beide Operatoren ihre Eingaberelation schrittweise auf eine kleine Ergebnismenge reduzieren. Das Symbol $_{=}\triangleright$ steht für Equi -Stream-Join, weil er interne Gruppierungen vornimmt und Gruppierungen auf Gleichheitsrelationen beruhen. Sequi -Stream-Join wird mit $_{\geq}\triangleright$ bezeichnet, da jeder Sequenz eine Ordnung zugrunde liegt, die der Operator durch eine interne Sortierung berücksichtigt.

Auf die Begriffsdefinitionen in diesem Kapitel folgt eine Einführung zu Allquantifizierung in Anfragen, die eine Klassifizierung der verschiedenen Anfragetypen einschließt. In Kapitel 3 wird mit Hash -Division ein bekannter und effizienter Ansatz zur Bearbeitung von Allquantifizierung vorgestellt. Kapitel 4 beschreibt den neuen Operator Equi -Stream-Join zur Bearbeitung von Allquantifizierung. Seine Implemen - tierung und die damit verbundenen Eigenschaften des Operators werden einschließlich Aufwandsabschätzungen angegeben. Kapitel 5 stellt die beiden Ansätze Hash -Division und Equi -Stream-Join unter dem Gesichtspunkt einer optimalen Ausführung von Allquantifizierung in Anfragen einander gegenüber. Der Operator Sequi -Stream-Join wird in Kapitel 6 einschließl ich Implementierung beschrieben. Seine Einsatz - möglichkeiten zur Analyse von Sequenzen werden angegeben. Kapitel 7 stellt die Anfragesprache Sorted Relational Query Language SRQL zur Verarbeitung von Sequenzen vor und integriert die Konzepte Sequi-Stream-Join und SRQL.

1.1. Begriffsdefinitionen

In der Diplomarbeit werden Begriffe aus der Relationenalgebra, der mathematischen Algebra und dem SQL-Kontext verwendet. Darüber hinaus werden spezifische Begriffe bezüglich des Operators Stream-Join benötigt. Dieses Kapitel definiert die zentralen Begriffe und passt sie gegebenenfalls dem Kontext des Stream-Join an.

Belegung
Gegeben sei eine Relation R mit Attributen A. Eine Belegung ist eine Funktion, die jedem Tupel von R für jedes Attribut A einen Wert zuordnet. Die Belegung bzw. den Wert des Tupels t im Attribut $attr \in A$ gibt man mit $t.attr$ an. Haben zwei Tupel t_1, t_2 die gleiche Belegung in einem gemeinsamen Attribut a_G (d.h. $t_1.a_G = t_2.a_G$), sagt man auch, t_1 und t_2 stimmen in a_G überein.

Gruppierung
Eine *Gruppierung* bzw. eine *Gruppierung nach Attributen* A_G ist eine vollständige, disjunkte Partitionierung einer Relation R. Zwei Tupel werden derselben Partition zugeordnet, wenn ihre Belegungen in den Attributen A_G übereinstimmen. A_G werden auch Gruppenattribute genannt.

Gruppe
Eine Gruppe bezeichnet eine durch eine Gruppierung entstandene Partition. Sie zeichnet sich dadurch aus, dass alle in ihr enthaltenen Tupel in den Gruppenattributen übereinstimmen.

Stream
Im Zusammenhang mit Stream-Join wurde der Begriff Stream zur Bezeichnung einer Gruppe mit bestimmten Eigenschaften eingeführt. Ein *Stream* ist eine Untergruppe, also eine Gruppe innerhalb einer übergeordneten Gruppe. Alle in einem Stream enthaltenen Tupel stimmen in den Attributen A_D überein. A_D heißen Divisor-Attribute. Streams partitionieren eine Gruppe disjunkt und vollständig. Je nach Algorithmus werden sie innerhalb ihrer Gruppe beliebig (equivalence Stream-Join) oder nach A_D sortiert (sequence Stream-Join) angeordnet.

Dividenden und Sequenzen
Im Zusammenhang mit Stream-Join werden zwei weitere Begriffe für (Unter-) Gruppen eines anderen Typs verwendet, die orthogonal zu den Streams verlaufen. Sie heißen *Dividenden* bei Equi-Stream-Join und *Sequenzen* bei sequence Stream-Join. Alle Tupel eines Dividenden bzw. einer Sequenz stimmen in den Attributen A_R überein. Es gilt $A_R \cap A_S = \emptyset$. Ein Dividend bzw. eine Sequenz ist in einem Stream enthalten, wenn in dem Stream mindestens ein Tupel existiert, das zu dem Dividenden bzw. zu der Sequenz gehört.

Allquantifizierung
Der Allquantor $\forall u \in D: P$ ist eine Wahrheitswertfunktion auf einer Menge D, die das Prädikat P für alle Elemente d der Menge D prüft. Die Anwendung des Allquantors auf D wird als Allquantifizierung über der Menge D bezeichnet. Der Allquant or kann mit dem Existenzquantor kombiniert werden, um Bedingungen der Art $\forall d \in D\ \exists r \in R: P$ auszudrücken. Ein solcher Ausdruck wird als Allquantifizierung von R über D bezeichnet.

Relationale Division
Relationale Division $\div_{P_D,\, A_Q}$ ist ein Operator der Relationen algebra zur Ausführung der Allquantifizierung einer Relation $R(A_Q, X)$ über einer Relation $D(Y)$ mit Prädikat P_D und Ausgabeattributen A_Q. Man bezeichnet R als Dividendenrelation, D als Divisor, P_D als Divisionsprädikat, die Ergebnisrelation $R \div_{P_D,A_Q} D$ als Q uotient Q und A_Q als Quotien - tenattribute.

$$R \div_{P_D,A_Q} D = \{q \mid q \in \pi_{A_Q}(R) \ \wedge\ (\{q\} \times D) \subseteq (\pi_{A_Q}(R) \bowtie_{P_D} D)\}$$

Das kartesische Produkt eines Tupels q mit dem Divisor erzeugt eine Menge D_q mit gleicher Kardinalität und den Attributen (A_Q, Y). Der Join erzeugt eine Menge D_R, die ebenfalls die Attribute (A_Q, Y) besitzt. Wenn die Partition $\sigma_{q=A_Q}(D_R)$ alle Tupel aus D_q enthält, ist D_q eine Untermenge von D_R und das Tupel q ist in dem Quotienten enthalten. Von der Partition $R(q) = \sigma_{q=A_Q}(R)$ hängt es ab, ob der Kandi dat q in den Quotienten übernommen wird. $R(q)$ wird aus diesem Grund als Dividend (von Kandidat q) bezeichnet.
In der Literatur ist die relationale Division meist ohne Prädikat und Quotientenattribute definiert. Man geht von zwei Relationen $R(A_Q, B)$ und $D(B)$ aus, die in den Attributen B übereinstimmen, und definiert sie als reine Mengenoperation:

$$R \div D = \{q \mid q \in \pi_{A_Q}(R) \wedge (\{q\} \times D) \subseteq R\}$$

Im Zusammenhang mit der Allquantifizierung ist diese Definition zu unflexibel, da sie das Divisionsprädikat implizit auf $P_D = (R.B = D.B)$ festlegt und die beteiligten Relationen außer den Attributen A_Q und B keine weiteren Attribute enthalten dürfen.

Allgemeine Schreibweisen
Seien $R(A, B)$ und $S(A)$ Relationen eines relationalen Datenbankschemas mit den Attributmengen $A = \{a_1, a_2\}$ und $B = \{b_j\}$, so bezeichnet

- $r \in R$ ein Tupel dieser Relation
- $r.A$ die Belegung des Tupels r in den Attributen A
- $R.A$ $= \{r.A \mid r \in R\}$ die Menge der Belegungen der Relation R in den Attributen A. In schematischen SQL -Anfragen wird die Schreibweise zur abgek ürzten Darstellung von Prädikaten verwendet. Das Prädikat $(R.A = S.A)$ entspricht ausgeschrieben $(R.a_1 = S.a_1\ AND\ R.a_2 = S.a_2)$.
- $A(R)$ die Attributmenge A mit Verweis auf ihre Herkunftsrelation R. Diese Schreibweise wird benötigt, um gleiche Attributmengen v erschiedener Relationen R und S voneinander unterscheiden zu können.
- $P(r,s)$ ein Prädikat P, das sich auf Tupel r und s der Relationen R und S bezieht. Es enthält also Attribute aus beiden Relationen.

2. Allquantifizierung in Anfragen

Von der Prädikatenlogik her betrachtet ist der Allquantor eine Wahrheitswertfunktion. Er wird verwendet, um ein Prädikat für alle Elemente einer Menge zu prüfen. Die einzelnen Wahrheitswerte, die jedem Element durch das Prädikat zugeordnet werden, ergeben mit dem logischen UND v erknüpft den Wahrheitswert des Allquantors. In diesem Sinne stellt er eine Verallgemeinerung der aussagenlogischen Konjunktion dar.

Viele in einer natürlichen Sprache formulierte Aussagen enthalten einen Allquantor. Wenn der Chef eines Unternehmens sagt, **allen** Mitarbeitern sei das Gehalt pünktlich ausbezahlt worden, hat er bereits einen Allquantor in seiner Behauptung verwendet. Dieses einfache Beispiel verdeutlicht, wie grundlegend der Zusammenhang ist, den der Allquantor zwischen einer Menge (von Mitarbei tern) und einer Teilaussage (Gehalt pünktlich ausbezahlt) beschreibt. Hierin liegt seine allgemeine Bedeutung und Wichtigkeit begründet.

Ein relationales Datenbanksystem soll mindestens das Auswahlvermögen bieten, dass in seiner Anfragesprache Aussagen der Prädikatenlogik erster Ordnung formuliert werden können. Trotz dieses Anspruchs und trotz der Bedeutung des Allquantors ist er in SQL bislang nicht verfügbar. Es war ursprünglich vorgesehen, ihn durch das Schlüsselwort FOR ALL in den Standard von 1999 auf zunehmen (siehe [11]), was aber letztlich nicht erfolgte. In SQL gibt es lediglich das Schlüsselwort ALL. Mit ALL kann man nur einfache Allquantifizierungen formulieren, nicht aber Allquantifizierungen, die einen Existenzquantor enthalten. Es ist nicht möglich, solche Allquantifizierungen in SQL direkt auszudrücken. Stattdessen muss der Allquantor in solchen Anfragen umschrieben werden, z. B. durch den negierten Existenzquantor.

Als Konsequenz daraus stehen Datenbanksysteme bis her nicht vor der Aufgabe, Anfragen mit Allquantoren zu verarbeiten. Die Anfragen können lediglich Operatoren enthalten, die eventuell eine Allquantifizierung umschreiben. Für Optimierer ist es aufwendig, Allquantifizierung in Anfragen zu erkennen. Der Fra ge, wie Allquanti - fizierung in Datenbanksystemen effizient realisiert werden kann, ist entsprechend wenig Beachtung geschenkt worden. Sie wird erst allmählich als eine von Datenbanksystemen zu lösende Aufgabe wahrgenommen. Neue Datenbanksysteme bieten dem Anwender die Möglichkeit, komplexe, vom System zu überwachende Integritätsbedingungen über Relationen zu formulieren. Dies impliziert Allquantifizierung in Anfragen ebenso wie es bei Decision Support Systems, Data Mining und wissensbasierten Datenbank - systemen der Fall ist. Die Effizienz dieser Systeme hängt zunehmend davon ab, wie

effizient Allquantifizierung in Anfragen ausgeführt wird. Daher ist zu erwarten, dass sie weiter an Bedeutung gewinnt.

2.1. Formale Darstellung

Eine Allquantor-Anfrage ist eine Anfrage an ein Datenbanksystem, die einen Allquantor enthält. Allquantor -Anfragen können hinsichtlich formaler Eigenschaften und ihrer Komplexität sehr unterschiedlich sein. Es gibt z. B. einfachere Anfragen, die nur einen Allquantor enthalten, und komplexere An fragen, die einen Allquantor in Verbindung mit einem Existenzquantor enthalten. Bevor man darüber nachdenkt, wie ein Daten - banksystem Allquantor -Anfragen am besten bearbeiten soll, ist es unumgänglich, sie formal zu spezifizieren und in verschiedene Anfragetypen einzuteilen.

2.1.1. Die allgemeine Allquantor-Anfrage

$R(A_{out}, X)$ und $D(Y)$ seien Relationen, die unmittelbar oder durch Anwendung von Operatoren der Relationenalgebra aus den Relationen eines beliebigen relationalen Schemas hervorgegangen sind. A_{out} bezeichne diejenigen Attribute von R, aus denen die Ergebnisrelation der Allquantifizierung von R über D aufgebaut wird. Die übrigen Attribute X und Y der Relationen können in den Prädikaten der Anfrage verwendet werden.

Die **allgemeine Allquantor-Anfrage** ist der Anfragetyp, der die übrigen Anfragetypen als Spezialfälle enthält. Jede Allquantor -Anfrage kann durch diesen Anfragetyp zum Ausdruck gebracht werden. Die allgemeine Allquantor-Anfrage wählt

aus der Projektion	$unique(\pi_{A_{out}}(R))$	
diejenigen Tupel	q	aus,
für die die Partition	$R(q) = \{r \in R \mid r.A_{out} = q.A_{out}\} \subseteq R$	
bezüglich **aller** in einer Teilmenge	$D(q) = \{d \in D \mid P_B(d, q)\} \subseteq D$	
enthaltenen Tupel d die Bedingung	$\exists r \in R(q): P_Q(r, d)$	erfüllt.

Die Attribute A_{out} sind nicht nur die Ausgabeattribute der Allquantifizierun g. Sie haben wesentlichen Einfluss auf das Zustandekommen des Ergebnisses, denn sie definieren die **Kandidaten** q, die durch die Allquantifizierung überprüft werden. Die Relation R wird anhand A_{out} disjunkt und vollständig partitioniert, wobei jede Partition $R(q)$ die Tupel eines Kandidaten q enthält. Die Attributmenge A_{out} darf daher nicht leer sein.

$P_B(d, q)$ heißt **Bereichsprädikat**, da es aus D die Teilmenge bzw. den Bereich $D(q)$ auswählt, über den die Allquantifizierung der Partition $R(q)$ durchgeführt wird. Es kann Attribute aus den Mengen A_{out} und Y enthalten. Zu einem Kandidaten q nimmt P_B also die Auswahl bezüglich der Relation D vor und A_{out} bezüglich der Relation R. Die von

P_B erzeugten Teilmengen $D(q)$ sind im Allgemeinen allerdings weder disjunkt, noch müssen sie vereinigt die gesamte Menge D ergeben.

Das Bereichsprädikat ermöglicht es, flexible Anfragen der Art zu stellen: „Gib die Kunden aus, die alle von ihnen bestellten Waren bezahlt haben." Es wählt zu jedem Kunden q die von ihm bestellten Waren $D(q)$ aus der Menge der verfügbaren Waren D aus. Für jeden in $D(q)$ enthaltenen Artikel wird dann geprüft, ob er bezahlt ist.

$P_Q(r, d)$ heißt **Quantorprädikat**, da es die Bedingung enthält, auf die sich Allquantor und Existenzquantor beziehen. Eine Allquantifizierung muss keinen Existenzquantor enthalten, einen Allquantor dagegen schon. Also ist P_Q in jedem Fall **für alle** Tupel $d \in D(q)$ zu prüfen. Es muss mindestens ein Attribut der Relation D enthalten, da es ansonsten unabhängig vom Allquantor ausgewertet werden könnte und keine Allquantifizierung mehr vorläge. In P_Q können Attribute aus beiden Relationen R und D verwendet werden.

In Übereinstimmung mit dem Sprachgebrauch der relationalen Division nennen wir R auch Dividendenmenge, $R(q)$ Dividend, D Divisormenge, $D(q)$ Divisor und die Ergebnisrelation Q Quotient (vgl. [8]).

Darstellung in der Relationenalgebra und in SQL

Die allgemeine Allquantor-Anfrage mit Ergebnisrelation Q können wir in der Relationenalgebra folgendermaßen darstellen:

$$Q = \{q \in \pi_{A_{out}}(R) \mid \forall d \in D(q)\ \exists r \in R(q): P_Q(r, d)\}$$

mit $D(q) = \{d \in D \mid P_B(q, d)\}$ ergibt sich

$$Q = \{q \in \pi_{A_{out}}(R) \mid \forall d \in D\ \exists r \in R(q): P_B(q, d) \Rightarrow P_Q(r, d)\}$$

Um Allquantifizierungen wie diese in Anfragen direkt formulieren zu können, muss SQL um ein Schlüsselwort für den Allquantor erweitert werden. Das Schlüsselwort heiße FOR ALL und habe die Syntax:

```
FOR ALL (Relation) (Bedingung)
```

Relation ist eine Basisrelation oder eine beliebige Unteranfrage, die eine Relation erzeugt. Das Prädikat FOR ALL ist wahr, wenn *Bedingung* für alle Tupel von *Relation* wahr ist; ansonsten ist es falsch. Das Anfrageschema der allgemeinen Allquantor-Anfrage lautet:

```
select  distinct Q.A_out
from    R as Q
where   for all  (select  *
                  from    D
                  where   P_B(q,d) )
                 (exists  (select  *
                           from    R
                           where   R.A_out = Q.A_out  and  P_Q(r,d) ) )
```

Die SELECT-Klausel beschreibt die Kandidaten für die Ergebnisrelation Q der Anfrage. Jede in der Dividendenmenge R auftretende Belegung der Attribute A_{out} definiert einen eindeutigen Kandidaten q. Die WHERE-Klausel prüft für jeden Kandidaten, ob sein

Dividend *R(q)* für alle Tupel des zugehörigen Divisors *D(q)* das Quantorprädikat P_Q erfüllt. In der ersten Unteranfrage wird der Divisor erzeugt. Das Bereichsprädikat P_B wählt aus der Divisorrelation *D* die entsprechenden Tupel aus. Der Allquantor FOR ALL veranlasst die Überprüfung des Existenzquantors EXISTS für jedes dieser Tupel. Die Unteranfrage des Existenzquantors erzeugt den Dividend des Kandidaten durch die Bedingung $(R.A_{out} = Q.A_{out})$. Der zweite Teil der Bedingung enthält das Quantorprädikat P_Q. Wenn mindestens ein Tupel diese beiden Bedingungen erfüllt, ist die Unteranfrage nicht leer und der Existenzquantor liefert den Wert *wahr* zurück. Kandidaten, die die Allquantifizierung der WHERE-Klausel erfüllen, werden in die Ergebnisrelation übernommen.

Umschreibungsmöglichkeiten

In relationalen Datenbanksystemen kann Allquantifizierung in Anfragen auch mit Hilfe der folgenden algebraischen Operatoren ausgedrückt werden:

- relationale Division ÷
- negierter Existenzquantor $\neg\exists$
- Gruppierung mit Aggregation (Count)
- Mengendifferenz –

Unabhängig von der Formulierung der Anfrage in SQL kann sie vom Optimierer in einen Ausführungsplan überführt werden, der auf einem dieser vier Operatoren basiert. Voraussetzung ist, dass der Optimierer die Allquantifizierung auch dann als solche erkennt, wenn sie umschrieben wird und das Schlüsselwort FOR ALL bzw. ALL nicht auftaucht. Die relationale Division entspricht einer direkten Umsetzung des Allquantors in Verbindung mit dem Existenzquantor. Die anderen drei Operatoren führen zu algebraisch äquivalenten Ausdrücken. Sie müssen in SQL 92 herangezogen werden, um Allquantifizierung in Anfragen überhaupt ausdrücken zu können.

Die folgenden Darstellungen entsprechen Ausführungsplänen der allgemeinen Allquantor-Anfrage. Die Pläne haben nicht das Ziel, die optimale Ausführung im Falle des jeweiligen Operators zu beschreiben. Sie sollen einen Eindruck von der Vielfalt an Darstellungsmöglichkeiten vermitteln, die in relationalen Datenbanksystemen gegeben ist.

Bereichs- und Quantorprädikat können Attribute aus den beiden Relationen *R* und *D* enthalten. Für Anfragetypen, bei denen eines bzw. beide Prädikate Attribute aus nur einer oder keiner dieser Relationen enthalten, können die Ausführungspläne vereinfacht werden.

(1) Ausführung mit **relationaler Division**

$$Q = (\pi_{A_{out}}(R) \bowtie_{\neg P_B \vee P_Q} D) \div D$$

(2) Ausführung mit **negiertem Existenzquantor**

$$Q = \{q \in \pi_{A_{out}}(R) \mid \neg\exists d \in D\ \neg\exists r \in R(q): P_B(q, d) \Rightarrow P_Q(r, d)\}$$

(3) Ausführung mit **Mengendifferenz**

$$Q = \pi_{A_{out}}(R) - \pi_{A_{out}}((\pi_{A_{out}}(R) \bowtie_{P_B} D) - (\pi_{A_{out}}(R) \bowtie_{P_B \wedge P_Q} D))$$

(4) Ausführung mit **Aggregation und Zählen**

Auf die Angabe eines formalen Ausführungsplans wird an dieser Stelle verzichtet. Einfacher als einen solchen Plan nachzuvollziehen ist es, der in [8] gegebenen Argumentation zu folgen, wonach jede Allquantor -Anfrage durch Ag gregation und Zählen ausgedrückt werden kann.

2.1.2. Klassifizierung von Allquantor-Anfragen

Allquantifizierung ist als eine boolesche Funktion definiert, die ein Quantorprädikat P_Q für alle von einem Bereichsprädikat P_B aus einer Menge D ausgewählten Elemente überprüft. Allquantor-Anfragen können anhand der Prädikate P_B und P_Q in verschiedene Typen eingeteilt werden. Für P_B werden vier Fälle unterschieden:

(1) Es enthält Attribute aus beiden Relationen R und D.

(2) Es enthält Attribute nur aus R.

(3) Es enthält Attribute nur aus D.

(4) Das Prädikat fehlt bzw. ist wahr.

Im Unterschied zu der Klassifizierung in [9] werden für P_Q nur die Fälle (1) und (3) betrachtet. Bei einer Allquantifizierung ist das Quantorprädikat für alle Tupel aus D zu prüfen und muss daher auch von D abhängen. Ansonsten hat der Allquantor kein Prädikat und es liegt keine Allquantifizierung vor. Für P_Q wird im Fall (1) weiter unterschieden, ob das Prädikat in Verbindung mit einem Existenzquantor verwendet wird. Diese drei Fallunterscheidungen von P_Q führen in Kombination mit den vier Fall -unterscheidungen von P_Q zu einer Klassifizierung in zwölf Anfragetypen:

	$P_Q(d)$	$P_Q(r,d)$	$\exists P_Q(r,d)$
$P_B()$	1	2	**3**
$P_B(r)$	4	5	**6**
$P_B(d)$	7	8	**9**
$P_B(r,d)$	10	11	**12**

Tabelle 2.1: Klassifizierung der Allquantor-Anfragen

Die Typen 1 bis 3 besitzen kein Bereichsprädikat und schließen alle Anfragen ein, bei denen die Allquantifizierung über die gesamte Divisorrelation D ausgeführt wird. Ähnlich verhält es sich mit den Typen 4 bis 6. Sie besitzen ein Bereichsprädikat, das ausschließlich von R abhängt. Als Konsequenz daraus wird die Allquantifizierung in Abhängigkeit vom Kandidaten q entweder über ganz D oder über der leeren Menge ausgeführt. Der Allquantor auf die leere Menge angewandt führt per Definition unabhängig von P_Q zu einer wahren Aussage. In Anbetracht ihrer Eigenschaften sind diese Anfragetypen ungewöhnlich und von untergeordneter Bedeutung.

Bei den Anfragetypen 7 bis 9 wählt das Bereichsprädikat einen konstanten Divisor $D(q)$ aus der Relation D aus. Bei den Typen 10 bis 12 enthält das Bereichsprädikat auch Attribute A_R des Dividenden. Das bedeutet, dass der Divisor in Abhängigkeit des

Kandidaten q ausgewählt wird und nicht konstant ist. Er wird durch eine korrelierte Unteranfrage erzeugt. Zu jeder in R vorkommenden Belegung v von A_R stellt P_B einen anderen Divisor D_v aus D zusammen. Diese Anfragen stellen durch ihren variablen Divisor manche Algorithmen vor Probleme.

Bei den Anfragetypen der linken Spalte bezieht sich das Quantorprädikat nur auf die Menge D. Sie vergleichen für einen Kandidaten q alle Tupel aus $D(q)$ mit konstanten Werten. Die Anfragetypen der mittleren Spalte sind ihnen ähnlich. Ihr Quantorprädikat bezieht sich auch auf die Mengen R, so dass sie alle Tupel aus $D(q)$ mit Werten des Kandidaten q vergleichen. Interessant sind die Anfragen der rechten Spalte, deren Quantorprädikat P_Q sich auf beide Mengen R und D bezieht und im Zusammenhang mit einem Existenzquantor verwendet wird. Sie basieren auf der relationalen Division, da sie das Enthalten-Sein von $D(q)$ in $R(q)$ prüfen und zu jedem Tupel aus $D(q)$ nach einem korrespondierenden Tupel in $R(q)$ suchen, das P_Q erfüllt.

Die Anfragetypen werden in zwei übergeordnete Gruppen eingeteilt. Die fett gedruckten Typen der rechten Tabellenspalte sind relationale Divisionen. Eine ***relationale Division*** ist eine Allquantifizierung, deren prädikatenlogischer Ausdruck einen Allquantor gefolgt von einem Existenzquantor und einem Prädikat enthält (Schema: $\forall x\, \exists y$: *Prädikat(x,y)*). Sie können durch eine oder mehrere Anwendungen des in Abschnitt 1.1 definierten Operators relationale Division ausgewertet werden. Die Anfragetypen der ersten und zweiten Spalte bilden die einfachen Allquantor-Anfragen. Ein ***einfacher Anfragetyp*** enthält in seinem prädikatenlogischen Ausdruck nur einen Allquantor und ein Prädikat. Er kann in der Relationenalgebra nicht in Form einer oder mehrerer relationaler Divisionen dargestellt werden.

Wie der Name bereits zum Ausdruck bringt, sind die einfachen Anfragetypen im Vergleich zu den relationalen Divisionen weniger komplex und können von einem Datenbanksystem unter geringem Aufwand ausgeführt werden. Die Typen 1 und 7 enthalten in keinem der Prädikate P_B, P_Q Attribute aus R. Sie stellen die simpelsten Anfragen dar, da die Unteranfrage des Allquantors nicht korreliert ist und unabhängig ausgewertet werden kann. Relationale Divisionen sind komplexe und aufwendige Anfragen für ein Datenbanksystem. Bei der Optimierung und effizienten Ausführung von Allquantifizierung stehen sie im Mittelpunkt des Interesses. Typ 12 entspricht der allgemeinen Allquantor-Anfrage aus dem vorigen Abschnitt und enthält zwei korrelierte Unteranfragen. Er stellt die interessanteste und mächtigste Allquantifizierung dar.

2.1.3. Einfache Anfragetypen

Bei den einfachen Anfragetypen mag es genau genommen nicht korrekt sein, von einem Dividenden und einem Divisor zu sprechen, da die Allquantifizierung nicht auf eine relationale Division zurückgeführt werden kann. Um aber den Vorzug einer gleich-bleibenden Begrifflichkeit nicht zu verlieren, werden die Begriffe Dividend und Divisor in angepasster Form auch hier verwendet. Der Divisor ist stets die Relation, auf die sich der Allquantor bezieht und über der die Allquantifizierung durchgeführt wird. Der Dividend ist die Relation, aus der die Kandidaten für das Ergebnis entnommen werden.

Die folgenden Beispiele sollen die Klassifizierung und die Unterschiede zwischen den einzelnen Anfragetypen verdeutlichen. Die Beispielanfragen folgen dem SQL-Schema, das für die allgemeine Allquantor-Anfrage angegeben wurde.

Beispiel 2.1

Die Datenbank eines Unternehmens enthalte die Relation *Abteilungen* mit Abteilungsnummer, Abteilungsleiter und Gehalt der Abteilungsleiter sowie die Relation *Mitarbeiter* mit Personalnummer, Abteilungsnummer und Gehalt der Mitarbeiter.

Abteilungen (Anr, ALeiter, Gehalt)

Mitarbeiter (Pnr, Anr, Gehalt)

Der Chef möchte wissen, ob es „teure" Abteilungen gibt, in denen alle Mitarbeiter mehr als 80.000 DM Jahresgehalt verdienen. Er stellt die Anfrage:

```
select  Anr
from    Abteilungen as Q
where   for all (select *
                 from   Mitarbeiter as D
                 where  D.Anr = Q.Anr)
                (D.Gehalt > 80.000)
```

Die Relation *Mitarbeiter* ist die Divisorrelation, da sich der Allquantor auf sie bezieht. Die Relation *Abteilungen* alias *Q* entspricht der Dividendenrelation *R*; aus ihr stammen die Kandidaten *q* der Allquantifizierung. Das Bereichsprädikat hängt von beiden Relationen *R* und *D* ab, das Quantorprädikat dagegen nur von *D*. Die Anfrage ist also vom Typ 10.

Beispiel 2.2

Der Chef hat gehört, dass alle Mitarbeiter der Abteilung 12 ein höheres Gehalt bekämen als ihr Abteilungsleiter. Um herauszufinden, ob das stimmt, und gegebenenfalls das Gehalt des Abteilungsleiters in Erfahrung zu bringen, stellt er die Anfrage:

```
select  Gehalt
from    Abteilungen as Q
where   Anr = 12  and
        for all (select *
                 from   Mitarbeiter as D
                 where  D.Anr = 12)
                (D.Gehalt > Q.Gehalt)
```

Die Anfrage ist vom Typ 8, da das Bereichsprädikat nur von der Divisorrelation, das Quantorprädikat dagegen von beiden Relationen *R* und *D* abhängt.

Beispiel 2.3

Das Gerücht hat sich leider bestätigt, weshalb der Chef wissen möchte, ob es noch weitere Abteilungen gibt, deren Abteilungsleiter ein niedrigeres Gehalt bekommen als alle Mitarbeiter in ihrer Abteilung. Er formuliert die Anfrage:

```
select  Anr
from    Abteilungen as Q
where   for all (select *
                 from   Mitarbeiter as D
                 where  D.Anr = Q.Anr)
                (D.Gehalt > Q.Gehalt)
```

Bei dieser Anfrage hängen beide Prädikate P_B, P_Q sowohl von R als auch von D ab. Sie ist vom Typ 11 und repräsentiert den allgemeinsten Typ der einfachen Anfragen. Die einfachen Anfragety pen enthalten keinen Existenzquantor. Daher kann für den Allquantor das Schlüsselwort ALL anstatt FOR ALL verwendet werden, wenn P_Q einen Vergleich der Divisortupel mit einem zusammenhängenden Wertebereich durchführt. Im Beispiel ist dies gegeben, da der W ertebereich der Gehälter, die über dem Wert *Q.Gehalt* liegen, zusammenhängend ist. Da zudem nur die untere Bereichsgrenze geprüft werden muss, ist die Formulierung mit ALL kompakter:

```
select  Anr
from    Abteilungen as Q
where   Gehalt < all (select Gehalt
                      from   Mitarbeiter as D
                      where  D.Anr = Q.Anr)
```

Wenn beide Bereichsgrenzen zu prüfen sind, ist die Verwendung von FOR ALL vorzuziehen, da ALL zwei Unterabfragen benötigt. Man betrachte dieselbe Anfrage mit $P_Q = (Gehalt \geq 50.000 \text{ and } Gehalt \leq 60.000)$. Sie ermittel t die Abteilungen, deren Mitarbeiter alle zwischen 50.000 und 60.000 DM verdienen. Die WHERE-Klausel hat bei Verwendung von ALL die Form:

```
where  50.000 ≤ all (select Gehalt
                     from   Mitarbeiter as D
                     where  D.Anr = Q.Anr)  and
       60.000 ≥ all (select Gehalt
                     from   Mitarbeiter as D
                     where  D.Anr = Q.Anr)
```

Bei einem nicht zusammenhängenden Wertebereich ist keine Formulierung der Anfrage mit ALL möglich. Das ist beispielsweise der Fall, wenn alle Gehälter in einer Abteilung 50.000 oder 60.000 DM betragen sollen. Das Quantorprädikat muss dann tupelweise geprüft werden und kann nicht als Vergleich von Werten mit der Menge *D(q).Gehalt* ausgedrückt werden.

Beispiel 2.4

Die Anfragen zu den bisherigen Beispielen können be i zusammenhängendem Werte - bereich auch als Gruppierung der Relation *Mitarbeiter* nach dem Attribut *Anr* und Anwendung der Aggregatfunktionen MIN bzw. MAX formuliert werden. Dies ist allerdings nur deshalb möglich, weil das Bereichsprädikat den Divisor nach *Anr* partitioniert. In Abschnitt 2.1.1 wurde bereits erwähnt, dass die Divisoren *D(q)* im

Allgemeinen nicht disjunkt sind. Die folgende Anfrage enthält Divisoren, die nicht disjunkt sind, und schließt die Beispiele zu den einfachen Anfragetypen ab.

Die Relation der Mitarbeiter wird um ein Attribut ergänzt, das das Alter der Mitarbeiter enthält: *Mitarbeiter(PNr, Anr, Gehalt, Alter)* . Um auf weitere Unannehmlichkeiten vorbereitet zu sein, stellt der Chef die Anfrage nach Personalnummer und Gehalt derjenigen Mitarbeiter, die weniger verdienen als alle jüngeren Mitarbeiter.

```
select  Pnr, Gehalt
from    Mitarbeiter as Q
where   for all (select *
                 from   Mitarbeiter as D
                 where  D.Alter < Q.Alter)
                (D.Gehalt > Q.Gehalt)
```

Eine Besonderheit dieser Anfrage ist, dass es sich bei der Dividenden- und der Divisorrelation um dieselbe physische Relation *Mitarbeiter* handelt. Aus ihr entstammen sowohl die Kandidaten *q* als auch die Tupel des zugehörigen Divisors *D(q)*. An dieser Stelle kann man sich nochmals verdeutlichen, dass die Bezeichnung einer Relation als Dividenden- oder Divisorrelation eine semantische Rollenbezeichnung ist. Sie sagt nichts darüber aus, welche Eigenschaften die Relation besitzt oder wie sie erzeugt wurde, sondern sie sagt aus, in welcher Rolle die Relation bei der Allquantifizierung verwendet wird, nämlich als Dividenden- oder als Divisorrelation. Die Anfrage ist vom Typ 11, da beide Prädikate P_B und P_Q sowohl Attribute aus der Dividenden- als auch aus der Divisorrelation enthalten.

2.1.4. Relationale Divisionen

Relationale Divisionen sind die Allquantifizierungen, die einen Allquantor in Kombination mit einem Existenzquantor enthalten. Erst in der Kombination beider Quantoren kommt ihre prädikatenlogische Ausdruckskraft zur vollen Entfaltung. Dies spiegelt sich in sprachlichen und mathematischen Aussagen wieder, sobald es um komplexere Zusammenhänge geht. Aufgrund ihrer Komplexität ist die Ausführung der zugehörigen Allquantor-Anfragen mit hohen Kosten verbunden. In den nächsten beiden Kapiteln werden zwei Algorithmen vorgestellt, die die effiziente Ausführung solcher Anfragen zum Ziel haben. Die für die Anwendung bedeutenden Anfragetypen 3, 9 und 12 werden in den folgenden Beispielen dargestellt.

Beispiel 2.5

Die Datenbank einer kleinen Hochschule, an der nur Informatik studiert werden kann, enthält zwei Relationen mit den Schemata *Abgelegt(Pnr, Sname, Datum, Note)* und *Prüfungen(Pnr, Pname)* . Die Namen der Studenten sind eindeutig, so dass sie einer Matrikelnummer entsprechen. Jede von einem Studenten abgelegte Prüfung wird mit Prüfungsnummer, Name des Studenten, Datum und Note in die Tabelle *Abgelegt* eingetragen. Die Tabelle *Prüfungen* enthält alle Prüfungen des Studienganges Informatik. Die Anfrage, welche Studenten bereits alle Prüfungen abgelegt und bestanden haben, ist eine Allquantifizierung von *Abgelegt* über *Prüfungen*. In

Übereinstimmung mit dem Anfrageschema aus Abschnitt 2.1.1 und der Klassifizierung wird die Dividendenrelation mit *R* bzw. *Q* und die Divisorrelation mit *D* bezeichnet. Die Anfrage gibt die gesuchten Studenten aus und lautet:

```
select  distinct Sname
from    Abgelegt as Q
where  for all  (select  *
                 from    Prüfungen as D)
                (exists  (select  *
                          from    Abgelegt as R
                          where  R.Sname = Q.Sname  and
                                 R.Pnr = D.Pnr  and  R.Note ≤ 4,0 ) )
```

Die Anfrage enthält kein Bereichsprädikat, da die Allquantifizierung über die gesamte Divisorrelation *Prüfungen* durchzuführen ist. Die letzte Zeile der WHERE-Klausel enthält das Quantorprädikat. Es bezieht sich durch das gemeinsame Attribut *Pnr* auf beide Relationen *R* und *D*. Die Anfrage ist vom Typ 3.

Beispiel 2.6

Für die Datenbank einer Universität wird das Schema des vorigen Beispiels erweitert, da es an einer Universität eine Vielzahl von Studiengängen gibt.

Studenten (Sname, Studium)

Prüfungen (Pnr, Pname, Studium)

Abgelegt (Pnr, Sname, Datum, Note)

Die Relation *Studenten* enthält zum Namen eines Studenten (Primärschlüssel) den Studiengang, in dem er eingeschrieben ist. Die Relation *Prüfungen* beinhaltet die Prüfungen zu jedem Studiengang. In die Relation *Abgelegt* wird jede Prüfung eingetragen, die ein Student abgelegt hat. Sie steht über die gemeinsamen Attribute *Sname* und *Pnr* in einer n:1-Beziehung zu den Relationen *Studenten* und *Prüfungen*.

Die Anfrage, welche Studenten des Studienganges Informatik bereits alle Prüfungen abgelegt und bestanden haben, lautet nun:

```
select  Sname
from    Studenten as Q
where  Studium = 'Informatik'  and
       for all  (select  *
                 from    Prüfungen as D
                 where  D.Studium = 'Informatik')
                (exists  (select  *
                          from    Abgelegt as R
                          where  R.Sname = Q.Sname and
                                 R.Pnr = D.Pnr  and  R.Note ≤ 4,0 ) )
```

Die Dividendenrelation *R* setzt sich aus den Relationen *Studenten* und *Abgelegt* zusammen[1]. Die Anfrage betrachte t nur die Informatikstudenten als Kandidaten *q*. Sie werden durch das Attribut *Studium* ausgewählt. Der Divisor wird anhand des Bereichsprädikats gebildet, das über das Attribut *Studium* die Informatikprüfungen aus der Divisorrelation *Prüfungen* auswählt. An der Unteranfrage des Existenzquantors, die den Dividend erzeugt und das Quantorprädikat anwendet, ändert sich nichts. Sie wird von den Kandidaten der äußeren Anfrage durch *R.Sname = Q. Sname* gesteuert. Die Anfrage ist vom Typ 9.

Beispiel 2.7

An dieselbe Datenbank wird die Anfrage gestellt, welche Studenten in ihrem Studien - gang bereits alle Prüfungen abgelegt und bestanden haben. Diese in der sprachlichen Formulierung geringfügige Änderung führt auf den Anfragetyp 12. Er stellt manch e Algorithmen vor Probleme, weil der Divisor nicht mehr konstant ist, sondern vom Studiengang des jeweiligen Kandidaten der Dividendenrelation abhängt. Beide Prädikate P_B und P_Q enthalten Attribute aus den Relationen *R* und *D*. Die Anfrage lautet:

```
select  Sname
from    Studenten as Q
where   for all  (select  *
                  from    Prüfungen as D
                  where   D.Studium = Q.Studium)
                 (exists  (select  *
                           from    Abgelegt as R
                           where   R.Sname = Q.Sname and
                                   R.Pnr = D.Pnr  and  R.Note ≤ 4,0 ) )
```

[1] Wen dies irritiert, der möge sich die Dividendenrelation *R* explizit durch den natürlichen Verbund *Studenten*⋈*Abgelegt* erzeugen und darauf die Anfrage anwenden.

3. Der Ansatz Hash-Division

In relationalen Datenbanksystemen, die über keinen algebraischen Operator für relationale Division verfügen, muss eine solche Allquantifizierung durch andere Operatoren ausgeführt werden (vgl. Kapitel 2). Die so entstehenden Ausführungspläne sind nicht optimal (siehe [8]). Der Algorithmus Hash-Division implementiert einen Operator ÷ für relationale Division. Er wird in [8] beschrieben und gängigen Algorithmen für relationale Division gegenübergestellt. Aus den Vergleichen, die Duplikate in Dividend und Divisor, referentielle Integrität zwischen beiden Relationen, Two-Pass Varianten und Parallelisierbarkeit der Algorithmen berücksichtigen, geht Hash-Division als effizientester Algorithmus hervor. Er erreicht die Performance von hash-basierten Semi-Join Algorithmen, die Anfragen mit Existenzquantifizierung bearbeiten. Es wird damit geschlossen, dass ein relationales Datenbanksystem All- und Existenzquantifizierung mit gleicher Effizienz bearbeiten kann, wenn es über Hash-Division verfügt.

Die letzte Aussage ist falsch. In [8] wird nicht ausreichend zwischen Allquantifizierung und relationaler Division differenziert und die relationale Division ist ohne Divisionsprädikat und Quotientenattribute definiert. Es entsteht der falsche Eindruck, dass jede Allquantifizierung mit Existenzquantor auf eine relationale Division ohne Prädikat abgebildet und von Hash-Division bearbeitet werden kann. An keiner Stelle wird darauf eingegangen, ob und wie Bereichs- und Quantorprädikat einer Allquantifizierung auf den Operator relationale Division übertragen werden können und wie Hash-Division mit diesen Prädikaten verfahren müsste. Richtig wäre an jener Stelle die Aussage, dass relationale Division und Existenzquantifizierung mit gleicher Effizienz bearbeiten werden können, wenn das Datenbanksystem Hash-Division einsetzt.

Bei einer Allquantifizierung von $R(A_{out}, X)$ über $D(Y)$ ist für jeden Kandidaten $q \in \pi_{A_{out}}(R)$ der Ausdruck $\forall d \in D \; \exists r \in R(q): P_Q$ zu prüfen. Für die Anwendung des Operators relationale Division zur Bearbeitung einer solchen Allquantifizierung gilt: Das Quantorprädikat P_Q entspricht dem Divisionsprädikat P_D und die Ausgabeattribute A_{out} den Quotientenattributen A_Q. Das Divisionsprädikat wird in zwei Teilprädikate P_1 und P_2 zerlegt. P_1 enthält alle Gleichheitsbedingungen der Form $(r.x_i = d.y_j)$ mit den Attributen $x_i \in X_1 \subseteq X$ und $y_j \in Y_1 \subseteq Y$. Es ordnet einem Tupel r des Dividenden das korrespondierende Tupel d im Divisor über die Hash-Funktion h_D zu und darf für die Anwendung von Hash-Division nicht leer sein. Die Attribute A_{out} können in P_1 nicht verwendet werden, da sie die Kandidaten q definieren und innerhalb eines Dividenden

R(q) stets gleich belegt sind. Das Teilprädikat P_2 enthält alle übrigen Bedingungen des Prädikats P_D. Es wird auf die von P_1 erzeugten Paare korrespondierender Tupel *(r, d)* angewandt. Prädikat $P_D(r, d)$ ist also erfüllt, wenn zu einem Tupel *r* über die Hash-Funktion h_D ein korrespondierendes Tupel *d* gefunden wird und $P_2(r, d)$ erfüllt ist.

Hash-Division setzt einen konstanten Divisor voraus und kann daher nur Allquantor-Anfragen der Typen 3, 6 und 9 unmittelbar verarbeiten. Bei Anfragen vom Typ 3 fehlt das Bereichsprädikat P_B, so dass die Allquantifizierung über ganz *D* ausgeführt wird. Bei Anfragen vom Typ 9 führt das Datenbanksystem die im Bereichsprädikat $P_B(d)$ enthaltene Restriktion auf der Relation *D* aus und erzeugt so den Divisor der relationalen Division. Anfragen vom Typ 6 können durch eine einfachen Filter ebenfalls bearbeitet werden. Die Kandidaten *q*, für die $P_B(r)$ falsch ist, haben einen leeren Divisor und können direkt in den Quotienten übernommen werden. Für die Kandidaten, auf die dies nicht zutrifft, wird durch relationale Division mit Divisor *D* geprüft, ob sie in den Quotienten übernommen werden. Bei Anfragen vom Typ 12 werden durch das Bereichsprädikat $P_B(r,d)$ mehrere verschiedene Divisoren erzeugt. Hash-Division kann bei solchen Anfragen nicht unmittelbar eingesetzt werden, da der Algorithmus nicht in der Lage ist, mehrere Divisoren zu verwalten. In Abschnitt 3.2 ist dargestellt, wie diese Anfragen aufbereitet werden müssen, damit Hash-Division sie bearbeiten kann.

3.1. Algorithmus Hash-Division

Die hier angegebene Spezifikation des Algorithmus Hash-Division korrespondiert mit der Definition der relationalen Division $\div_{P_D, A_Q}$ aus Abschnitt 1.1. Die Spezifikation aus [8] wurde an zwei Stellen geringfügig ergänzt, damit Hash-Division eine relationale Division mit Prädikat realisiert. Der Algorithmus arbeitet mit zwei Hash-Tabellen. Die zugehörigen Hash-Funktionen werden mit h_D und h_Q bezeichnet. Die Funktion h_Q berechnet zu Tupeln aus der Dividendenrelation *R* unter Verwendung der Quotientenattribute A_Q das Bucket des zugehörigen Kandidaten *q*. Die Funktion h_D wurde bereits erläutert; sie ermittelt zu einem Tupel aus *R* das korrespondierende Tupel im Divisor *D*.

(1) Baue die Hash-Tabelle des Divisors *D* auf:
   setze Variable *divisor_count* auf 0
   für jedes Tupel $d \in D$
   - füge *d* im Bucket $h_D(d)$ ein
   - weise *d* die Divisor-Nummer $nr(d) = divisor\_count$ zu
   - erhöhe *divisor_count*

(2) Baue die Hash-Tabelle Q_C der Kandidaten des Quotienten auf:
für jedes Tupel $r \in R$
suche anhand $h_D(r)$ zu r gehöriges Tupel d in der Divisor-Tabelle
wenn Tupel d gefunden und $P_2(r,d)$ erfüllt, dann *// Ergänzung um P_2*
- suche anhand $h_Q(r)$ zu r gehöriges Tupel q in der Kandidaten-Tabelle
  wenn Kandidat q nicht gefunden, dann
  - erstelle zu Tupel r einen neuen Kandidaten q mit der Belegung $r.A_Q$ und einer mit 0 initialisierten Bitmap $B(q)$ *// Ergänzung um A_Q*
  - füge q in die Kandidaten-Tabelle Q_C im Bucket $h_Q(r)$ ein
- setze das zur Divisor-Nummer $nr(d)$ gehörige Bit der Bitmap $B(q)$ auf 1

(3) Ermittle den Quotienten Q mit Hilfe der Kandidaten-Tabelle Q_C:
prüfe die Bitmap $B(q)$ von jedem Kandidaten q
wenn alle Bits auf 1 gesetzt sind, dann übertrage q in den Quotienten Q

Beim Aufbau der Hash-Tabelle des Divisors wird jeder Eintrag d mit einer fortlaufenden Divisor-Nummer versehen, die ihm genau ein Bit in den Bitmaps der Kandidaten-Tabelle zuordnet. Nach dieser Vorarbeit muss der Dividend einmal sequentiell gelesen werden. Nur diejenigen Tupel $r \in R$, zu denen ein Tupel d in der Divisor-Tabelle gefunden wird und die $P_2(r,d)$ erfüllen, werden berücksichtigt. Auf Basis der Quantorattribute A_Q werden die in diesen Tupeln enthaltene Kandidaten q gebildet und in der Kandidaten-Tabelle Q_C eingetragen. Das Bit ihrer Bitmap, welches mit dem Divisortupel d korrespondiert, wird auf 1 gesetzt. Nach dem Lesen des Dividenden wird die Tabelle Q_C gescannt. Als Ergebnis der Division werden diejenigen Tupel q ausgegeben, deren Bitmap nur Einsen enthält.

Duplikate im Dividenden $R(q)$ führen dazu, dass Bits des Kandidaten q, die bereits den Wert 1 enthalten, nochmals auf 1 gesetzt werden; sie werden also ignoriert. Duplikate im Divisor werden beim Erstellen der Hash-Tabelle automatisch eliminiert, da für jede Belegung bezüglich der Attribute Y_1 genau ein Eintrag in der Hash-Tabelle erfolgt. Diese implizite Duplikateliminierung bedeutet, dass Duplikate im Divisor dem Algorithmus keine Schwierigkeiten bereiten und nicht explizit entfernt werden müssen.

Beispiel 3.1

Die Vorgehensweise von Hash-Division soll anhand des Szenarios aus Beispiel 2.5 verdeutlicht werden. Der Anschaulichkeit wegen werden die Tabellen in der verkürzten Form *Abgelegt(Sname, Pnr, Note)* und *Prüfungen(Pnr)* nur mit den in der Anfrage verwendeten Attributen dargestellt. Die Anfrage soll diejenigen Studenten ausgeben, die alle Prüfungen des einzigen Studienganges Informatik abgelegt und bestanden haben (Anfragetyp 3). Das Quantorprädikat lautet $P_Q = (R.Pnr = D.Pnr \text{ and } R.Note \leq 4{,}0)$. Der Algorithmus Hash-Division kann angewandt werden, da $P_1 = (R.Pnr = D.Pnr)$ nicht leer ist und $P_2 = (R.Note \leq 4{,}0)$ sich nur auf R bezieht. Die Division $Abgelegt \div_{P_Q,\, Sname} Prüfungen$ liefert.

Abgelegt	Sname	Pnr	Note
	Anna	1	1,7
	Mark	2	2,3
	Mark	3	3,0
	Alex	5	1,3
	Alex	1	2,0
	Anna	3	5,0
	Alex	3	2,3
	Mark	1	5,0
	Anna	2	2,5
	Mark	4	2,0
	Alex	2	1,0
	Alex	6	3,3
	Mark	1	3,7
	Anna	4	1,7

Tabelle 3.1: Abgelegte Prüfungen

Prüfungen	Pnr
	1
	2
	3
	4

Tabelle 3.2: Informatik-Prüfungen

Die Studenten Anna und Mark haben alle Informatik-Prüfungen abgelegt. Anna hat die Prüfung 3 allerdings nicht bestanden. Mark hat die Prüfung 1 beim zweiten Versuch bestanden. Er erfüllt die Allquantifizierung und wird in den Quotienten übernommen. Student Alex hat zwei Prüfungen abgelegt, die keine Informatik-Prüfungen sind; ihm fehlt noch die Prüfung 2.

Tabelle 3.3 zeigt die aus *Prüfungen* aufgebaute Hash-Tabelle *D* mit Divisor-Nummer und Hash-Funktion *h(Pnr) = Pnr mod 3*. Sie besitzt einen Überlauf für *h(Pnr) = 1*. Tabelle 3.4 zeigt die Kandidaten-Tabelle Q_C nach Abschluss des Algorithmus. (Ihre Hash-Funktion ist der Einfachheit halber nicht dargestellt.) Jede Eins in den Bitmaps entspricht einer bestandenen Prüfung. Lediglich die Bitmap von Mark ist voll mit Einsen besetzt. Er wird als Ergebnis der Division ausgegeben.

h(Pnr)	Pnr	nr(d)
0 →	3	2
1 →	1	0
2 →	2	1
	4	3 ←

Tabelle 3.3: Divisor *D*

Sname	Bitmap			
	0	1	2	3
Anna	1	1	0	1
Mark	1	1	1	1
Alex	1	1	1	0

Tabelle 3.4: Kandidaten Q_C

Eine Variante des Algorithmus mit Namen *Incremental Hash-Division* kann eingesetzt werden, wenn der Dividend nach den Quotienten-Attributen A_Q sortiert oder gruppiert vorliegt. Der Dividend wird dann gruppenweise verarbeitet. In unserem Beispiel würde die Tabelle *Abgelegt* studentenweise gelesen. Das Anlegen einer Hash-Tabelle von Kandidaten ist nicht notwendig. Es wird lediglich der Kandidat *q* inklusive Bitmap

benötigt, dessen Gruppe gerade verarbeitet wird. Nachdem das letzte Tupel der Gruppe gelesen worden ist, überprüft Incremental Hash-Division, ob die Bitmap des Kandidaten nur Einsen enthält. Je nach dem wird *q* entweder in die Ergebnisrelation übernommen oder verworfen. Die Bitmap wird auf Null gesetzt und die nächste Gruppe verarbeitet.

3.2. Eigenschaften des Algorithmus

Graefe und Cole stellen in [8] die Ausführungsmöglichkeiten der Allquantifizierung durch den negierten Existenzquantor, durch Aggregieren und Zählen und durch die Bildung von Mengendifferenzen der Ausführung durch relationale Division gegenüber. Die relationale Division, implementiert durch Hash-Division, geht als effizienteste Ausführungsmöglichkeit aus den Vergleichen in verschiedenen Szenarien hervor. Eine weitere Stärke des Algorithmus ist seine universelle Einsetzbarkeit im Hinblick auf Duplikate und einzuhaltende referentielle Integrität. Er benötigt im Gegensatz zu anderen Ansätzen keine vorgeschaltete Duplikateliminierung und auch keine Semi-Join Operationen, um referentielle Integrität zwischen Dividend und Divisor zu erzwingen, denn er ignoriert Duplikate und erzwingt referentielle Integrität automatisch. Bezüglich seiner Eingaberelationen Dividend und Divisor setzt Hash-Division keine bestimmten physischen Eigenschaften voraus. Wenn allerdings der Dividend nach den Quotienten-Attributen A_Q gruppiert ist, kann dies durch die Variante Incremental Hash-Division ausgenutzt werden.

Der Einsatz von Hash-Division ist dadurch eingeschränkt, dass der Algorithmus einen Divisionsvorgang nur über einem konstanten Divisor ausführen kann. Diese Einschränkung ist gravierend, da sie die effiziente Anwendung von Hash-Division bei Anfragen vom Typ 12 ausschließt, der den allgemeinsten und interessantesten Anfragetyp darstellt! Wie in Abschnitt 2.1.1 dargestellt, ist es zwar möglich, solche Anfragen durch eine relationale Division zu beschreiben. Die Ausführungspläne, die sich ergeben, sind jedoch alles andere als effizient. Das folgende Beispiel verdeutlicht diesen Sachverhalt.

Beispiel 3.2

An die Datenbank aus Beispiel 2.7 wird die Anfrage gestellt, welche Studenten in ihrem Studiengang bereits alle Prüfungen abgelegt haben (Anfragetyp 12). Zur kompakten Darstellung wurde aus den Relationen *Studenten* und *Abgelegt* die Relation *Abg2(Sname, Studium, Pnr)* erzeugt. Sie enthält Duplikate, wenn ein Student eine Prüfung wegen Nicht-Bestehens mehrmals abgelegt hat. Aufgrund referentieller Integrität besteht zwischen den Attributen *Sname* und *Studium* eine n:1-Beziehung; jeder Student ist in genau einem Studiengang eingeschrieben. Von der Relation *Prüfungen* werden nur die Attribute *Pnr* und *Studium* dargestellt.

Abg2	Sname	Studium	Pnr
	Tina	B	9
	Mark	I	2
	Mark	I	3
	Max	M	5
	Alex	I	5
	Alex	I	1
	Tina	B	8
	Alex	I	3
	Mark	I	1
	Max	M	6
	Tina	B	3
	Mark	I	4
	Alex	I	2
	Tina	B	10
	Alex	I	6
	Mark	I	1

Tabelle 3.5: Abgelegte Prüfungen

Prüf	Studium	Pnr
	I	1
	I	2
	I	3
	I	4
	M	5
	M	6
	M	7
	B	8
	B	9
	B	10

Tabelle 3.6: Prüfungen zu den Studiengängen

Tabelle 3.5 enthält die Studenten Alex und Mark (beide Informatik), Max (Mathematik) und Tina (Biologie). Mark und Tina haben alle Prüfungen ihres Studienganges abgelegt, Alex und Max fehlen j e eine Prüfung. In Tabelle 3.6 sind die Prüfungen der drei Studiengänge aufgeführt.

Es ist schwierig, eine einzige relationale Division zu finden, die die Anfrage löst. Der direkte Weg, um das Ergebnis zu ermitteln, besteht darin, Div idend und Divisor nach Studiengängen zu gruppieren und für jeden Studiengang eine relationale Division durchzuführen. Hash -Division ist dazu aber nicht in der Lage und kann bei solchen Anfragen nicht direkt angewandt werden. Die Anfrage kann dennoch mit Hash-Division bearbeitet werden, wenn der Dividend entsprechend aufbereitet wird:

$$Q = (\pi_{A_{out}}(R) \bowtie_{\neg P_B \vee P_Q} D) \div D$$

Dieser Ausdruck ist aus der allgemeinen Allquantor -Anfrage mit Bereichs - und Quantorprädikat hergeleitet. Im Beispiel ist das Ergebnisattribut $A_{out} = \{Sname\}$. Das Bereichsprädikat $P_B = (Abg2.Studium = Prüf.Studium)$ und wählt zu jedem Kandidaten $q \in \pi_{A_{out}}(R)$ den passenden Divisor $D(q) = \{d \in D \mid P_B(q, d)\}$ aus der Divisormenge D aus. Das Quantorprädikat P_Q lautet $(Abg.Pnr = Prüf.Pnr)$ und prüft zu einem Studenten, ob er alle Prüfungen seines Studienganges abgelegt hat.

Die Join -Operation erzeugt den Dividenden S der relationalen Division. Ist $P_B(r, d)$ bezüglich der Tupel $r \in R$, $d \in D$ erfüllt, werden r und d miteinander verknüpft, wenn sie im Attribut *Pnr* übe reinstimmen (P_Q). Ist $P_B(r, d)$ nicht erfüllt, werden beide Tupel unabhängig von P_Q miteinander verknüpft. Jedes Tupel aus R wird also mit allen Tupeln aus D kombiniert, die zu einem anderen Studiengang gehören, und im selben Studien -

gang mit dem Tupel, das dieselbe Prüfungsnummer hat. Der Join vergrößert den Dividend annähernd auf das kartesische Produkt $R \times D$, wobei Duplikate erzeugt werden. Wie die Aufwandsabschätzungen im nächsten Abschnitt zeigen, ist der Ausführungsplan aus diesem Grund ineffizient.

Bei der anschließenden Division durch ganz D geben bei jedem Kandidaten $q \in \pi_{A_{out}}(S)$ letztlich diejenigen Tupel den Ausschlag, die aus Kombinationen mit Tupeln desselben Studienganges hervorgegangen sind. Ein Student wird nur dann in die Ergebnismenge Q übernommen, wenn er zu allen Prüfungen seines Studienganges einen Eintrag in S besitzt. (Der Dividend S zu diesem Beispiel ist aufgrund seiner Größe im Anhang aufgeführt.)

Hash-Division ist ein blockierender Operator. Er produziert sein Ergebnis beim abschließenden Lesen der Kandidatentabelle Q_C. Die Variante Incremental Hash-Division blockiert nicht und liefert das Ergebnis als kontinuierlichen Datenstrom. Sie kann im Operatorbaum der Anfrage zum nachfolgenden Operator eine Pipeline aufbauen. Sowohl Hash-Division als auch Incremental Hash-Division benötigen die Relation D vollständig, um mit der Verarbeitung der Relation R beginnen zu können. Zu dem Operator, der die Relation R erzeugt, können sie dann eine Pipeline aufbauen.

Der Algorithmus ist parallelisierbar. Eine Möglichkeit besteht darin, die Relation R anhand der Belegungen in den Attributen A_Q zu Partitionieren und die Relation D zu kopieren. Jede Partition erzeugt ihr Ergebnis unabhängig. Ist die Relation D auch sehr groß, können beide Relationen nach den Belegungen in den gemeinsamen Attributen (siehe Prädikat P_1) partitioniert werden. Jede Partition liefert eine Menge von Kandidaten als Teilergebnis. Die Schnittmenge aller Teilergebnisse ergibt den Quotienten Q.

Fazit

Hash-Division kann also nur auf Anfragen der Typen 3, 6 und 9 effizient angewandt werden und das Teilprädikat P_2 von P_Q darf sich nicht auf die Divisorrelation D beziehen oder das Teilprädikat P_1 muss ein Schlüsselattribut von D enthalten. Ist seine Anwendbarkeit gegeben, bearbeitet er die in der Anfrage enthaltene Allquantifizierung sehr effizient und setzt keine physischen Eigenschaften bezüglich seiner Eingaberelationen voraus (required physical properties). Er kann flexibel parallelisiert werden. Eine Pipeline kann er nur zu dem Vorgänger aufbauen, der die Relation R erzeugt, nicht aber zu seinem Nachfolger.

Die Variante Incremental Hash-Division erzeugt eine kontinuierliche Ausgabe und blockiert nicht. Sie setzt allerdings bezüglich der Relation R die physische Eigenschaft voraus, dass sie nach den Quotientenattributen A_Q gruppiert oder sortiert vorliegt.

3.2.1. Aufwandsabschätzungen

Die folgenden Abschätzungen dienen dazu, die Aussagen des vorigen Abschnitts zu untermauern und eine theoretische Grundlage für den Vergleich von Hash-Division mit dem Ansatz Equi-Stream-Join im Kapitel 5 zu schaffen. Es werden zwei Szenarien betrachtet. Beim One-Pass Szenario können die Eingaberelationen R und D sowie die Zwischenrelation der Kandidaten Q_C im Datenbankpuffer gehalten werden. Beim Two-

Pass Szenario kann die kleinere Divisorrelation D im Datenbankpuffer gehalten werden; die Relation R und die Zwischenrelation Q_C müssen jedoch auf Platte ausgeschrieben werden. Der Aufwand für das Lesen von R und D von und das Schreiben von Q auf Platte wird in beiden Szenarien nicht berücksichtigt, da er unabhängig vom eingesetzten Algorithmus stets zu erbringen ist.

One-Pass Szenario

Relationale Division und unmittelbare Anwendbarkeit

Der Aufwand zum Aufbau einer Hash-Tabelle der Kardinalität n_1 beträgt $k_H n_1$, wobei $k_H \geq 1$. Die Konstante k_H bringt den Aufwand zum Ausdruck, der durch die Kollisions- bzw. Überlaufbehandlung für die Buckets der Hash-Tabelle entsteht. Sie beträgt bei gut gewählten Hash-Funktionen ca. 1,2. Das Proben einer Relation der Kardinalität n_2 mit der Hash-Tabelle verursacht einen Aufwand von $k_H n_2$.

Seien $n_R = |R|$, $n_D = |D|$ und $n_C = |Q_C|$ die Kardinalitäten der Eingaberelationen und der Kandidatenrelation mit $n_D < n_R$ und $n_C < n_R$. Für Hash-Division ergibt sich bei direkter Anwendbarkeit (Typen 3, 6 und 9) ein Aufwand von:

(1) Aufbau der Hash-Tabelle D des Divisors:	$k_H\, n_D$
(2) Aufbau der Hash-Tabelle Q_C der Kandidaten:	$k_H\, n_C$
Proben des Dividenden R mit D:	$k_H\, n_R$
Proben des Dividenden R mit Q_C:	$k_H\, n_R$
(3) Scannen von Q_C zur Erzeugung des Quotienten Q:	$1{,}0\, n_C$
Gesamtaufwand:	$2\, k_H\, n_R + (1 + k_H)\, n_C + k_H\, n_D$
mit $k_H = 1{,}2$:	$2{,}4\, n_R + 2{,}2\, n_C + 1{,}2\, n_D$

Die Variante Incremental Hash-Division benötigt keine Hash-Tabelle Q_C der Kandidaten. Im Vergleich zum Standardalgorithmus ist der Aufwand nur noch etwa halb so groß:

(1) Aufbau der Hash-Tabelle D des Divisors:	$k_H\, n_D$
(2) Proben der Dividenden R(q) mit D:	$k_H\, n_R$
Gesamtaufwand:	$k_H\, n_R + k_H\, n_D$
mit $k_H = 1{,}2$:	$1{,}2\, n_R + 1{,}2\, n_D$

Relationale Division und keine unmittelbare Anwendbarkeit

Liegt eine Anfrage vom Typ 12 vor, bei dem Hash-Division nicht direkt angewandt werden kann, müssen der Aufwand der Join-Operation zur Erzeugung des Dividenden S und dessen höhere Kardinalität zusätzlich berücksichtigt werden. Die Kardinalität von S liegt im Allgemeinen unwesentlich unter der des kartesischen Produkts $R \times D$. Sie betrage ebenso wie der Aufwand der Join-Operation $n_R\, n_D$. Unter der begünstigenden Annahme, dass die Relation S trotz ihrer Größe im Datenbankpuffer gehalten werden kann, ergibt sich ein Aufwand von:

(1) Aufbau des Dividenden S:	$n_R\ n_D$
(2) Aufbau der Hash-Tabelle D des Divisors:	$k_H\ n_D$
(3) Aufbau der Hash-Tabelle Q_C der Kandidaten:	$k_H\ n_C$
Proben des Dividenden S mit D:	$k_H\ n_R\ n_D$
Proben des Dividenden S mit Q_C:	$k_H\ n_R\ n_D$
(4) Scannen von Q_C zur Erzeugung des Quotienten Q:	$1{,}0\ n_C$
Gesamtaufwand:	$(1 + 2\ k_H)\ n_R\ n_D + (1 + k_H)\ n_C + k_H\ n_D$
mit $k_H = 1{,}2$:	$3{,}4\ n_R\ n_D + 2{,}2\ n_C + 1{,}2\ n_D$

Die direkte Herangehensweise, bei der Dividend und Divisor zuerst gruppiert werden und pro Gruppe eine relationale D ivision durchgeführt wird, verspricht bei solchen Allquantor-Anfragen eine wesentlich effizientere Ausführung als Hash -Division. Der Aufwand, um eine Relation mit Kardinalität n durch Hashing zu gruppieren, betrage ebenfalls $k_H\ n$. Der Aufwand, um k relatio nale Divisionen mit durchschnittlichen Kardinalitäten (n_R/k) und (n_D/k) für Dividend und Divisor durchzuführen, entspricht aufgrund von Linearität dem Aufwand einer relationalen Division mit Kardinalitäten n_R und n_D. Somit ergibt sich bei diesem Vorgehen ein Aufwand von:

(1) Gruppieren der Dividendenmenge R:	$k_H\ n_R$
(2) Gruppieren der Divisormenge D und Aufbau der Hash-Tabellen D_G der Divisoren:	$k_H\ n_D$
(3) Aufbau der Hash-Tabellen $Q_{C,G}$ der Kandidaten:	$k_H\ n_C$
Proben der Dividenden R_G mit D_G:	$k_H\ n_R$
Proben der Dividenden R_G mit $Q_{C,G}$:	$k_H\ n_R$
(4) Scannen der $Q_{C,G}$ zur Erzeugung des Quotienten Q:	$1{,}0\ n_C$
Gesamtaufwand:	$3\ k_H\ n_R + (1 + k_H)\ n_C + k_H\ n_D$
	$3{,}6\ n_R + 2{,}2\ n_C + 1{,}2\ n_D$

Das Gruppieren der Divisormenge D kann zusammen mit dem Aufbau der Hash - Tabellen der Divisoren durch eine Hash-Funktion vollzogen werden. Man spart dadurch einen Aufwand von $k_H\ n_D$. Der von Hash -Division verursachte Aufwand liegt ziemlich genau um den Faktor n_D und damit um mehrere Größenordnungen über dem Aufwand der direkten Herangehensweis e. Es bestätigt sich die Aussage von [9], dass Hash - Division bei Anfragen mit variierendem Divisor nicht effizient ist.

Two-Pass Szenario

Wenn Algorithmen auf Platte arbeiten müssen, interessiert nicht mehr die Komplexität eines Algorithmus in Bezug auf die Ein - und Ausgabekardinalitäten. Die Anzahl der Lese- und Schreibzugriffe auf den externen Speicher (E/A -Operationen) entscheidet über die Effizienz solcher Algorithmen, da eine E/A -Operation um etwa fünf Größen - ordnungen länger dauert als eine Operation im Hauptspeicher. Eine E/A -Operation liest oder schreibt einen Block. $B(R)$ gibt die Anzahl der Blöcke an, in denen die Relation R auf Platte abgelegt ist. M gibt den zur Verfügung stehenden Datenbankpuffer in Seiten

an, wobei die Größe einer Seite mit der eines Blockes übereinstimmt. Hash-Division kann als Two-Pass Algorithmus realisiert werden, wenn D im Hauptspeicher gehalten werden kann, für Q_C eine freie Seite und für die Relation R insgesamt M freie Seiten mit $M < B(R) \leq M^2$ und für Q_C zur Verfügung stehen.

Die Dividendenrelation R wird partitionsweise verarbeitet, da sie zu groß für den Datenbankpuffer ist. Ein hash-basierter Algorithmus gruppiert sie nach den Attributen A_{out} in die verfügbaren M Buckets. Ist eine Seite voll, wird sie ausgeschrieben. Das Gruppieren erfordert $B(R)$ Lese- und $B(R)$ Schreiboperationen. Die auf Platte befindlichen Buckets umfassen durchschnittlich $B(R)/M$ Blöcke. Zur weiteren Verarbeitung muss jedes Bucket in den Hauptspeicher passen, weshalb für alle Buckets $B(R)/M \leq M$ gelten muss. Daraus leitet sich die Voraussetzung $B(R) \leq M^2$ für die Anwendbarkeit des Algorithmus ab.

Die Divisorrelation D wird eingelesen und als Hash-Tabelle im Hauptspeicher gehalten. Es fallen $B(D)$ Leseoperationen an. Die gruppierte Relation R wird bucketweise in den Hauptspeicher gelesen. Ein Bucket K enthält aufgrund der Gruppierung die Dividenden $R(q)$ einer Menge von Kandidaten $q \in Q_K$. Alle Tupel eines Kandidaten q befinden sich also stets im selben Bucket. Die Verarbeitung eines Buckets erfolgt wie beim One-Pass Algorithmus anhand der beiden Hash-Tabellen D und Q_K. Die Hash-Tabelle von Q_K kann im Worst Case annähernd so groß werden wie das Bucket K. Sie kann aber nicht schneller wachsen, als K freigegeben wird, da ein Tupel aus K höchstens einen neuen Kandidaten in Q_K produzieren kann. Nach der Verarbeitung des Buckets wird Q_K gelesen und die Kandidaten, deren Matrix nur Eisen enthält, werden als Ergebnis des Buckets ausgeschrieben. Q_K wird freigegeben und das nächste Bucket eingelesen. Es fallen $B(R)$ Lese- und $B(Q)$ Schreiboperationen an.

Vom Gesamtaufwand des Two-Pass Algorithmus für Hash-Division werden die Operationen für das Einlesen der Relationen R und D sowie das Ausschreiben der Relation Q abgezogen, um den algorithmus-spezifischen Anteil an E/A-Operationen zu ermitteln:

E/A Gesamt:	3 B(R) + B(D) + B(Q)
E/A Hash-Division	2 B(R)

Anmerkung zu Paper [9]

In [9] wird die Ausführung von Allquantor-Anfragen durch den Operator Anti-Semi-Join vorgestellt und mit anderen Ansätzen, darunter auch Hash-Division, verglichen. Anti-Semi-Join geht als effizientester Ansatz aus den Vergleichen hervor. Die gemachten Aussagen hinsichtlich Vorgehensweise und Effizienz dieses Ansatzes gelten allerdings nur für objektorientierte Datenbanksysteme und sind nicht auf relationale Systeme übertragbar. Dies wird deutlich, wenn man die relationenalgebraische Darstellung der allgemeinen Allquantor-Anfrage in einen Ausführungsplan überführt, der auf dem Operator Anti-Semi-Join basiert.

Darstellung mit negiertem Existenzquantor:

$$Q = \{q \in \pi_{A_{out}}(R) \mid \neg\exists d \in D(q)\ \neg\exists r \in R(q): P_Q\}$$

Darstellung mit Anti-Semi-Join $\overline{\ltimes}$:

$$Q = \{q \in \pi_{A_{out}}(R) \mid \neg\exists d \in D(q)\ \neg\exists r \in R(q): P_Q\}$$

$$= \{q \in \pi_{A_{out}}(R) \mid \neg\exists d \in (D(q) \overline{\ltimes}_{P_Q} R(q))\}$$

$$= \pi_{A_{out}}(R) \overline{\ltimes} (D(q) \overline{\ltimes}_{P_Q} R(q))$$

Die Darstellung der Anfrage mit Anti-Semi-Join führt bei relationalen Datenbanksystemen zu einem Ausführungsplan mit zwei Anti-Semi-Join Operatoren. In objektorientierten Systemen enthält der Ausführungsplan dagegen nur einen Anti-Semi-Join Operator. Der Grund dafür ist, dass es in relationalen Systemen keine mengenwertige Attribute gibt.

Der Vergleich mit Hash-Division und anderen Verfahren relationaler Systeme wird in [9] nicht korrekt vollzogen. Es wird vorausgesetzt, dass zu den Objekten die entsprechenden mengenwertigen Attribute vorhanden sind. Der Existenz dieser mengenwertigen Attribute entspricht in relationalen Systemen die physische Eigenschaft, dass der Dividend R bereits nach A_{out} gruppiert vorliegt. Für einen korrekten Vergleich müsste daher entweder der Aufwand zur Erstellung der mengenwertigen Attribute mit berücksichtigt werden oder im relationalen System von bereits gruppierten Dividenden ausgegangen werden. Dies wird jedoch nicht gemacht. Die in den mengenwertigen Attributen versteckte physische Eigenschaft einer vorhandenen Gruppierung verschafft so dem Anti-Semi-Join im Vergleich mit den anderen Algorithmen einen entscheidenden Effizienzvorteil, der allerdings nicht auf den Algorithmus sondern auf die unterschiedlichen Vergleichsbedingungen zurückzuführen ist.

4. Der Ansatz Equi-Stream-Join

In diesem Kapitel wird mit Equi -Stream-Join ein neuer Ansatz für relationale Datenbanksysteme vorgestellt, um Allquantifizierung in Anfragen zu bearbeiten. Der Ansatz beinhaltet einen logischen Operator, der im folgenden Abschnitt formal beschrieben wird. Abschnitt 4.2 widmet sich der algorithmischen Realisierung des Operators. Der zugrunde liegende Algorithmus wird spezifiziert und effiziente Implementierungsmöglichkeiten werden angegeben. Eigenschaften des Algorithmus werden beschrieben und sein Aufwand bei der Bearbeitung von Allquantor-Anfragen wird abgeschätzt. Beispiele zeigen die konkrete Anwendung des Operators.

4.1. Logischer Operator

Equi-Stream-Join ist ein unärer Operator zur Ausführung von Allquantifizierung. Dies mag zunächst überraschen, da Allquantifizierung in ihrer allgemeinen Form eine Operation ist, die aus zwei Eingaberelationen R und D eine Ergebnisrelation erzeugt (siehe Kapitel 2). Equi-Stream-Join arbeitet logisch betrachtet auch mit den Mengen R und D bzw. mit den in ihnen enthaltenen Dividenden und Divisoren. Er unterscheidet sich aber von allen anderen Ansätzen darin, dass er die Mengenpaare von Dividenden und Divisoren aus seiner Eingaberelation R_{in} durch geeignete Strukturierung erzeugt. Die Information, die er für diese Strukturierung benötigt, wird ihm durch drei Attributmengen A_G, A_R, A_D übergeben. Die Allquantifizierung führt er unter Verwendung eines Prädikates P über den erzeugten Mengenpaaren durch.

Bezeichne A die Attribute der Eingaberelation R_{in}. Für die Gruppenattribute A_G, die Dividendenattribute A_R und die Divisorattribute A_D gilt $A_G, A_R, A_D \subseteq A$, da der Operator keine neuen Attribute erzeugt bzw. berechnet. Für Equi -Stream-Join wird folgende Signatur verwendet:

$$=\triangleright(R_{in}, A_G, A_R, A_D, P)$$

Sinn und Funktion der in der Signatur angegebenen Parameter ergeben sich im Zusammenhang mit der Allquantifizierung. Ausgehend von zwei Relationen $R(A_{out}, X)$ und $D(Y)$ prüft eine Allquantifizierung für jeden Kandidaten $q \in \pi_{A_{out}}(R)$ den Ausdruck $\forall d \in D(q)\ \exists r \in R(q): P_Q$. Ihre Parameter sind die Relationen R und D, die Prädikate P_B und P_Q sowie die Ausgabeattribute A_{out}. Sie sind auf die Parameter von Equi-Stream-Join abzubilden. Für die Ausgabeattribute gilt:

$$A_{out} = A_G \cup A_R$$

Sie werden in die Attributmengen A_G und A_R aufgeteilt. Den Grund dafür liefern Anfragen vom Typ 12. Das Bereichsprädikat P_B solcher Anfragen enthält Attribute der Relation D und Attribute $A_B \subseteq A_{out}$ der Relation R. Die Divisoren $D(q)$ variieren in Abhängigkeit der Attribute A_B. Zwei Kandidaten q_1 und q_2 besitzen denselben Divisor $D(q_1)$, wenn sie in A_B übereinstimmen; ihre Divisoren sind verschieden, wenn $q_1.A_B \neq q_2.A_B$ ist. Equi-Stream-Join fasst alle Kandidaten mit gleichem Divisor zusammen. Die Aufteilung von A_{out} in die Mengen A_G und A_R erfolgt daher anhand der in P_B enthaltenen Attribute der Relation R:

$$A_G = A_B \subseteq A_{out}$$

$$A_R = A_{out} - A_B$$

Die Ausgabeattribute sind damit vollständig abgebildet. Die übrige im Bereichsprädikat enthaltene Information zur Auswahl der Divisoren wird bei der Generierung der Eingaberelation R_{in} aus den Relationen R und D verwendet. Dieser Vorgang wird später erläutert, da man die Funktionsweise von Equi-Stream-Join kennen muss, um ihn zu verstehen. Da Equi-Stream-Join ebenfalls relationale Divisionen ausführt, wird das Quantorprädikat P_Q in zwei Teilprädikate P_1 und P_2 zerlegt, wie in Kapitel 3 beschrieben: P_1 enthält alle Gleichheitsbedingungen der Form $(r.x_i = d.y_j)$ mit den Attributen $x_i \in X_1 \subseteq X$ und $y_j \in Y_1 \subseteq Y$. Das Teilprädikat P_2 enthält alle übrigen Bedingungen des Prädikats P_Q. Das so zerlegte Quantorprädikat P_Q wird auf die Attributmenge A_D und das Prädikat P abgebildet:

$$A_D = X_1 \text{ oder } Y_1$$

$$P = P_2$$

Das Quantorprädikat wird ebenfalls zur Generierung der Eingaberelation herangezogen. Von den Attributmengen X_1 und Y_1 wird wie bei einem natürlichen Gleichverbund nur eine nach R_{in} übernommen, um Redundanz zu vermeiden. Equi-Stream-Join arbeitet mit Gruppierungen und benötigt für eine gegebene Eingaberelation R_{in} lediglich die Information, in welchen Attributen die ursprünglichen Relationen R und D übereinstimmen müssen. Die anderen in P_Q enthaltenen Bedingungen bilden das Prädikat P.

Die Attributmengen A_G und A_R sind per Definition disjunkt. Die Attributmenge A_D kann keine Attribute aus A_{out} enthalten, da in P_1 nur die Gleichheitsbedingungen aus P_Q aufgenommen werden, deren eines Attribut aus der Menge X stammt. Damit sind die Attributmengen A_G, A_R, A_D paarweise disjunkt.

4.1.1. Funktionsweise

Im vorigen Abschnitt wurde beschrieben, mit welchem Inhalt die Attributmengen A_G, A_R, A_D und das Prädikat P zur Bearbeitung einer Allquantifizierung belegt werden. Dieser Abschnitt widmet sich der Vorgehensweise des Operators und erläutert, wie Equi-Stream-Join die in seinen Parametern enthaltene Information umsetzt.

In der Eingaberelation R_{in} sind die Divisoren $R(q)$ der Kandidaten q enthalten, die durch die Allquantifizierung überprüft werden (siehe Abschnitt 2.1.1). Die Divisoren sind in der durch $R_{in}.A_D$ gegebene Menge von Belegungen enthalten, weshalb die Attribute A_D als Divisorattribute bezeichnet werden. Um Dividenden mit gleichem Divisor zusammenzufassen, partitioniert der Operator seine Eingabe R_{in} zunächst anhand von A_G in Gruppen G_i. Eine Gruppe G_i kann Dividenden von mehreren Kandidaten $q \in G_i.A_{out}$ enthalten. Sie enthält aber stets nur einen konstanten Divisor. Über diesen Divisor wird die Allquantifizierung der in der Gruppe enthaltenen Dividenden durchgeführt. A_G ist leer, wenn sich das Bereichsprädikat der Allquantifizierung nicht auf beide Relationen R und D bezieht (Typen 1-9). Equi-Stream-Join interpretiert dann die gesamte Eingaberelation R_{in} als eine Gruppe und führt die Allquantifizierung über den einzigen vorhandenen Divisor durch.

Um innerhalb einer Gruppe G die einzelnen Dividenden und den Divisor zu erzeugen, strukturiert der Operator seine Eingabe durch eine zweidimensionale Gruppierung nach den Attributen A_R und A_D. Die logische Gruppierung nach A_R unterteilt die Gruppe in Dividenden $R(q)$. Die Menge $G.A_D$ der Belegungen von A_D bildet den Divisor. Die physische Gruppierung nach A_D unterteilt die Gruppe in Streams S_j. Zu jedem Element des Divisors wird ein Stream aus denjenigen Dividendentupeln gebildet, die mit dem Element in A_D übereinstimmen.

Equi-Stream-Join realisiert die Allquantifizierung innerhalb einer Gruppe für einen Kandidaten q, indem für den zugehörigen Dividenden $R(q)$ geprüft wird, ob in jedem Stream S_j ein Tupel von $R(q)$ enthalten ist, welches das Prädikat P erfüllt. Existiert ein solches Tupel in einem Stream, so sagt man auch, der Dividend $R(q)$ erfüllt das Prädikat P für den Stream S_j. Die Kandidaten deren Dividenden P für alle Streams S_j erfüllen, werden in die Ergebnisrelation R_{out} eingetragen. Sei $J(G)$ die Indexmenge der Streams der Gruppe G, so gilt für die Kandidaten der Gruppe:

$$q \in R_{out} \Leftrightarrow \forall j \in J(G)\ \exists t \in G: t \in R(q) \wedge t \in S_j \wedge P(t)$$

Die Allquantifizierung von ganz R_{in} wird gruppenweise durchgeführt. Es wird geprüft, welche Dividenden $R(q)$ einer Gruppe G_i für alle Streams $S_{i,j}$ der Gruppe das Prädikat P erfüllen. Die Dividenden müssen nicht mit dem Gruppenindex i versehen werden, da durch die Belegung $q.A_G$ des Kandidaten bereits die Zuordnung seines Dividenden $R(q)$ zur entsprechenden Gruppe G_i gegeben ist. Sei $I(R_{in})$ die Indexmenge der Gruppen der Eingaberelation, so gilt für ganz R_{in}:

$$q \in R_{out} \Leftrightarrow \exists i \in I(R_{in}):\ \{q.A_G\} = G_i.A_G \ \wedge\ \forall j \in J(G_i)\ \exists t \in R(q) \wedge t \in S_{i,j}:\ P(t)$$

Die Menge A_R ist leer, wenn alle Attribute A_{out} im Bereichsprädikat enthalten sind und auf A_G abgebildet werden. Jeder Dividend $R(q)$ wird durch einen anderen Divisor $D(q)$ geteilt. Die einzelnen relationalen Divisionen der Allquantifizierung können dann nicht zusammengefasst werden. Da A_{out} nicht leer sein darf, ist mindestens eine der Mengen A_G, A_R nicht leer, und die Ausgabe von Equi-Stream-Join ist definiert.

Das Prädikat P fehlt, wenn das Quantorprädikat P_Q nur Gleichheitsbedingungen zwischen Dividenden - und Divisorrelation enthält und vollständig auf die Attributmenge A_D abgebildet wird. Eine solche Allquantifizierung kann durch eine relationale Division ohne Prädikat ausgeführt werden. Equi-Stream-Join muss für jede Gruppe lediglich prüfen, welche Dividenden in allen Streams mit mindestens einem Tupel vertreten sind.

Die Menge der Divisorattribute A_D ist leer, wenn P_Q keine Gleichheitsbedingung zwischen den Relationen R und D enthält. Die Allquantifizierung ist dann keine relationale Division sondern von einem der einfachen Typen 1, 2, 4, 5, 7, 8, 10 oder 11. Für Equi-Stream-Join stellt dies einen Sonderfall dar, da der Operator in erster Linie zur Ausführung relationaler Divisionen entworfen wurde. Solche Allquantifizierungen sind jedoch leicht zu handhaben. Die Gruppierung in Streams entfällt, da es keinen Divisor gibt. Die Eingabe R_{in} wird stattdessen durch $A_G \cup A_R = A_{out}$ in die in ihr enthaltenen Dividenden $R(q)$ partitioniert. Für alle Tupel eines Dividenden ist lediglich zu prüfen, ob sie das Prädikat P erfüllen. Wenn ja, wird der zugehörige Kandidat q ausgegeben.

Beispiel 4.1

Anhand eines einführenden Beispiels wird die Anwendung von Equi-Stream-Join auf eine Allquantifizierung gezeigt und seine Funktionsweise anhand der Strukturierung der Dividenden in Streams verdeutlicht. Die Struktur Gruppen wird im Beispiel nicht benötigt. Als Szenario betrachten wir eine Übungsgruppe von Studenten. Gegeben sei die Liste *Übungen* mit den Attributen *Matrikelnummer*, *Termin* und *abgegebene Aufgaben*.

Übungen (Mnr, Termin, Abgabe)

Sie gibt an, welche Studenten an welchen Terminen wie viele Übungsaufgaben abgegeben haben. An jedem Termin werden sechs Aufgaben besprochen, so dass ein Student zwischen null und sechs Aufgaben abgeben kann. Die Anfrage ermittelt die Matrikelnummern derjenigen Studenten, die an allen Terminen mindestens vier Aufgaben abgegeben haben.

```
select  distinct Mnr
from    Übungen as Q
where   for all  (select  distinct Termin
                  from    Übungen as D)
                 (exists  (select  *
                           from    Übungen as R
                           where   R.Mnr = Q.Mnr  and
                                   R.Termin = D.Termin  and  R.Abgabe ≥ 4 ) )
```

Die Dividenden - und Divisorrelation der Allquantifizierung werden aus derselben physischen Relation *Übungen* gewonnen. Die SELECT-Klausel definiert die Studenten als Kandidaten q. Die erste Unteranfrage erzeugt den Divisor als Menge der Übungstermine. Es gibt kein Bereichsprädikat, da die Übungstermine für alle Studenten dieselben sind. Die zweite Unteranfrage erzeugt den Dividenden $R(q)$ zu einem Studenten q. Im Dividenden ist zu jedem Termin ein Eintrag mit der Anzahl der abgegebenen Aufgaben enthalten. Die Anfrage ist vom Typ 3.

Für die Anwendung von Equi-Stream-Join gilt:

A_{out} = Mnr, P_B fehlt $\Rightarrow$ $A_G = \emptyset$ und A_R = Mnr

P_1 = (R.Termin = D.Termin) $\Rightarrow$ A_D = Termin und P = (R.Abgabe $\geq$ 4)

Möchte man in SQL die Möglichkeit bieten, Allquantifizierungen in der Logik von Equi-Stream-Join zu formulieren, ist die Syntax von SQL um ein sprachliches Konstrukt mit der Signatur des Operators zu erweitern:

```
eqStreamJoin R_in
group by     A_G
stream by    A_D
get          A_R
where        P
```

Die alternative Formulierung der Anfrage mit Equi-Stream-Join lautet:

```
select *
from   (eqStreamJoin Übungen
        stream by    Termin
        get          Mnr
        where        Abgabe ≥ 4)
```

Sie ist wesentlich kompakter als die ursprüngliche Anfrage, da der Operator bei Allquantor-Anfragen mit nur einer physischen Relation unmittelbar angewandt werden kann. Die Kandidaten werden durch die Attribute $A_G \cup A_R$ = *Mnr* definiert. Der Divisor wird durch das Divisorattribut A_D = *Termin* gebildet. Equi-Stream-Join gruppiert die Eingaberelation *Übungen* nach *Termin* in Streams S_j. Das Prädikat *P* enthält die Bedingung, die für einen Kandidaten *q.Mnr* bezüglich aller Streams S_j zu prüfen ist. Seine Matrikelnummer wird als Ergebnis ausgegeben, wenn *P* für alle Streams wahr ist.

4.2. Entwurf des Algorithmus

Equi-Stream-Join soll alle zwölf Typen von Allquantor-Anfragen effizient bearbeiten können. Relationale Divisionen sind die aufwendigsten Allquantifizierungen, da die Dividenden und Divisoren durch Unteranfragen erzeugt werden müssen und die Unteranfrage der Dividenden in jedem Fall korreliert ist. Sie enthalten das größte Optimierungspotential, weshalb ihre effiziente Ausführung beim Entwurf des Algorithmus im Vordergrund steht.

4.2.1. Vorüberlegungen

Gegeben sei eine Eingaberelation R_{in} mit den Attributen A. Sie wird von Equi -Stream-Join durch A_G in Gruppen G_i partitioniert und gruppenweise verarbeitet. Innerhalb der Gruppen ist die zweidimensionale Gruppierung nach A_D und A_R vorzunehmen. Eine Relation bzw. eine Gruppe ist eine lineare Datenstruktur und kann zu einem Zeitpunkt immer nur nach einer Attributmenge physisch gruppiert werden. Es ist zu entscheiden, nach welcher Attributmenge gruppiert werden soll. Da die Bearbeitung relationaler Divisionen Vorrang hat, gruppiert Equi -Stream-Join seine Eingaberelation R_{in} grundsätzlich nach den Divisorattributen A_D. Durch die Strukturierung in Streams $S_{i,j}$ erzeugt er zu jeder Gruppe G_i den Divisor. Wenn A_D leer ist, liegt keine relationale Division vor . Das Zusammenfassen von Dividenden mit gleichem Divisor in einer Gruppe ist hinfällig, da es keine Divisoren gibt. In diesem Fall gruppiert Equi -Stream-Join R_{in} nach $A_G \cup A_R = A_{out}$.

Bei relationalen Divisionen werden die Tupel der Dividenden durch die phys ische Gruppierung nach A_D in den Streams der Gruppe verteilt angeordnet. Die logische Gruppierung nach A_R in Dividenden $R(q)$ muss auf andere Weise erfolgen. Der Algorithmus verwendet hierzu ein internes Prädikat P_R. Es enthält alle Attribute aus A_R und übe rprüft bezüglich zweier Tupel t_1 und t_2, ob sie in den Attributen A_R überein - stimmen. P_R ist genau dann wahr, wenn beide Tupel zu demselben Dividenden gehören. Beispielsweise hat es für $A_R = \{a_1, a_2\}$ die Form:

$$P_R(t_1, t_2) = ((t_1.a_1 = t_2.a_1) \wedge (t_1.a_2 = t_2.a_2))$$

Zum Bearbeiten einer Gruppe G_i benötigt der Algorithmus eine Menge, in der die Kandidaten q der Gruppe abgelegt werden. Sie soll nur die Kandidaten enthalten, die sich noch nicht disqualifiziert haben, und wird daher mit *IR* für **I**ntermediate **R**esult bezeichnet. Der Kern des Algorithmus besteht darin, die Kandidaten von *IR* jeweils mit den Kandidaten zu vergleichen, die in einem Stream enthalten sind. Der Vergleich von *IR* mit einem Stream $S_{i,j}$ erfolgt tupelweise: Für jedes Tupel $q \in IR$ wird geprüft, ob ein Tupel $t_S \in S_{i,j}$ existiert, das $P_R(q, t_S)$ und $P(t_S)$ erfüllt. Dem tupelweisen Vergleich liegt also ein zusammengesetztes Prädikat $P_J(q, t_S) = P_R(q, t_S) \wedge P(t_S)$ zugrunde. Der Kandidat q erfüllt die Bedingung P für den Stream $S_{i,j}$ genau dann, wenn

$$\exists t_S \in S_{i,j}: P_J(q, t_S)$$

In Bezug auf *IR* und die Streams entspricht P_J einem Joinprädikat, da es zu einem Tupel aus *IR* prüft, ob im jeweiligen Stream ein korrespondierendes Tupel enthalten ist. Der Algorithmus erzeugt sein Ergebnis durch sukzessive Vergleiche von *IR* m it allen Streams der Gruppe. Die Kandidatenmenge *IR* enthält dabei nur diejenigen Kandidaten, die die Bedingung P für alle Streams erfüllt haben, mit denen *IR* bis dahin verglichen worden ist. *IR* stellt also zu jedem Zeitpunkt das Zwischenergebnis des Algori thmus dar.

Anhand einer zweidimensionalen Matrix kann man den Vorgang veranschaulichen, der jeder Allquantifizierung mit Existenzquantor zugrunde liegt und im Algorithmus durch die zweidimensionale Gruppierung umgesetzt wird. Tabelle 4.1 stellt eine solche Matrix für eine Gruppe G_i mit fünf Dividenden dar; der Divisor enthält vier Elemente. Jede Zeile entspricht dem Stream eines Divisorelements und jede Spalte dem Dividenden eines Kandidaten q. Der in einer Zelle $[j, q]$ eingetragene Wahrheitswert gibt an, ob der Dividend $R(q)$ das Prädikat P für den Stream $S_{i,j}$ erfüllt.

$\mathbf{G_i}$	$R(q_3)$	$R(q_1)$	$R(q_5)$	$R(q_2)$	$R(q_4)$
$S_{i,2}$	f	w	w	f	w
$S_{i,3}$	w	f	w	f	w
$S_{i,1}$	f	w	w	w	w
$S_{i,4}$	w	w	w	w	w

Tabelle 4.1: Zweidimensionale Gruppierung von Dividenden

Eine Allquantifizierung durchzuführen bedeutet, dass der Algorithmus prüfen muss, welche Dividenden für alle Streams die Bedingung *P* erfüllen. In der Matrix sind das genau diejenigen Dividenden, die in ihrer Spalte nur den Wert WAHR enthalten. Das Ergebnis der Allquantifizierung (hier q_4 und q_5) hängt nicht davon ab, in welcher Reihenfolge die Dividenden bezüglich der Streams geprüft werden. Entscheidend ist alleine, dass jeder Dividend für alle Streams unter Verwendung des Prädikats *P* geprüft wird. Für den Algorithmus bedeutet dies, dass die Reihenfolge beliebig ist, in der er die Streams zum Vergleich mit *IR* anordnet. Ebenso spielt es beim Vergleich von *IR* mit einem Stream keine Rolle, in welcher Reihenfolge die in *IR* enthaltenen Kandidaten geprüft werden. Die beliebige Anordnung der Dividenden und Streams in der Matrix veranschaulicht diesen Aspekt.

Eine Allquantifizierung vom Typ 12 setzt der Algorithmus dadurch um, dass er jede Gruppe G_i isoliert und somit unabhängig von der Existenz anderer Gruppen bearbeitet. Die Reihenfolge, in der die Gruppen bearbeitet werden, ist aus diesem Grund ebenfalls beliebig.

4.2.2. Spezifikation

Aufbauend auf den Vorüberlegungen kann der Algorithmus von Equi-Stream-Join für relationale Divisionen auf folgende Weise spezifiziert werden:

(0) wenn $A_D \neq \emptyset$,
  gruppiere R_{in} nach A_G in Gruppen G_i
  gruppiere jede Gruppe G_i nach A_D in Streams $S_{i,j}$

(1) initialisiere *IR* mit den Kandidaten des ersten Streams $S_{i,1}$: $IR = unique(\pi_{A_R} \sigma_P(S_{i,1}))$

(2) überprüfe die Kandidaten aus *IR* anhand des nächsten Streams $S_{i,j}$:
  (a) belasse Kandidat *q* in *IR*, wenn $\exists t_S \in S_{i,j}: P_J(q, t_S)$
    ansonsten entferne Kandidat *q* aus *IR*
  (b) wiederhole Schritt (a) für alle anderen Kandidaten $q \in IR$

(3) solange $IR \neq \emptyset$, wiederhole Schritt (2) für alle anderen Streams $S_{i,j}$ der Gruppe G_i

(4) ergänze die in *IR* enthaltenen Kandidaten um die Belegung $G_i.A_G$ der Gruppe und übertrage sie in die Ausgaberelation R_{out}

(5) wiederhole die Schritte (1) bis (4) für alle anderen Gruppen von R_{in}

Nach der Vorbereitungsphase, in der die Eingabe durch zweifache Gruppierung aufbereitet wird, beginnt der Algorithmus mit der Verarbeitung der Gruppe G_1. Er initialisiert die Kandidatenmenge *IR* mit den Kandidaten, die in Stream $S_{1,1}$ enthalten sind. *IR* benötigt lediglich die Attribute A_R und enthält keine Duplikate.

In Schritt (2) werden die Kandidaten aus *IR* bezüglich aller Streams der Gruppe überprüft. Die Überprüfung beginnt mit dem Stream $S_{i,2}$, da *IR* nur diejenigen Kandidaten aus dem ersten Stream enthält, die das Prädikat *P* erfüllen. Für jeden Kandidaten *q* aus *IR* wird geprüft, ob im Stream $S_{i,j}$ mindestens ein Tupel t_S enthalten ist, das $P_J(q, t_S)$ erfüllt. Existiert ein solches Tupel, verbleibt der Kandidat *q* in *IR*; wenn nicht, wird er aus *IR* entfernt und in darauf folgenden Überprüfungen nicht weiter berücksichtigt.

Mit jedem Schritt (2a) kann ein Kandidat aus *IR* entfernt werden. Daher wird nach jedem Vergleich mit einem Stream in (3) geprüft, ob *IR* leer ist. Wenn nicht, wird Schritt (2) wiederholt, bis *IR* für alle Streams der Gruppe überprüft wurde. Das Ergebnis der Gruppe steht in Schritt (4) fest. Es enthält maximal alle Kandidaten, die im ersten Stream enthalten sind, und minimal die leere Menge. Die Ergebnistupel werden um die Gruppenattribute ergänzt und in R_{out} abgelegt. Der Algorithmus fährt mit der nächsten Gruppe fort, bis er alle Gruppen von R_{in} verarbeitet hat.

Für einfache Allquantifizierungen, die keinen Existenzquantor enthalten und somit keine relationale Divisionen sind (Tabelle 2.1, linke und mittlere Spalte), ist eine andere algorithmische Vorgehensweise erforderlich. Das Prädikat *P* ist nur für die Dividenden *R(q)* zu prüfen.

(a) wenn $A_D = \emptyset$,
gruppiere R_{in} nach $A_G \cup A_R$ in Partitionen *R(q)*

(b) überprüfe Prädikat *P* für alle Tupel der Partition $R(q_1)$ des ersten Kandidaten q_1
wenn *P* für alle Tupel erfüllt ist, übertrage Kandidat q_1 in die Ausgaberelation R_{out}

(c) wiederhole Schritt (b) für alle anderen Kandidaten $q_i \in R_{in}$

Beispiel 4.2

Die Anfrage vom Typ 11 aus Beispiel 2.3 soll von Equi-Stream-Join bearbeitet werden. Sie prüft für ein Unternehmen, ob es Abteilungen gibt, deren Abteilungsleiter ein niedrigeres Gehalt bekommen als alle Mitarbeiter derselben Abteilung. Die Eingaberelation R_{in} wird durch einen Gleichverbund der Relationen *Abteilungen* und *Mitarbeiter* über das gemeinsame Attribut *Anr* erzeugt und hat das Schema:

R_{in} (Pnr, Anr, MGehalt, ALeiter, AGehalt)

Im Attribut *MGehalt* sind die Gehälter der Mitarbeiter, in *AGehalt* die der Abteilungsleiter abgelegt. In Schritt (a) wird die Relation nach $A_{out} = \{Anr\}$ gruppiert. Innerhalb jeder Partition wird geprüft, ob das Quantorprädikat $P_Q = (MGehalt > Agehalt)$ für alle Tupel der Partition erfüllt ist. Wenn ja, wird die Abteilungsnummer der Partition ausgegeben.

Zum Erzeugen der Eingaberelation kann wie in diesem Fall eine Join-Operation erforderlich sein. Die Anfrage in Beispiel 2.1 kann unter alleiniger Verwendung der Relation *Mitarbeiter* bearbeitet werden, wenn aufgrund referentieller Integrität beide Relationen im Attribut *Anr* dieselben Belegungen enthalten. In Beispiel 2.4 ist wiederum eine Join-Operation notwendig, obwohl sich die Anfrage ausschließlich auf

die Relation *Mitarbeiter* bezieht. Es ist Aufgabe des Optimierers, beim Generieren von Ausführungsplänen für eine Anfrage den Operator Equi-Stream-Join, wenn notwendig, zusammen mit einer vorgeschalteten Join-Operation in den Plan zu integrieren.

4.2.3. Effiziente Implementierung

In [8] wird gezeigt, dass Hash-Division effizient ist, weil er die zu vergleichenden Tupel aus der Dividenden- und der Divisor-Relation nicht durch Sortieren sondern durch hash-basiertes Gruppieren zusammenführt. Er nutzt die Freiheit, dass das Vergleichen der Tupel in beliebiger Reihenfolge vorgenommen werden kann. Diese Freiheit setzt er in einen Effizienzgewinn um, indem er dem Grundsatz folgt, nur das unbedingt Notwendige zu tun. Im Falle der relationalen Division bedeutet dies, nur zu gruppieren und nicht zu sortieren. Diese Strategie verschafft Hash-Division gegenüber sortierenden Algorithmen eine höhere Effizienz beim Umsetzen der relationalen Division.

Eine Implementierung von Equi-Stream-Join ist ebenfalls effizient, wenn sie derselben Strategie folgt und vorhandene Freiheiten nutzt, um Aufwand zu vermeiden. Die Spezifikation lässt dem Algorithmus die Freiheit, erstens die Gruppen in beliebiger Reihenfolge zu bearbeiten, zweitens innerhalb der Gruppen die Streams in beliebiger Reihenfolge mit der Kandidatenmenge *IR* zu vergleichen und drittens beim Vergleich eines Streams mit *IR* die Tupel aus beiden Mengen wiederum in beliebiger Reihenfolge zusammenzuführen.

Die ersten beiden Freiheiten können dadurch genutzt werden, dass die Eingaberelation nach den Attributen A_G und A_D lediglich gruppiert und nicht sortiert wird. Diese geschachtelte Gruppierung kann physisch in einem Schritt als Gruppierung nach $A_G \cup A_D$ durchgeführt werden, wenn die Priorität der Attribute A_G berücksichtigt wird. Jedes relationale Datenbanksystem verfügt standardmäßig über einen logischen Operator zum Gruppieren. Je nach Größe von R_{in} und dem verfügbaren Datenbankpuffer wird dieser physisch auf einen One- oder Two-Pass Algorithmus abgebildet (siehe [13]), der die Gruppierung vornimmt. Es kommen meist hash-basierte Algorithmen zum Einsatz, da sie linearen Aufwand garantieren und am effizientesten sind, wenn nicht gerade ein passender Index für A_G oder A_D zur Verfügung steht.

Der tupelweise Vergleich von *IR* mit dem Inhalt eines Streams wird logisch durch eine Join-Operation mit dem Joinprädikat P_J durchgeführt. Sie kann durch die bekannten physischen Varianten Nested-Loops-Join, Merge-Join und Hash-Join realisiert werden. Die beste Performance wird meist mit Hash-Join erreicht. Er verursacht stets linearen Aufwand bezüglich seiner Eingabekardinalitäten $|IR|$ und $|S_{i,j}|$. Merge-Join ist nur dann vorzuziehen, wenn die Eingaberelation R_{in} bereits nach den Attributen A_R sortiert ist und die Gruppierung nach $A_G \cup A_R$ diese Sortierordnung erhält bzw. lediglich umkehrt. In diesem Fall sind die Tupel innerhalb der Streams und *IR* bereits nach A_R sortiert und er verursacht geringeren Aufwand als Hash-Join. Ansonsten liegt der Aufwand von Merge-Join in $O(n \cdot log(n))$ aufgrund der vorzunehmenden Sortierung. Nested-Loops-Join führt zu quadratischem Aufwand.

Die folgende Implementierung von Equi-Stream-Join realisiert den in (2) spezifizierten Vergleich der Kandidaten aus *IR* mit den Streams $S_{i,j}$ der Gruppe G_i durch einen Hash-Join-Algorithmus. Build-Relation ist die Kandidatenmenge *IR*. Die zugehörige Hash-Tabelle *H* enthält die Attribute A_R und ein Attribut *c*, das einen Zähler (counter)

realisiert. Sie wird zur Initialisierung mit den Tupeln des ersten Streams aufgebau t. Die Hash-Funktion hängt nur von den Dividendenattributen A_R ab. In der Hash-Tabelle wird zu jedem Tupel t, das P erfüllt, der zugehörige Kandidat $t.A_R$ abgelegt. Sein Zähler $t.c$ wird anfangs auf den Wert Eins gesetzt. Die in (1) spezifizierte Duplikateli minierung kann bei dieser Implementierung zusammen mit dem Aufbau der Hash -Tabelle durchgeführt werden, ohne dass sie zusätzlichen Aufwand verursacht. Wenn beim Belegen der Buckets zwei Kandidaten $t_1.A_R$ und $t_2.A_R$ zusammenfallen, muss im Rahmen der Kollisio nsbehandlung sowieso geprüft werden, ob es sich um denselben Kandidaten handelt. Wenn dem so ist, handelt es sich bei dem Kandidaten $t_2.A_R$, der in der Hash-Tabelle abgelegt werden soll, um ein Duplikat. Er ist bereits eingetragen und Tupel t_2 wird verworfen. Ist $t_1.A_R \neq t_2.A_R$, liegt eine Kollision vor und der Kandidat $t_2.A_R$ wird je nach Hash -Verfahren in einer Überlaufliste oder in einem anderen, freien Bucket untergebracht.

Die Streams der Gruppe sind die Proberelationen. Die Überprüfung der Kandidaten au s IR beginnt durch Proben mit dem jeweils nächsten Stream $S_{i,j}$. Zu jedem Tupel t des Streams, welches das Prädikat P erfüllt, wird über die Hash -Funktion der zugehörige Kandidat q_t mit $q_t.A_R = t.A_R$ gesucht. Existiert q_t, wird anhand der Bedingung $q_t.c = j\text{-}1$ geprüft, ob der Zähler des Kandidaten den Index des vorigen Streams enthält. Wenn ja, hat der Dividend des Kandidaten in allen bisher geprüften Streams das Prädikat P erfüllt. Der Kandidat ist gültig und sein Zähler wird auf den Index j des aktuellen Streams gesetzt. Nach Proben eines Streams $S_{i,j}$ kann so anhand der Zähler festgestellt werden, ob ein Kandidat q gültig ist $(q.c = j)$. Enthält sein Zähler einen Wert $w < j$, ist der Kandidat ungültig. Sein Dividend war im Stream $w+1$ nicht vertreten bzw. hat P nicht erfüllt und wurde in darauf folgenden Vergleichen nicht weiter berücksichtigt.

Die Realisierung mit dem Zähler hat den Vorteil, dass die Hash -Tabelle nicht nach jedem Proben gescannt werden muss, um die ungültigen Kandidaten aus H zu löschen und H ev tl. neu zu strukturieren. Beim Proben mit dem letzten Stream $S_{i,last}$ einer Gruppe wird mit dem Setzen des Zählers auf $last$ der entsprechende Kandidat q zusammen mit den Gruppenattributen A_G in der Ausgaberelation R_{out} abgelegt. Dadurch entfällt ein abschli eßendes Scannen der Hash -Tabelle. Der Operator fährt mit der nächsten Gruppe fort.

H	$S_{i,1}$	$S_{i,2}$	$S_{i,3}$	$S_{i,4}$
$c(q_1)$	1	2	2	2
$c(q_2)$	1	1	1	1
$c(q_4)$	1	2	3	4
$c(q_5)$	1	2	3	4

Tabelle 4.2: Zähler der Kandidaten in der Hash-Tabelle

Die Tabelle zeigt die Veränderung der Zähler der Kandidaten für das in Tabelle 4.1 vorgegebene Szenario. Die Hash -Tabelle zu IR wird mit den Kandidaten q_1, q_2, q_4 und q_5 des Streams $S_{i,1}$ aufgebaut. Alle Zähler entha lten den Wert Eins (erste Spalte). Nach Proben des Streams $S_{i,2}$ ist Kandidat q_2 ungültig und $IR = \{q_1, q_4, q_5\}$. Im nächsten Schritt disqualifiziert sich Kandidat q_1. Beim Proben des letzten Streams werden die Kandidaten q_4 und q_5 als Ergebnis ausgegeben.

Optimierungen

Die Effizienz der angegebenen Implementierung kann durch zwei Optimierungen erhöht werden. Der Algorithmus soll für eine Gruppe vorzeitig terminieren, wenn die Kandidatenmenge *IR* leer ist. Dies kann über ein Flag *IR_leer* geprüft werden. Es wird vor jedem Proben auf *wahr* gesetzt. Beim ersten Kandidaten, dessen Zähler erhöht wird, wird *IR_leer* auf *falsch* gesetzt. Ist *IR_leer* nach dem Proben des Streams immer noch auf *wahr* gesetzt, so sind alle Kandidaten ungültig. Die Gruppe trägt zum Gesamtergebnis nichts bei, da *IR* bereits leer ist. Der Algorithmus fährt mit der Bearbeitung der nächsten Gruppe fort.

Die Anwendung des Prädikats *P* kann vorgezogen werden. Die in Prädikat *P* enthaltene Bedingung wird ausschließlich auf die Tupel der Streams angewandt und nicht zum Vergleich von Tupel aus *IR* mit Tupeln der Streams verwendet. Tupel, die *P* nicht erfüllen, werden vom Algorithmus übergangen. Diese Filterung kann als Restriktion vor den Gruppierungen in (0) durchgeführt werden. In (2a) ist dann nur noch mit Hilfe von P_R zu prüfen, ob zu einem Kandidaten $q \in IR$ im Stream $S_{i.j}$ ein Tupel t seines Dividenden existiert mit $t.A_R = q.A_R$. Diese effizientere Vorgehensweise liegt nicht unmittelbar auf der Hand. Der prädikatenlogische Ausdruck einer allgemeinen Allquantifizierung sowie ihre Darstellung in einer SQL-Anfrage bzw. in der Relationenalgebra folgen dem Aufbau:

$\forall \text{Ausdruck}_1\ \exists \text{Ausdruck}_2:\ \text{Bedingung}$

Dieser Aufbau gibt die logische Auswertungsreihenfolge Allquantor –Existenzquantor–Bedingung vor. $Ausdruck_1$ enthält die Divisorrelation *D*, $Ausdruck_2$ die Dividendenrelation *R* und die Bedingung wird bei Anwendung von Equi-Stream-Join auf die Teilprädikate $P_R(r,d) \wedge P(r)$ abgebildet. Die Auswertung von Prädikat *P* kann vorgezogen werden, da es sich nur auf *R* bezieht. Es ergibt sich die Auswertungsreihenfolge *P*–Allquantor–Existenzquantor–P_R. Für die Implementierung ist sie von Vorteil, da die Kardinalität der Eingaberelation R_{in} zum frühestmöglichen Zeitpunkt reduziert wird. Dadurch sinkt der Aufwand für die Gruppierungen in Schritt (0) und für das Proben der Streams (2). Insgesamt ergibt sich ein Effizienzgewinn hinsichtlich Zeit- und Speicherbedarf.

Beispiel 4.3

Equi-Stream-Join soll die Anfrage aus Beispiel 4.1 mit vorgezogener Anwendung des Prädikats P bearbeiten. Der Aufruf des Operators erfolgt mit den Parametern:

$R_{in} = \text{Übungen}$

$A_G = \varnothing$

$A_R = \text{Mnr}$

$A_D = \text{Termin}$

$P = (\text{Abgabe} \geq 4)$

Das Prädikat P wird als Restriktion auf die Relation *Übungen* angewendet. Die Eingaberelation R_{in} enthält dann nur noch Tupel, die P erfüllen. Sie wird gemäß der Spezifikation aus Abschnitt 4.2.2 weiterverarbeitet, wobei in Schritt (2a) für den tupelweisen Vergleich von *IR* mit einem Stream gilt:

$$P_J(q, t_S) = P_R(q, t_S) = (q.Mnr = t_S.Mnr)$$

4.2.4. Erzeugen der Eingaberelation

Equi-Stream-Join ist ein unärer Operator. Er setzt voraus, dass seine Eingaberelation R_{in} alle Informationen enthält, die bei einer relationalen Division in den Relationen R und D enthalten sind. Die Tupel von R_{in} werden mit Hilfe der Attribute A_G und A_R in Dividenden $R(q)$ strukturiert. Die Divisoren $D(q)$ werden für jede Gruppe G_i aus den Belegungen in den Divisorattributen A_D gewonnen. Im Beispiel 4.1 stellt dieses Vorgehen kein Problem dar, da zu jedem Termin der Übungsgruppe Einträge vorhanden sind. Im Allgemeinen muss allerdings explizit sichergestellt werden, dass die Menge $G_i.A_D$ genau die Belegungen enthält, die das Bereichsprädikat P_B für die Kandidaten der Gruppe G_i aus der Relation D auswählt.

Enthält ein Dividend $R(q)$ ein Tupel r mit $r.A_D \notin D(q).A_D$, führt Equi-Stream-Join die Allquantifizierung der betreffenden Gruppe G nicht über den Divisor $D(q).A_D$ sondern über der Menge $D(q).A_D \cup r.A_D$ durch. Das Tupel r ist aus $R(q)$ zu entfernen, um referentielle Integrität zum Divisor $D(q)$ herzustellen. Fehlt umgekehrt vom Divisor ein Wert d_f in allen Dividenden $R(q)$ der Gruppe ($d_f \in D(q)$ und $\forall R(q): d_f.A_D \notin R(q).A_D$), wird die Allquantifizierung nicht über den kompletten Divisor sondern über $D(q) \backslash d_f$ durchgeführt. Zu der Belegung $d_f.A_D$ wird kein Stream gebildet. Um Vollständigkeit bezüglich des Divisors herzustellen, ist für die Gruppe G ein Tupel r in R_{in} hinzuzufügen mit $r.A_D = d_f.A_D$ und $r.A_G = G.A_G$ sowie NULL in den übrigen Attributen. Alternativ dazu kann die gesamte Gruppe G aus R_{in} entfernt werden, da jedem ihrer Dividenden der Wert $d_f.A_D$ fehlt und die Allquantifizierung für G keine Ausgabe erzeugt.

Die beschriebene referentielle Integrität zwischen einem Dividend $R(q)$ und dem zugehörigen Divisor $D(q)$ darf nicht mit referentieller Integrität zwischen den Relationen R und D gleichgesetzt werden. Erstere bezieht sich auf kombinierte Belegungen in den Attributen $A_G \cup A_D$ und verlangt, dass $R.A_G \cup A_D \subseteq D.A_G \cup A_D$. Sie impliziert die „übliche“ referentielle Integrität, die für Belegungen in den Attributen A_G und A_D lediglich $R.A_G \subseteq D.A_G$ und $R.A_D \subseteq D.A_D$ garantiert. Umgekehrt gilt die Implikation nicht.

Da es sich bei Stream-Join um einen unären Operator handelt, muss seine Eingaberelation R_{in} im Allgemeinen aus den Relationen R und D erzeugt werden. Die Operation muss für referentielle Integrität zwischen den Dividenden $R(q)$ und ihren Divisoren $D(q)$ sorgen. Ein rechter äußerer Gleichverbund der Relationen R und D mit $P_B \wedge P_1$ als Join-Prädikat erfüllt diese Vorgabe. Das Prädikat P_1 enthält alle Gleichheitsbedingungen des Quantorprädikats P_Q. Es ordnet einem Tupel des Dividenden das korrespondierende Tupel des Divisors anhand der gemeinsamen Attribute A_D zu (siehe Abschnitt 4.1).

$R_{in} = R \ltimes\sqsubset_{P_B \wedge P_1} D$

Das Bereichsprädikat $P_B(r, d)$ wählt zu einem Tupel r den zugehörigen Divisor $D(r.A_{out})$ aus. Aus dem Divisor wählt P_1 wiederum das mit r korrespondierende Tupel d_r aus. Existiert das Tupel d_r, wird r nach R_{in} übernommen. Existiert zu einem Tupel r kein Tupel d_r im Divisor, wird r verworfen. Das zusammengesetzte Prädikat $P_B \wedge P_1$ stellt also referentielle Integrität zwischen den Dividenden $R(q)$ und den Divisoren $D(q)$ her. Der äußere Gleichverbund sorgt dafür, dass Tupel aus D, zu denen in R kein korrespondierendes Tupel enthalten ist, nach R_{in} übernommen werden. Die in Prädikat P_B bezüglich R und D enthaltenen Attribute bilden die Gruppenattribute A_G und die in Prädikat P_1 enthaltenen Attribute die Divisorattribute A_D des Operators Equi -Stream-Join. Tu pel aus D, die der äußere Gleichverbund in R_{in} einfügt, sind daher in den Attributen A_G und A_D mit Werten belegt. In allen anderen Attributen enthalten sie den Wert NULL. Das Anfrageschema, um die Eingaberelation R_{in} zu erzeugen, lautet in SQL:

```
select  R.*
from    R  right join  D
where   PB(r,d)  and  R.AD = D.AD
```

Es ist vom Parser bzw. Optimierer zu erzeugen, wenn die Relationen R und D einer Allquantor-Anfrage aus zwei oder mehreren, physisch verschiedenen Relationen hervorgehen und die Anfrage mit Equi-Stream-Join bearbeitet werden soll.

Beispiel 4.4

Die Anfrage aus den Beispielen 2.7 bzw. 3.2 soll mit Equi -Stream-Join bearbeitet werden. Gegeben sei die Dividendenrelation *Abg3(Studium, Sname, Pnr, Note)*, die aus den Relationen *Studenten* und *Abgelegt* hervorgeht. Sie enthält die von den Studenten abgelegten Prüfungen. Die Divisorrelation *Prüf(Pnr, Studium)* stellt die Relation *Prüfungen* ohne das Prädikat *Pname* dar. Die Anfrage vom Typ 12, welche Studenten in ihrem Studiengang bereits alle Prüfungen abgelegt und bestanden haben, lautet:

```
select  distinct Sname, Studium
from    Abg3 as Q
where   for all  (select  *
                  from    Prüf as D
                  where   D.Studium = Q.Studium)
                 (exists  (select  *
                           from    Abg3 as R
                           where   R.Sname = Q.Sname and
                                   R.Pnr = D.Pnr  and  R.Note ≤ 4,0 ) )
```

Der Optimierer entscheidet, die Allquantifizierung mit Equi -Stream-Join auszuwerten. Um die Eingaberelation R_{in} zu erzeugen, zerlegt er das Quantorprädikat P_Q in P_1 und P_2.

$P_1 = (R.Pnr = D.Pnr)$

$P_2 = (R.Note \leq 4,0)$

Für den äußeren Gleichverbund ergibt sich das Join-Prädikat:

$P_B \wedge P_1$ = (R.Studium = D.Studium) $\wedge$ (R.Pnr = D.Pnr)

Die vom Optimierer generierte Anfrage zur Erzeugung von R_{in} lautet:

```
select  *
from    Abg3  right join  Prüf
where  Abg3.Studium = Prüf.Studium  and  Abg3.Pnr = Prüf.Pnr
```

Gegeben sei folgender Inhalt für die Relationen *Abg3* und *Prüf*. Die Relation *Abg3* enthält drei Informatikstudenten Alex, Mark und Tom sowie zwei Biologiestudenten Tina und Frank. In beiden Studiengängen sind je vier Prüfungen abzulegen:

Abg3	Sname	Studium	Pnr	Note
	Tina	B	9	2,3
	Mark	I	2	2,0
	Mark	I	3	3,3
	Frank	B	10	3,0
	Tom	I	4	2,0
	Alex	I	1	2,3
	Tina	B	8	3,0
	Alex	I	3	1,0
	Mark	I	1	5,0
	Tom	I	2	2,3
	Frank	B	9	1,3
	Tom	I	1	4,0
	Mark	I	4	2,3
	Frank	B	8	5,0
	Alex	I	4	1,7
	Tina	B	10	2,7
	Tom	I	2	2,3
	Mark	I	1	3,7
	Frank	B	2	3,3
	Alex	I	6	2,3

Tabelle 4.3: Abgelegte Prüfungen – R

Prüf	Studium	Pnr
	I	1
	I	2
	I	3
	I	4
	M	5
	M	6
	M	7
	B	8
	B	9
	B	10
	B	11

Tabelle 4.4: Prüfungen zu den Studiengängen – D

Die letzten beiden Einträge enthalten Prüfungen aus einem anderen Studiengang als dem des jeweiligen Studenten. Sie genügen der referentiellen Integrität zwischen beiden Relationen, nicht aber der zwischen Dividend und zugehörigem Divisor. Diese Tupel werden von der Anfrage nicht nach R_{in} übernommen, da sie das Join -Prädikat nicht erfüllen. Von den Biologiestudenten hat ke iner die Prüfung 11 abgelegt. Der äußere Gleichverbund fügt für diese Prüfung das Tupel *t* mit *t.Studium = B* und *t.Pnr = 11* in

die Relation R_{in} ein. Tabelle 4.5 zeigt die aus *Abg3* und *Prüf* erzeugte Eingaberelation für den Operator Equi-Stream-Join.

Der Optimierer bestimmt nun die übrigen Eingabeparameter des Operators. Aus A_{out} = *{Sname, Studium}* und P_B = *(D.Studium = R.Studium)* folgt:

$A_G = \{Studium\}$ und $A_R = \{Sname\}$

Das Prädikat P_1 wird auf die Menge der Divisorattribute und das Prä dikat P_2 auf das Prädikat *P* abgebildet:

$A_D = \{Pnr\}$ und $P = (Note \leq 4{,}0)$

Die Allquantifizierung ist eine relationale Division, da $A_D \neq \emptyset$. Der Operator gruppiert seine Eingabe also nach *Studium* in zwei Gruppen und strukturiert die Gruppen nach *Pnr* in je vier Streams. Die so aufbereitete Relation R_{in} ist in Tabelle 4.6 dargestellt.

R_{in}	Sname	Studium	Pnr	Note
	Tina	B	9	2,3
	Mark	I	2	2,0
	Mark	I	3	3,3
	Frank	B	10	3,0
	Tom	I	4	2,0
	Alex	I	1	2,3
	Tina	B	8	3,0
	Alex	I	3	1,0
	Mark	I	1	5,0
	Tom	I	2	2,3
	Frank	B	9	1,3
	Tom	I	1	4,0
	Mark	I	4	2,3
	Frank	B	8	5,0
	Alex	I	4	1,7
	Tina	B	10	2,7
	Tom	I	2	2,3
	Mark	I	1	3,7
	NULL	B	11	NULL

Tabelle 4.5: Eingaberelation R_{in}

R_{in}	Sname	Studium	Pnr	Note
$S_{1,1}$	Tina	B	9	2,3
	Frank	B	9	1,3
$S_{1,2}$	Tina	B	8	3,0
	Frank	B	8	5,0
$S_{1,3}$	NULL	B	11	NULL
$S_{1,4}$	Frank	B	10	3,0
	Tina	B	10	2,7
$S_{2,1}$	Alex	I	1	2,3
	Mark	I	1	5,0
	Mark	I	1	3,7
	Tom	I	1	4,0
$S_{2,2}$	Mark	I	4	2,3
	Tom	I	4	2,0
	Alex	I	4	1,7
$S_{2,3}$	Tom	I	2	2,3
	Mark	I	2	2,0
	Tom	I	2	2,3
$S_{2,4}$	Alex	I	3	1,0
	Mark	I	3	3,3

Tabelle 4.6: Aufbereitete Relation R_{in}

Die Gruppe G_1 der Biologiestudenten enthält zwei Dividenden *R(Tina)* und *R(Frank)* sowie den Divisor *{8, 9, 10, 11}* . Die Menge *IR* wird mit den Kandidaten *Tina* und *Frank* initialisiert. Beim Vergleich mit Stream $S_{1,2}$ disqualifiziert sich Kandidat *Frank*. Das einzige Tupel seines Dividenden erfüllt das Prädikat *P* nicht. Nach dem Vergleich mit Stream $S_{1,3}$ ist *IR* leer und der Algorithmus bricht die Verarbeitung von Gruppe G_1

mit leerer Ergebnismenge ab. Wäre das Tupel der Prüfung 11 nicht eingefügt worden, hätte Equi-Stream-Join fälschlicherweise die Studentin Tina als Ergebnis ausgegeben.

Die Gruppe G_2 der Informatikstudenten enthält drei Dividenden *R(Alex), R(Mark)* und *R(Tom)* sowie den Divisor *{1, 2, 3, 4}*. Beim Initialisieren der Menge *IR* wird das erste Tupel des Studenten Mark in $S_{2,1}$ übergangen, da es das Prädikat *P* nicht erfüllt. Nach der Initialisierung ist *IR = {Alex, Mark, Tom}*. Beim Vergleich mit Stream $S_{2,3}$ disqualifiziert sich der Kandidat *Alex*. Der Kandidat *Tom* besitzt zwei identische Einträge. Der Zähler *c(tom)* wird durch das erste Tupel des Streams auf den Wert *3* gesetzt. Die Bedingung *c(tom) = j-1 = 2* für das Ändern des Zählers ist beim Duplikat nicht erfüllt; es hat keine Auswirkung. Nach Verarbeitung des letzten Streams enthält *IR* den Kandidaten *Mark*. Er wird mit der Belegung *I* der Gruppe im Attribut *Studium* versehen und als einziges Ergebnis der Allquantifizierung in der Relation R_{out} abgelegt.

R_{out}	Sname	Studium
	Mark	I

Tabelle 4.7: Ausgaberelation R_{out}

Bei vorgezogener Auswertung des Prädikats *P* werden die beiden Einträge von nicht bestandenen Prüfungen vor den Gruppierungen (Schritt 0) entfernt. Das korrekte Restriktionsprädikat lautet:

$$P = (R_{in}.\text{Note} \leq 4{,}0 \text{ or } R_{in}.\text{Note is NULL})$$

Fehlt das Zulassen von NULL-Werten, werden diejenigen Tupel, die der äußere Gleichverbund zur Vervollständigung der Divisoren einfügt, von der Restriktion wieder entfernt.

In Schritt 2 müssen die Streams der Gruppe nur noch mit *IR* geprobt werden, da das Prädikat *P* bereits angewendet worden ist. Durch das Proben wird das Prädikat P_R auf *IR* und die Streams angewendet, womit die Anwendung des Joinprädikats $P_J = P_R \wedge P$ abgeschlossen wird. Das Prädikat P_R ist im Beispiel:

$$P_R = (IR.\text{Sname} = S_{i,j}.\text{Sname})$$

4.3. Eigenschaften des Algorithmus

Equi-Stream-Join ist in der Lage, alle zwölf Typen von Allquantor-Anfragen zu bearbeiten. Seine Einsetzbarkeit unterliegt in dieser Hinsicht keinen Einschränkungen.

Auf Anfragen, deren Relationen *R* und *D* aus derselben physischen Relation hervorgehen, kann der Operator unmittelbar angewendet werden (siehe Beispiel 4.1). Gehen die Relationen *R* und *D* aus zwei oder mehreren physischen Relationen hervor, muss die Eingaberelation R_{in} des Operators explizit aus diesen Relationen erzeugt werden. Bei relationalen Divisionen geschieht dies durch einen (rechten) äußeren Gleichverbund.

Bei einfachen Anfragetypen sind die Relationen R und D nicht mit den Rollen Dividenden- und Divisorrelation belegt, weshalb die Eingaberelation durch einen gewöhnlichen Join erzeugt wird.

Welche Auswirkungen haben Duplikate in der Eingaberelation R_{in}? Sie führen dazu, dass ein Dividend in einem Stream mit mehreren Tupeln vertreten ist. Wie bereits im Beispiel beschrieben, haben Duplikate auf das Ergebnis keine Auswirkungen. Der Algorithmus setzt den Zähler eines Kandidaten höchstens einmal bei jedem Vergleich mit einem Stream. Duplikate werden also übergangen und müssen nicht explizit entfernt werden.

Beim Erzeugen der Eingaberelation R_{in} durch einen rechten äußeren Gleichverbund der Relationen R und D werden diejenigen Tupel der Relation R nach R_{in} übernommen, die das Join-Prädikat $P_B \wedge P_1$ erfüllen. Dies gilt auch für in R enthaltene Duplikate. Duplikate der Divisorrelation D werden nicht übernommen. Wie im nächsten Abschnitt erläutert, wird der äußere Gleichverbund durch einen hash-basierten Algorithmus realisiert. Die Relation D ist die Build-Relation. Beim Aufbau ihrer Hash-Tabelle werden in D vorhandene Duplikate automatisch entfernt.

Bei relationaler Division blockiert der Operator während der Gruppierung (Schritt 0) und der Bearbeitung einer Gruppe (Schritte 1 bis 3). Enthält R_{in} nur eine Gruppe, ist Equi-Stream-Join ein blockierender Operator. Bei mehreren Gruppen ist die Blockierung phasenweise. Die Ausgabe des Operators geht mit zunehmender Anzahl an Gruppen in einen kontinuierlichen Datenstrom über. Bei einer Allquantifizierung einfachen Typs blockiert der Operator lediglich während der Gruppierung. Da die Eingaberelation kandidatenweise verarbeitet wird, liefert er seine Ausgabe als kontinuierlichen Datenstrom. Im Operatorbaum kann eine Pipeline zum Nachfolger aufgebaut werden, wenn die Anfrage eine Allquantifizierung einfachen Typs oder mehrere, gruppenweise ausführbare relationale Divisionen enthält.

4.3.1. Aufwandsabschätzungen

Die Aufwandsabschätzungen sollen es ermöglichen, die Effizienz von Equi-Stream-Join zu bewerten. Sie sind die Grundlage für kostenbasierte Optimierer bei der Bewertung von Ausführungsplänen. Der Aufwand von Equi-Stream-Join wird davon beeinflusst, ob der Operator unmittelbar angewandt werden kann oder ob zuvor seine Eingaberelation explizit erzeugt werden muss.

Es werden die übergeordneten Szenarien One-Pass und Two-Pass unter den Voraussetzungen und Annahmen betrachtet, wie sie den Aufwandsabschätzungen von Hash-Division in Kapitel 3 zugrunde liegen. Beim One-Pass Szenario können die Eingaberelationen R und D bzw. R_{in} sowie die Ergebnisrelation R_{out} im Datenbankpuffer gehalten werden. Beim Two-Pass Szenario kann die kleinere Divisorrelation D im Datenbankpuffer gehalten werden; die Relationen R bzw. R_{in} müssen jedoch auf Platte ausgeschrieben werden. Der Aufwand für das Lesen von R und D und das Schreiben von R_{out} wird in beiden Szenarien nicht berücksichtigt, da er unabhängig vom gewählten Operator zu erbringen ist. Der Aufwand zum Aufbau einer Hash-Tabelle sowie das Proben beträgt $k_H\, n$ für eine Relation der Kardinalität n.

One-Pass Szenario

Relationale Division und unmittelbare Anwendbarkeit

Sei $n_{in} = |R_{in}|$ die Kardinalität der Eingaberelation und $n_S = |R_{in}.A_G \cup A_D|$ die Summe der Kardinalitäten der Divisoren aller Gruppen. Es gilt $n_S \leq n_{in}$ und es gibt insgesamt genau n_S Streams. Bei Gleichverteilung beträgt die Kardinalität eines Streams n_{in}/n_S. Sie entspricht der Kardinalität von IR. Die Eingaberelation wird in n_G Gruppen G_i partitioniert. Der Aufwand von Equi-Stream-Join beträgt bei relationalen Divisionen und unmittelbarer Anwendung:

(1) Gruppieren der Eingaberelation R_{in}:	$k_H\, n_{in}$
(2) Aufbau der n_G Hash-Tabellen von IR:	$(k_H\, n_{in} / n_S)\, n_G$
(3) Proben der übrigen $(n_S - n_G)$ Streams mit IR: $(k_H\, n_{in} / n_S)\, (n_S - n_G) =$	$k_H\, n_{in} - (k_H\, n_{in} / n_S)\, n_G$
Gesamtaufwand:	$2\, k_H\, n_{in}$
mit $k_H = 1{,}2$:	$2{,}4\, n_{in}$

Relationale Division mit Erzeugen der Eingaberelation

Seien $n_R = |R|$ und $n_D = |D|$ die Kardinalitäten der Relationen R und D. Bezüglich der Kardinalitäten gilt $n_D < n_R$ und $n_{in} \approx n_R$. Die Eingaberelation R_{in} kann aus den Relationen R und D mit relativ geringem Aufwand erzeugt werden, wenn sie mit der Gruppierung von R_{in} nach $A_G \cup A_D$ kombiniert wird. Dies ist möglich, weil der äußere Gleichverbund in seinem Join-Prädikat $P_B \wedge P_1$ genau die Attribute $A_G \cup A_D$ verwendet und weil zu jeder Belegung der Menge $D.A_G \cup A_D$ durch die Gruppierung genau ein Stream erzeugt wird. Der äußere Gleichverbund wird durch einen hash-basierten Algorithmus realisiert. Die Relation D ist die Build-Relation. Beim Aufbau der zugehörigen Hash-Tabelle wird jeder Eintrag d mit einem Pointer auf das Bucket des entsprechenden Streams versehen. In diesem Bucket werden diejenigen Tupel aus R abgelegt, die mit d in den Attributen $A_G \cup A_D$ übereinstimmen. Beim Proben der Relation R wird jedem Tupel $r \in R$, das das Join-Prädikat erfüllt, über die Hash-Funktion das korrespondierende Tupel d mit $d.A_G \cup A_D = r.A_G \cup A_D$ zugewiesen. Es erfolgt die vorgezogene Anwendung des Prädikats P von Equi-Stream-Join auf das Tupel r. Ist P erfüllt, wird r in dem Bucket abgelegt, auf das der Pointer von Tupel d zeigt. Nach Abschluss des äußeren Gleichverbunds ist R_{in} bereits gruppiert und das Prädikat P angewendet.

Der gesamten Aufwand für dieses Vorgehen wird von den Kardinalitäten n_R, n_D und n_{in} bestimmt. Das Erzeugen der Eingaberelation findet in Kombination mit der Gruppierung nach $A_G \cup A_D$ und der vorgezogenen Anwendung des Prädikats P statt. Die daraus hervorgehende Relation R_{in} hat durch die Restriktion des Prädikats P eine bereits verminderte Kardinalität n_{in}. Der Aufwand des Operators beträgt für dieses Szenario:

(1) Aufbau der Hash-Tabelle der Relation D:	$k_H\ n_D$
(2) Proben der Relation R und Gruppieren von R_{in}:	$k_H\ n_R$
(3) Vervollständigen von R_{in} bezüglich D (äußerer Gleichverbund):	$1{,}0\ n_D$
(3) Anwendung des Algorithmus ohne Gruppieren:	$k_H\ n_{in}$
Gesamtaufwand mit $n_{in} = n_R$:	$2\ k_H\ n_R + (1 + k_H)\ n_D$
mit $k_H = 1{,}2$:	$2{,}4\ n_R + 2{,}2\ n_D$

Durch die Kombination des äußeren Gleichverbunds mit der Gruppierung nach $A_G \cup A_D$ spart man den Aufwand der Gruppierung komplett ein. Das Prädikat P reduziert die Größe der Eingaberelation bereits bei ihrer Erzeugung. Zum Abschluss des äußeren Gleichverbunds wird die Hash-Tabelle der Divisorrelation D einmal gelesen. Für jedes Tupel wird geprüft, ob es mit mindestens einem Tupel aus R zusammengeführt wurde. Tupel, auf die das nicht zutrifft, werden in die Relation R_{in} eingefügt. Die anschließende Anwendung von Equi-Stream-Join ohne Gruppierung verursacht nur noch einen Aufwand von $k_H\ n_{in}$.

Für die Kardinalitäten der Relationen R_{in} und R gilt grundsätzlich $n_{in} < n_R$. Sowohl das Prädikat $P_B \wedge P_1$ des äußeren Gleichverbunds als auch das Prädikat P selektieren Tupel aus R. Ihre Selektivität liegt in dem Intervall $[0; 1]$ und entspricht dem Quotienten n_{in}/n_R, sofern der äußere Gleichverbund keine Tupel aus D in R_{in} einfügt. Lediglich wenn die Selektivität beider Prädikate genau 1 beträgt und der äußere Gleichverbund ein oder mehrere Tupel in R_{in} einfügt, wird R_{in} größer als R. Die Abschätzung $n_{in} = n_R$ für die Kardinalität von R_{in} ist bis auf diesen Ausnahmefall eine gültige obere Schranke. Sie ermöglicht es, den Aufwand von Equi-Stream-Join mit dem Aufwand von Hash-Division direkt vergleichen zu können.

Einfache Anfragetypen

Bei Allquantifizierungen einfachen Typs und unmittelbarer Anwendbarkeit von Equi-Stream-Join beträgt der Aufwand:

(1) Gruppieren der Eingaberelation R_{in} nach A_{out} in Partitionen R(q):	$k_H\ n_{in}$
(2) Partitionsweises Prüfen des Prädikats P:	$1{,}0\ n_{in}$
Gesamtaufwand:	$(1 + k_H)\ n_{in}$
mit $k_H = 1{,}2$:	$2{,}2\ n_{in}$

Er liegt geringfügig unter dem Aufwand des Operators bei relationalen Divisionen. Dies legt die Vermutung nahe, dass Equi-Stream-Join relationale Divisionen sehr effizient ausführt und eine noch effizientere Ausführung kaum mehr möglich ist.

Muss die Eingaberelation aus den Relationen R und D durch einen Join erzeugt werden, ist der Aufwand dieser Operation hinzuzurechnen. Für die Kardinalität der Eingaberelation gilt $0 \le |R_{in}| \le |R|\ |D|$. Je nach Selektivität des Joinprädikats ist befindet sie sich näher an der oberen oder unteren Intervallgrenze. Der Aufwand des Joins kann durch eine Konstante k_J mal seiner Ausgaberelation abgeschätzt werden, wodurch sich der Gesamtaufwand auf $(1 + k_J + k_H)\ |R_{in}|$ erhöht.

Two-Pass Szenario

Die Anzahl der E/A-Operationen soll für den allgemeinen Fall einer relationalen Division bestimmt werden, wenn die Eingaberelation R_{in} aus den Relationen R und D erzeugt werden muss. $B(R)$ gibt die Anzahl der Blöcke an, in denen die Relation R auf Platte abgelegt ist. M gibt den zur Verfügung stehenden Datenbankpuffer in Seiten an, wobei die Größe einer Seite mit der eines Blockes übereinstimmt. Equi-Stream-Join kann als Two-Pass Algorithmus realisiert werden, wenn D im Hauptspeicher gehalten werden kann und für die Relation R_{in} insgesamt M freie Seiten mit $M < B(R_{in}) \leq M^2$ zur Verfügung stehen. Die Größe der Relation R ist nicht von Bedeutung.

Die Divisorrelation D wird eingelesen und als Hash-Tabelle im Hauptspeicher gehalten. Es fallen $B(D)$ Leseoperationen an. Zur Erzeugung von R_{in} wird die Dividendenrelation R mit der Hash-Tabelle D geprobt. Das Prädikat P wird angewendet. Die ausgewählten Tupel werden von den Pointern in D anhand der Attribute $A_G \cup A_D$ partitioniert und in den verfügbaren M Buckets abgelegt. Diese Partitionierung ist die erste Phase der Gruppierung nach $A_G \cup A_D$. Ist die Seite eines Buckets voll, wird sie ausgeschrieben. Der äußere Gleichverbund verursacht in Verbindung mit der Partitionierung $B(R)$ Lese- und $B(R_{in})$ Schreiboperationen. Die Größe der auf Platte befindlichen Buckets beträgt durchschnittlich $B(R_{in})/M$ Blöcke. Zur weiteren Verarbeitung muss jedes Bucket in den Hauptspeicher passen, weshalb für alle Buckets $B(R_{in})/M \leq M$ gelten muss. Daraus leitet sich die Voraussetzung $B(R_{in}) \leq M^2$ für die Anwendbarkeit des Algorithmus ab.

Das erste Bucket der so partitionierten Relation R_{in} wird in den Hauptspeicher gelesen. Die Gruppierung nach $A_G \cup A_D$ wird innerhalb des Buckets fertiggestellt (zweite Phase). Der Algorithmus kann mit der Verarbeitung der ersten Gruppe G beginnen. Aus dem ersten Stream wird die Hash-Tabelle von IR erzeugt. Die Relation wird nicht mehr benötigt, so dass die belegten Seiten freigegeben und IR dort abgelegt werden kann. Die folgenden Streams werden mit IR geprobt. Wenn die Gruppe G über das Bucket hinausgeht, wird das nächste Bucket eingelesen, gruppiert und weiterverarbeitet. Nach der Verarbeitung der Gruppe werden die in IR enthaltene Kandidaten ausgeschrieben. Die nächste Gruppe kann verarbeitet werden. Es fallen $B(R_{in})$ Lese- und $B(R_{out})$ Schreiboperationen an.

Vom Gesamtaufwand des Two-Pass Algorithmus für Equi-Stream-Join werden die Operationen für das Einlesen der Relationen R und D sowie das Ausschreiben der Relation R_{out} abgezogen, um den algorithmus-spezifischen Anteil an E/A-Operationen zu ermitteln:

E/A Gesamt:	$B(R) + 2\,B(R_{in}) + B(D) + B(R_{out})$
E/A Equi-Stream-Join	$2\,B(R_{in})$

5. Optimale Bearbeitung von Allquantifizierung in Anfragen

In den beiden vorangehenden Kapiteln wurden zwei sehr verschiedene Ansätze zur Bearbeitung von Allquantifizierung in Anfragen vorgestellt und beschriebe n. Die Frage zu stellen, welcher Ansatz der bessere ist, trifft nicht den Kern einer effizienten Anfragebearbeitung. Das zentrale Interesse besteht nicht darin, einen optimalen Ansatz zu haben, sondern jede Allquantor -Anfrage optimal bearbeiten zu können. Im ersten Abschnitt dieses Kapitels werden grundlegende Voraussetzungen benannt, die eine optimale Bearbeitung von Allquantifizierung überhaupt erst ermöglichen. Im zweiten Abschnitt wird zu jeder Anfrage gemäß ihren Randbedingungen eine optimale Bearbeitung angegeben.

5.1. Erweiterung der Anfragesprache

Wie bereits in Kapitel 2 erwähnt, steht der Allquantor in SQL bislang nicht zur Verfügung. Die Aufnahme des Schlüsselworts FOR ALL in den Standard von 1999 (siehe [11]) ist letztlich nicht erfolgt. Das in SQL vorhandene Schlüsselwort ALL entspricht dem Allquantor im Allgemeinen nicht. Es prüft relationen - und nicht tupelweise. Wie in Abschnitt 2.1.3 dargestellt, können deshalb mit ALL nicht einmal alle einfachen Anfragetypen ausgedrückt werden. Relationale Divisionen, also die Typen von Allquantifizierung, die einen Existenzquantor enthalten, können in SQL nicht direkt ausgedrückt werden. Der Allquantor muss in solchen Anfragen durch den negierten Existenzquantor, durch Aggregieren und Zählen oder durch Mengendifferenzen umschrieben werden. Die Umschreibungen, zu denen der Anfragesteller gezwungen wird, sind keine intuitiven sondern durch logische Umformungen konstruierte Formuli erungen. Sowohl das Erstellen als auch das Lesen und Interpretieren solcher Anfragen ist dadurch unnötig kompliziert, Fehler sind vorprogrammiert.

Für die Parser bzw. Optimierer der Datenbanksysteme ist es ebenso aufwendig, Allquantifizierung in Anfragen zu erkennen. Er muss in Anfragen jeden Ausdruck, der einen negierten Existenzquantor, eine Aggregation mit Zählen oder Mengendifferenzen enthält, daraufhin überprüfen, ob es sich nicht um eine umschriebene Allquantifizierung handeln könnte. Besonders bei Allquantifizierungen, die durch Aggregation und Zählen umschrieben sind, ist dies schwierig. Wenn er eine Allquantifizierung entdeckt hat, muss er die Umschreibung in einen Ausdruck mit Allquantor umwandeln. Erst auf diesen Ausdruck kann er effiziente, spezialisierte Algorithmen wie Hash-Division und Equi-Stream-Join anwenden. Der Optimierer macht also die Umformulierung, zu der der Anfragesteller mangels des Schlüsselworts FOR ALL gezwungen wird, unter hohem Aufwand wieder rückgängig.

Das Fehlen des Allquantors in SQL verursacht beim Benutzer und beim Datenbanksystem unnötigen und unerwünschten Aufwand. Die Formulierung einer Allquantor-Anfrage ist komplizierter als der Inhalt, den sie beschreibt. Die in vielerlei Hinsicht gelungene und mächtige Anfragesprache SQL bietet, was Allquantifizierung angeht, nur mangelhafte Ausdrucksmöglichkeiten. Dieses Defizit wirkt bis in die Anfragebearbeitung hinein und stellt eine erhebliche Hürde für eine effiziente Ausführung von Allquantor-Anfragen dar.

Es war eine Fehlentscheidung, das Schlüsselwort FOR ALL nicht in den SQL-Standard von 1999 aufzunehmen. Es ist eine Voraussetzung für die optimale Bearbeitung von Allquantifizierung in Anfragen, dass die Anfragesprache selbst über einen Allquantor verfügt und direkte, intuitive Formulierungen ermöglicht. So lange diese Voraussetzung nicht gegeben ist, werden Datenbanksysteme nicht vor der Aufgabe stehen, Anfragen mit Allquantoren zu verarbeiten. Die Frage nach der optimalen Ausführung solcher Anfragen wird man entsprechend selten stellen.

5.2. Anfrageszenarien

Ein Anfrageszenario wird durch die Randbedingungen der Anfrage bestimmt und vom Datenbanksystem durch einen Operatorbaum mit logischen Operatoren beschrieben. Wenn man viele Randbedingungen berücksichtigt, werden die Betrachtungen komplexer aber auch differenzierter. Die Zuordnung einer bestimmten algorithmischen Vorgehensweise zu einem Szenario kann spezifischer vorgenommen werden. Sie wird in Datenbanksystemen vom Optimierer dadurch vollzogen, dass er einen logischen Operator auf einen physischen abbildet. Man betrachtet zu viele Randbedingungen, wenn die Unterschiede zwischen einzelnen Szenarien nur noch von untergeordneter Bedeutung für die algorithmische Vorgehensweise sind. Die Differenzierung verliert dadurch ihre Berechtigung und sollte zu Gunsten der Übersichtlichkeit reduziert werden. Hier werden als Randbedingungen der Anfragetyp, vorhandene Gruppierungen, die Größe der Relationen R und R_{in} sowie die Anzahl der physischen Eingaberelationen berücksichtigt.

Als physische Operatoren zur Bearbeitung einer Allquantor-Anfrage werden Hash-Division und Equi-Stream-Join betrachtet. [8] dokumentiert ausführlich die Überlegenheit von Hash-Division gegenüber anderen, gängigen Ansätzen. Auf eine nochmalige Betrachtung dieser Ansätze wird daher verzichtet. Um den Aufwand der Operatoren Hash-Division und Equi-Stream-Join in den Gegenüberstellungen miteinander zu vergleichen, werden der Übersichtlichkeit halber die Kostenformeln verwendet, in denen die Konstante k_H auf *1,2* gesetzt ist.

Einfache Anfragetypen

Bei den einfachen Anfragetypen 1, 2, 4, 5, 7, 8, 10 und 11 steht nur der Operator Equi-Stream-Join zur Auswahl. Er bearbeitet diese Anfragen mit einem Aufwand von *2,2* $|R_{in}|$. Liegt die Eingaberelation nach den Attributen A_{out} gruppiert oder sortiert vor, verringert sich der Aufwand auf *1,0* $|R_{in}|$ für das einmalige Lesen der Eingabe.

Relationale Divisionen

Bei Anfragen vom Typ 3, 6 und 9 ist der Unterschied zwischen beiden Algorithmen gering, wenn die Relationen R und D aus verschiedenen physischen Relationen hervorgehen. Der Aufwand beträgt bei One-Pass Szenarien jeweils:

Hash-Division: $2{,}4\ n_R + 2{,}2\ n_C + 1{,}2\ n_D$

Equi-Stream-Join: $1{,}2\ n_R + 1{,}2\ n_{in} + 2{,}2\ n_D \leq 2{,}4\ n_R + 2{,}2\ n_D$

Für die Kardinalität der Dividendenrelation R gilt bezüglich Quotient und Divisor $|Q|\ |D| \leq |R|$. Diese untere Schranke zeigt, dass R nur dann kleiner als D sein kann, wenn Q leer ist. Für zehn Ergebnistupel in Q übersteigt die Kardinalität von R die von D bereits um mindestens eine Größenordnung. Als Mittelwert kann in realen Anwendungen von mehrere Größenordnungen ausgegangen und $|R|$ als die dominierende Größe bei der Bestimmung des Aufwands betrachtet werden. Wenn das Teilprädikat P_2 des Quantorprädikats P_Q nicht selektiv und $|R_{in}| \approx |R|$ ist, unterscheiden sich beide Ansätze im Aufwand nur geringfügig. Hash-Division hat allerdings den Nachteil, dass er zwei Hash-Tabellen benötigt und sein Aufwand von der Kardinalität von Q_C abhängt. Wenn die Zwischenrelation Q_C sehr groß wird oder wenn das Teilprädikat P_2 selektiv und $|R_{in}| << |R|$ ist, liegt der Aufwand von Hash-Division erheblich über dem Aufwand von Equi-Stream-Join. Nur in dem Fall, dass P_2 nicht vorhanden bzw. seine Selektivität annähernd Eins beträgt und für die Größe der Kandidatenrelation *2,2* $|Q_C| < |D|$ gilt, ist Hash-Division effizienter als Equi-Stream-Join.

Bei Anfragen, deren Relationen R und D aus derselben physischen Relation hervorgehen (siehe Beispiel 4.1), ist Equi-Stream-Join ohne vorangehenden äußeren Gleichverbund anwendbar. Es gilt $|R_{in}| = |R|$ und er bearbeitet solche Anfragen mit einem Aufwand von *2,4* $|R_{in}|$ effizienter als Hash-Division.

Bei großen Relationen R, die nicht in den verfügbaren Datenbankpuffer passen, müssen die Two-Pass Varianten der beiden Ansätze angewendet werden. Der algorithmusspezifische Aufwand an E/A-Operationen beträgt jeweils:

E/A Hash-Division: $2\ B(R)$

E/A Equi-Stream-Join $2\ B(R_{in})$

Es hängt von der Selektivität des Teilprädikats P_2 ab, ob $|R_{in}| \approx |R|$ oder ob $|R_{in}| << |R|$ ist. Equi-Stream-Join bearbeitet die Anfragen bei großer Relation R also mit gleichem oder niedrigerem Aufwand als Hash-Division. Ein weiterer, nicht unbedeutender Vorteil von Equi-Stream-Join ist, dass er durch die vorgezogene Anwendung des Prädikats P_2 aus der Relation R vor dem Füllen der Buckets die Relation R_{in} erzeugt. Für seine Einsetzbarkeit ist allein die Größe von R_{in} in Blöcken ausschlaggebend, so dass er Konstellationen mit $B(R_{in}) \leq M^2 < B(R)$ noch mit seiner Two-Pass Variante bearbeiten kann, wohingegen Hash-Division bereits einen Multi-Pass Algorithmus einsetzten müsste. Je höher die Selektivität des Prädikats ist, umso stärker kommt dieser Vorteil zum Tragen.

Interessanter als der allgemeine Algorithmus Hash-Division ist seine Variante Incremental Hash-Division. Sie kann angewendet werden, wenn die Dividendenrelation R nach den Attributen A_{out} und damit nach Dividenden $R(q)$ gruppiert oder sortiert vorliegt. Ihr Aufwand liegt mit $1{,}2\ n_R + 1{,}2\ n_D$ in jedem Fall deutlich unter dem Aufwand von Equi-Stream-Join. Ein weiterer Effizienzvorteil besteht darin, dass Incremental Hash-Division seine Ausgabe kontinuierlich produziert und Pipelining zum nachfolgenden Operator ermöglicht. Die Einsetzbarkeit des Operators hängt nicht von der Kardinalität der Relation R ab. Solange die einzelnen Dividend $R(q)$ in den Haupt-speicher passen, kann der One-Pass Algorithmus beliebig große Relationen R verarbeiten.

Die allgemeinen Allquantor-Anfragen vom Typ 12 können nur von Equi-Stream-Join bearbeitet werden. Der Aufwand für diese Anfragen liegt nicht höher als bei den übrigen relationalen Divisionen. Sie werden ebenso effizient bearbeitet.

Selektivität der Allquantifizierung

Die Selektivität s_A einer Allquantor-Anfrage kann definiert werden als:

$$s_A = |D|\,|Q| / |R|$$

Der Inhalt der Daten in Dividenden- und Divisorrelation beeinflusst die Selektivität, da von ihm z. B. abhängt, ob Q leer und $s_A = 0$ ist oder ob $|D|\,|Q| = |R|$ gilt und $s_A = 1$ ist. Der Inhalt der Daten und die Selektivität spielen für die Auswahl des physischen Operators keine Rolle, da nicht sie sondern die Kardinalitäten der Eingaberelationen den Aufwand beider Operatoren bestimmen.

Duplikate in den Relationen R und D bereiten beiden Operatoren keine Schwierigkeiten; die Ergebnisrelation R_{out} bzw. Q enthält keine Duplikate. Referentielle Integrität zwischen Dividenden und Divisor wird bei Hash-Division implizit und bei Equi-Stream-Join durch den äußeren Gleichverbund hergestellt. Diese beiden Randbedingungen liefern daher für die Auswahl des physischen Operators ebenfalls keine Anhaltspunkte.

Fazit

Die Einsetzbarkeit von Equi-Stream-Join unterliegt keinerlei Einschränkungen und umfasst bei Two-Pass Szenarien einen größeren Bereich hinsichtlich der Relation R. Hash-Division kann dagegen nur bei den Anfragetypen 3, 6 und 9 eingesetzt werden, wobei er einer wenn auch geringfügigen Einschränkung (siehe Abschnitt 3.1) unterliegt. Er kann größere Relationen R verarbeiten als Equi-Stream-Join, wenn diese nach den Attributen A_{out} gruppiert vorliegen.

Ist die Einsetzbarkeit beider physischer Operatoren gegeben, bearbeitet Equi -Stream-Join die Anfragen meist effizienter. Dies liegt vor allem daran, dass Hash -Division mit zwei Hash -Tabellen arbeitet, wohingegen Equi -Stream-Join mit einer Hash-Tabelle auskommt. Der Unterschied zwischen beiden Operatoren ist wesentlich, wenn das Teilprädikat P_2 des Quantorprädikats P_Q selektiv ist oder sich die Anfrage auf eine einzige physische Relation bezieht. Wenn die Relation R nach A_{out} gruppiert vorli egt, bearbeitet der Operator Incremental Hash-Division die Anfrage am effizientesten.

Damit ein relationales Datenbanksystem Allquantifizierung in Anfragen optimal bearbeiten kann, sind zweierlei Voraussetzungen zu schaffen. Die Schnittstelle zum Benutzer, also die Anfragesprache, muss ein Schlüsselwort für den Allquantor enthalten. Nur dann kann der Benutzer solche Anfragen direkt formulieren und der Parser bzw. Optimierer den Allquantor eindeutig erkennen. Intern muss das Datenbank -system über die Operatoren Equi-Stream-Join und Incremental Hash-Division verfügen. Die effiziente Bearbeitung der Anfrage erfolgt in den meisten Fällen mit Equi -Stream-Join; wenn die Randbedingungen es zulassen, wird Incremental Hash -Division eingesetzt. Der Aufwand für die Au sführung solcher Anfragen ist linear bezüglich der Eingabekardinalitäten und, wie bereits in [8] gezeigt, nicht höher als der Aufwand von entsprechenden Semi-Join Algorithmen zur Existenzquantifizierung.

6. Der Ansatz Sequi-Stream-Join

Der Operator Stream -Join kann unter gewissen Voraussetzungen zur Analyse von Sequenzen eingesetzt werden. In diesem Kapitel wird die Variante Sequi -Stream-Join vorgestellt. Im Abschnitt 6.2 wird der zugehörige Algorithmus spezifiziert und von dem des Equi -Stream-Join abgegrenzt. Abschnitt beschreibt die Einsatzmöglichkeiten des Operators anhand von Beispielen und geht auf die in [3] beschriebene Anwendung des Operators zur Mustererkennung in Genomdatenbanken ein.

Definition von Sequenz

Bevor man den Operator sequence Stream -Join beschreibt, ist zuerst der Begriff Sequenz zu klären. In der Literatur finden sich verschiedene Sequenzbegriffe im Zusammenhang mit Datenbanksystemen. Da ausschließl ich relationale Datenbank - systeme Gegenstand der Diplomarbeit sind, setzt der hier verwendete Begriff von Sequenz auf dem der Relation auf.

Definition Sequenz:
Eine Sequenz $\check{S}$ ist eine nach Attributen $A_{Ord} \neq \emptyset$ sortierte Menge R von Tupeln, die in den Attributen A_{ID} übereinstimmen. Zu jeder Belegung von $A_{Ord} \cup A_{ID}$ darf nur ein Tupel existieren.

Die Definition enthält zwei Grundvoraussetzungen für eine Sequenz. Ihr muss eine Ordnung zugrunde liegen und bezüglich dieser Ordnung darf es innerhalb einer Sequenz keine Duplikate geben, d.h. sie muss eindeutig sein. Die Ordnung ist durch die Sortierattribute A_{Ord} gegeben; A_{Ord} darf daher nicht leer sein. Zu jedem Attribut aus A_{Ord} kann angegeben werden, ob es auf - oder absteigend sortiert werden soll. Die Attribute A_{ID} identifizieren eine Sequenz $\check{S}$ innerhalb einer Menge von Sequenzen, z. B. die Kurse einer Aktie innerhalb einer Menge von Kurswerten mehrerer Aktien. Alle in R enthaltenen Sequenzen sind *eindeutig*, wenn $A_{Ord} \cup A_{ID}$ einen Schlüssel auf R definieren. Dann ist s ichergestellt, dass z. B. zu jeder Aktie A_{ID} und zu jedem Börsentag A_{Ord} nur ein Schlusskurs in R eingetragen ist.

Diese Definition von Sequenz verlangt lediglich die Grundvoraussetzungen, die implizit mit dem Begriff Sequenz verbunden sind, und schränkt diesen nicht weiter ein. Sie ist Voraussetzung für eine eindeutige Semantik von Sequi -Stream-Join. Abschnitt 6.4 stellt diesen Zusammenhang ausführlich dar und begründet die Notwendigkeit der geforderten Eindeutigkeit von Sequenzen.

6.1. Logischer Operator

Der Operator sequence Stream -Join ist ein Werkzeug, um Sequenzen auf bestimmte Eigenschaften (z. B. Monotonie) hin zu überprüfen. Er ist ein unärer Operator, dem drei disjunkte Attributmengen A_G, A_S, A_D und ein Prädikat P übergeben werden. Seine Eingaberelation R_{in} interpretiert er als eine Relation, die mindestens eine, im allgemeinen Fall mehrere Sequenzen enthält. Er baut die Sequenzen anhand der Attributmengen auf und überprüft sie unter Anwendung des Prädikats P. Das Prädikat P wird für alle aufeinanderfolgenden Tupel einer Sequenz geprüft. Sequi -Stream-Join realisiert damit eine Allquantifizierung innerhalb von Sequenzen. Als Ausgabe produziert er ein Ergebnistupel zu jeder Sequenz, die die Allquantifizierung erfüllt. Die Signatur von sequence Stream-Join ist:

$$\geq\!\triangleright(R_{in}, A_G, A_S, A_D, P)$$

Um den Operator und seine Aufrufparameter zu erläutern, sei das Schema einer Relation *WP1* gegeben, die die Tageskurse von Wertpapieren enthält. Im Attribut *WPK* ist die Wertpapierkennummer abgelegt. Die Aktien sollen wochenweise daraufhin untersucht werden, ob ihre Kurse innerhalb der jeweiligen Woche kontinuierlich gestiegen sind.

WP1 (WPK, Woche, Tag, Kurs)

Die Attribute $A_G \cup A_S$ identifizieren die in R_{in} enthaltenen Sequenzen Š. Sie entsprechen der Attributmenge A_{ID} des vorigen Abschnitts. Tupel aus R_{in}, die in $A_G \cup A_S$ übereinstimmen, werden derselben Sequenz zugeordnet. Diese Attribute stellen gleichzeitig die Ausgabeattribute des Operators dar, da zu jeder Sequenz ein Ergebnistupel produziert wird, sofern sie die Allquantifizierung erfüllt. Analog Equi-Stream-Join gilt:

$$A_G \cup A_S = A_{out}$$

Die Attribute A_G fassen alle Sequenzen in einer Gruppe G zusammen, die für die gleichen Werte in den Attributen A_D geprüft werden sollen. Die Allquantifizierung wird für die Sequenzen einer Gruppe über den gemeinsamen Divisor $G.A_D$ durchgeführt. Er ergibt sich aus:

$$G.A_D = \{r.A_D \mid r \in R_{in} \wedge r.A_G = G.A_G\}$$

Im Beispiel ist $R_{in} = WP1$ und $A_G = \{Woche\}$. Zu jeder Woche wird eine Gruppe G_i gebildet und der Divisor $G_i.A_D$ enthält die fünf Handelstage der jeweiligen Woche. Eine Sequenz Š besteht somit aus fünf Werten. Sie wird innerhalb ihrer Gruppe durch die Attribute A_S identifiziert. Im Beispiel ist $A_S = \{WPK\}$. Die Attribute A_D entsprechen den Attributen A_{Ord} des vorherigen Abschnitts. Sie definieren neben dem Divisor die Ordnung innerhalb der Sequenzen. Im Beispiel ist $A_D = \{Tag\}$, da die Kurse von Aktien nach dem zeitlichen Kriterium Tag geordnet werden müssen, um den Kursverlauf zu erhalten.

Die im Prädikat P enthaltene Bedingung wird für alle aufeinanderfolgenden Tupel einer Sequenz geprüft. Seine syntaktische Darstellung wird im folgenden Abschnitt angegeben. Im Beispiel enthält P das Attribut *Kurs* und prüft für aufeinanderfolgende Kurse einer Aktie, ob sie monoton steigen.

6.1.1. Funktionsweise

Dieser Abschnitt geht der formalen Spezifikation des Algorithmus voraus. Er erläutert schrittweise die algorithmische Vorgehensweise des Operators Sequi-Stream-Join.

Vorbereitungsphase

Da Relationen ungeordnete Tupelmengen sind, müssen in einer V orbereitungsphase zunächst die Sequenzen aufgebaut werden. Als Strukturinformation stehen dem Operator dazu die Gruppenattribute A_G, die Sequenzattribute A_S und die Stream -Attribute A_D zur Verfügung. Analog equivalence Stream-Join gruppiert er seine Eingabe anhand der Attribute A_G in Partitionen G_i. Innerhalb jeder dieser Partitionen erfolgt dann jedoch eine Sortierung der Tupel nach den Stream -Attributen A_D in Streams $S_{i,j}$. Ein Stream enthält alle Tupel, die in den Attributen A_D übereinstimmen. Die Sequenzattribute A_S entsprechen den Dividendenattributen A_R von Equi -Stream-Join. Sie ordnen innerhalb einer Gruppe jedes Tupel einer Sequenz zu.

G_i	$\check{S}_{i,2}$	$\check{S}_{i,3}$	$\check{S}_{i,1}$	$\check{S}_{i,5}$	$\check{S}_{i,4}$
$S_{i,1}$	t_2	t_{17}	t_{12}	t_5	t_9
$S_{i,2}$		t_1	t_7	t_{18}	t_{14}
$S_{i,3}$	t_{16}	t_{10}	t_{15}		t_{13}
$S_{i,4}$	t_8	t_{11}	t_6	t_3	t_4

Tabelle 6.1: Gruppe G_i nach Sortierung

Man betrachte die schematische Darstellung der Gruppe G_i als Tabelle. Die Gruppe enthält 18 Tupel, die sich auf vier aufsteigend angeordnete Streams verteilen. Quer zu den Streams verlaufen fünf Sequenzen. Aufgrund der Eindeutigkeit von Sequenzen darf jeder Stream von jeder Sequenz höchstens ein Tupel enthalten. Die Sequenzen $\check{S}_{i,2}$ und $\check{S}_{i,5}$ sind unvollständig, da sie im Stream $S_{i,2}$ bzw. $S_{i,3}$ kein Tupel besitzen. Innerhalb der Streams herrscht keine Sortierordnung. Das erste Tupel eines jeden Streams gehört also nicht notwendigerweise zur Sequenz $\check{S}_{i,2}$. Die beliebige Anordnung der Sequenzen soll dies zumindest andeutungsweise zum Ausdruck bringen.

Die Sortierordnung innerhal b einer einzelnen Sequenzen $\check{S}_{i,k}$ wird dadurch aufgebaut, dass die Datenbehälter ihrer Tupel, die Streams, sortiert angeordnet werden. Die Datenstruktur ermöglicht es nicht, sequentiell auf die Tupel einer einzelnen Sequenz zugreifen zu können. Es ist charakteristisch für Stream-Join, dass die Verarbeitung einer Gruppe nicht sequenzweise sondern streamweise erfolgt. Streams werden sequentiell und Sequenzen parallel zueinander verarbeitet. Eine Sequenz ist daher als eine logische Datenstruktur zu betrachten, deren Tupel über die entsprechende Belegung in den Attributen A_S identifiziert werden. Um die Sequenzen innerhalb einer Gruppe G_i bezeichnen zu können, wird jeder in der Gruppe auftretenden Belegung von A_S ein Index $k \in \angle$ zugeordnet. Die zu einer Belegung k gehörigen Tupel sind in G_i durch die Anordnung der Streams $S_{i,j}$ nach A_D sortiert und bilden so die Sequenz $\check{S}_{i,k}$.

Allquantifizierung

Nach der Strukturierung der Eingaberelation R_{in} in Gruppen G_i mit sortiert angeordneten Streams $S_{i,j}$ führt Sequi-Stream-Join in jeder Gruppe eine Allquantifizierung durch. $S_{i,j}$ und $S_{i,j+1}$ sind Streams einer Gruppe G_i, die gemäß der durch A_D gegebenen Sortierordnung unmittelbar aufeinander folgen. Für eine Sequenz $\check{S}_{i,k}$ wird geprüft, ob sie für alle Paare $(S_{i,j}, S_{i,j+1})$ der Gruppe die Bedingung P erfüllt. Zu jedem Streampaar muss ein Tupelpaar (t_j, t_{j+1}) mit $t_j, t_{j+1} \in \check{S}_{i,k}$, $t_j \in S_{i,j}$ und $t_{j+1} \in S_{i,j+1}$ existieren, das $P(t_j, t_{j+1})$ erfüllt. Sei $J(G_i)$ die Indexmenge der Streams der Gruppe G_i unter Ausschluss des Index j_{max} des letzten Streams, so gilt:

$$\check{S}_{i,k} \in R_{out} \Leftrightarrow \forall j \in J(G_i)\ \exists t_j \in S_{i,j},\ t_{j+1} \in S_{i,j+1}:\ t_j, t_{j+1} \in \check{S}_{i,k} \wedge P(t_j, t_{j+1})$$

Für eine Sequenz $\check{S}_{i,k}$, die die Allquantifizierung erfüllt, wird ein Ergebnistupel in der Ausgaberelation R_{out} abgelegt. Eine Sequenz kann die Allquantifizierung nur dann erfüllen, wenn sie in jedem Stream ihrer Gruppe mit einem Tupel vertreten ist. Die unvollständigen Sequenzen $\check{S}_{i,2}$ und $\check{S}_{i,5}$ aus Tabelle 6.1 scheiden daher unabhängig von der in P formulierten Bedingung aus. Bei einer vollständigen Sequenz entscheidet P darüber, ob sie in die Ergebnisrelation übernommen wird oder nicht.

Innerhalb einer Gruppe G_i definiert jede Belegung $G_i.A_D$ einen Stream. Die in einer Gruppe enthaltenen Sequenzen definieren also über die von ihnen in A_D eingebrachten Belegungen den Divisor $G_i.A_D$. In ihrer Gesamtheit legen sie für ihre Gruppe fest, welche Belegungen eine vollständige Sequenz in A_D haben muss. Nur Sequenzen, die vollständig sind ($\check{S}_{i,k}.A_D = G_i.A_D$), können sich für das Ergebnis qualifizieren. Unvollständige Sequenzen scheiden von vorne herein aus. Jede Gruppe wird unabhängig von der Existenz und dem Inhalt anderer Gruppen bearbeitet. Auf den Divisor einer Gruppe haben nur die Sequenzen derselben Gruppe Einfluss.

Sequi-Stream-Join verwendet die drei disjunkten Attributmengen A_G, A_S, A_D. Weiter bezeichne A_{out} die Attribute der Ausgaberelation R_{out} und A_P diejenigen Attribute, die im Prädikat P verwendet werden. Ist die Attributmenge A_G leer, entfällt die globale Gruppierung der Eingaberelation. Die Allquantifizierung wird für alle Sequenzen aus R_{in} über denselben Divisor $R_{in}.A_D$ durchgeführt. Ist A_S leer, wird jede Gruppe G_i als eine Sequenz interpretiert. Jeder Stream enthält genau ein Tupel und alle Sequenzen aus R_{in} werden unabhängig voneinander betrachtet. Beide Attributmengen A_G und A_S dürfen nicht leer sein, da sonst $A_{out} = A_G \cup A_S$ leer wäre und die Ausgaberelation R_{out} keine Attribute besäße.

Wenn A_P leer ist bzw. P fehlt, wird lediglich geprüft, welche Sequenzen in allen Streams mit einem Tupel vertreten sind. Sequi-Stream-Join prüft also nur, ob die Sequenzen vollständig sind ($\check{S}_{i,k}.A_D = G_i.A_D$). Die Menge der Stream-Attribute A_D darf nicht leer sein, da dann keine Sequenzen mehr vorlägen (siehe Definition von Sequenz).

Im Gegensatz zu equivalence Stream-Join ist P kein reines Restriktionsprädikat. Es müssen Bedingungen formulierbar sein, die innerhalb jeder Sequenz nur zwischen Tupeln aufeinanderfolgender Streams $S_{i,j}$ und $S_{i,j+1}$ überprüft werden. Um solche Bedingungen formulieren zu können, werden die Bezeichner *ir* und *next* eingeführt. Ersterer bezeichnet die Kandidatenmenge *IR* der aktuellen Gruppe (d.h. der Gruppe, die gerade verarbeitet wird). *IR* enthält zu jeder Sequenz der Gruppe ein Tupel t_{IR} mit den Attributen $A_S \cup A_P$ als Repräsentant. Jeder Repräsentant wird in den Attributen A_P mit den Werten initialisiert, die seine Sequenz $\check{S}_{i,k}$ im ersten Stream $S_{i,1}$ aufweist. Nach dem Vergleich von *IR* mit einem Stream $S_{i,j}$ werden die Werte in A_P aktualisiert, so dass die Repräsentanten t_{IR} stets die Werte des zuletzt verarbeiteten Streams $S_{i,j}$ enthalten. Der

zweite Bezeichner *next* steht für den Folgestream $S_{next} = S_{i,j+1}$ des zuletzt verarbeiteten Streams.

Sequi-Stream-Join vergleicht die Tupel von aufeinanderfolgenden Streams $S_{i,j}$, $S_{i,j+1}$ dadurch, dass er die Tupel aus *IR* mit den Tupeln aus dem Folgestream S_{next} vergleicht. Mit den Vergleichsoperatoren $\theta = \{<, \leq, =, \geq, >\}$ und den Bezeichnern *ir*, *next* können zu Attributen $a_{p_1}, a_{p_2} \in A_P$ außer den herkömmlichen Bedingungen auch Bedingungen der Art $(ir.a_{p_1}\ \theta\ next.a_{p_2})$ im Prädikat *P* formuliert werden. Eine solche Bedingung, die Tupel aus *IR* mit den Tupeln des Folgestreams S_{next} vergleicht, heißt *Sequenzbedingung*.

Der Algorithmus erzeugt sein Ergebnis durch sukzessive Vergleiche von *IR* mit allen Streams $S_{i,j}$ der Gruppe. Im Unterschied zu equivalence Stream -Join ist die Vergleichs - reihenfolge der Streams durch die Sortierung nach A_D vorgegeben. Der Vergleich von *IR* mit einem Stream S_{next} erfolgt tupelweise und wird so durchgeführt, dass die Kandidatenmenge *IR* zu jedem Zeitpunkt stets nur diejenigen Sequenzen enthält, die die Bedingung *P* für alle Streams erfüllt haben, mit denen *IR* bis dahin verglichen worden ist. *IR* stellt zu jedem Zeitpunkt das Zwischenergebnis (**I**ntermediate **R**esult) des Algorithmus dar.

Nach dem Vergleich von *IR* mit dem letzten Stream der Gruppe werden die Tupel von *IR* zusammen mit den Attributen A_G der Gruppe in die Ergebnisrelation R_{out} projiziert. Die Attribute A_P werden nicht übernommen. R_{out} enthält ein Ergebnistupel für jede Sequenz $\check{S}_{i,k}$, die die beschriebene Allquantifizierung erfüllt. Sie wird durch ihre Belegung in den Attributen $A_G \cup A_S$ identifiziert. Die Sequenz an sich wird nicht ausgegeben.

6.2. Der Algorithmus Sequi-Stream-Join

Die algorithmische Vorgehensweise des Operators Sequi -Stream-Join zur Realisierung von Allquantifizierung innerhalb von Sequenzen wird in diesem Abschnitt formal spezifiziert. Der Spezifikation gehen ähnliche Vorüberlegung wie bei equivalence Stream-Join im Abschnitt 4.2.1 voraus.

6.2.1. Vorüberlegungen

Gegeben ist eine Eingaberelation R_{in} mit den Attributen *A*. Sie wird von Sequi -Stream-Join durch A_G in Gruppen G_i partitioniert. Innerhalb der Gruppen werden die Tupel nach den Attributen A_D sortiert, um die Sequenzen $\check{S}_{i,k}$ aufzubauen. Die Tupel einer Sequenz sind durch die physische Sortierung nach A_D auf die Streams der Gruppe verteilt. Eine Relation bzw. eine Gruppe ist eine lineare Datenstruktur und kann zu einem Zeitpunkt immer nur nach einer Attributmenge physisch gruppiert oder sortiert werden. Da die Gruppen bereits nach A_D physisch sortiert werden, muss die Gruppierung nach A_S in Sequenzen $\check{S}_{i,k}$ auf andere Weise erfolgen.

Der Algorithmus verwendet hierzu ein internes Prädikat P_S. Es enthält alle Sequenz-attribute A_S und prüft bezüglich zweier Tupel t_1 und t_2, ob sie in den Attributen A_S übereinstimmen. P_S ist genau dann wahr, wenn beide Tupel zu derselben Sequenz gehören. Beispielsweise hat es für $A_S = \{a_1, a_2\}$ die Form:

$$P_S(t_1, t_2) = ((t_1.a_1 = t_2.a_1) \wedge (t_1.a_2 = t_2.a_2))$$

Zum Bearbeiten einer Gruppe G_i verwendet der Algorithmus die Kandidatenmenge IR. Sie enthält die Repräsentanten zu den Sequenzen der Gruppe, die sich noch nicht disqualifiziert haben. Der Kern des Algorithmus besteht darin, für die Sequenzen aus IR zu prüfen, ob sie im Folgestream enthalten sind und ob ihre Tupel $t_{IR} \in IR$ und $t_{next} \in S_{next}$ das Prädikat $P(t_{IR}, t_{next})$ erfüllen. Der Vergleich von IR mit einem Folgestream S_{next} erfolgt daher tupelweise: Für jedes Tupel $t_{IR} \in IR$ wird geprüft, ob ein Tupel $t_{next} \in S_{next}$ existiert, das $P_S(t_{IR}, t_{next})$ und $P(t_{IR}, t_{next})$ erfüllt. Dem tupelweisen Vergleich liegt also das zusammengesetztes Prädikat $P_J(t_{IR}, t_{next}) = P_S(t_{IR}, t_{next}) \wedge P(t_{IR}, t_{next})$ zugrunde. In Bezug auf IR und den Folgestream S_{next} entspricht P_J einem Joinprädikat, da es zu einem Tupel aus IR prüft, ob im Folgestream ein korrespondierendes Tupel enthalten ist. Der Algorithmus erzeugt sein Ergebnis durch sukzessive Vergleiche von IR mit allen Streams der Gruppe.

6.2.2. Spezifikation

Aufbauend auf den Vorüberlegungen im letzten Abschnitt kann nun die formale Spezifikation der algorithmischen Vorgehensweise des Operators Sequi-Stream-Join angegeben werden.

(0) gruppiere R_{in} nach A_G in Gruppen G_i
    sortiere jede Gruppe G_i nach A_D in Streams $S_{i,j}$
(1) initialisiere IR mit den Sequenzen des ersten Streams $S_{i,1}$: $IR = \pi_{A_S \cup A_P}(S_{i,1})$
(2) überprüfe die Sequenzen aus IR anhand des Folgestreams S_{next}:
    (a) wenn zum Repräsentant $t_{IR} \in IR$ einer Sequenz $\exists t_{next} \in S_{next}: P_J(t_{IR}, t_{next})$
        dann setze die Belegung des Repräsentanten $t_{IR}.A_P$ auf $t_{next}.A_P$
        wenn nicht, entferne Repräsentant t_{IR} aus IR
    (b) wiederhole Schritt (a) für alle anderen Sequenzen aus IR
(3) solange $IR \neq \varnothing$, wiederhole Schritt (2) für alle folgenden Streams $S_{i,j}$ der Gruppe G_i
(4) ergänze die in IR enthaltenen Repräsentanten um die Belegung $G_i.A_G$ der Gruppe und projiziere sie ohne die Attribute A_P die Ausgaberelation R_{out}
(5) wiederhole die Schritte (1) bis (4) für alle anderen Gruppen von R_{in}

In der Vorbereitungsphase (0) wird die Eingaberelation R_{in} in Gruppen G_i partitioniert. Die Sortierung nach A_D erzeugt die Streams innerhalb jeder Gruppe. Der Algorithmus beginnt mit der Verarbeitung der ersten Gruppe G_1. Er initialisiert die Kandidatenmenge IR mit den Tupeln des ersten Streams $S_{1,1}$, indem er die Attribute $A_S \cup A_P$ nach IR projiziert. Jedes Tupel in IR repräsentiert eine Sequenz $\check{S}_{1,k}$ und verfügt in A_P über die Attribute, die zur Auswertung des Prädikats P benötigt werden. Eine Duplikat-

eliminierung ist hier nicht notwendig, da die Sequenzen bezüglich $A_S \cup A_D$ eindeutig sein müssen, d.h. in einem Stream keine Duplikate enthalten dürfen.

In Schritt (2) wird *IR* mit dem jeweils nächsten Stream S_{next} der Gruppe verglichen. Die Reihenfolge, welcher Stream der jeweils nächste ist, ist durch die Sortierung nach A_D vorgegeben. Für jedes Tupel $t_{IR} \in IR$ wird geprüft, ob in S_{next} ein Tupel t_{next} enthalten ist, das $P_J(t_{IR}, t_{next})$ erfüllt. Existiert ein solches Tupel t_{next}, verbleibt die zu t_{IR} gehörige Sequenz in der Kandidatenmenge *IR*. Ihr Repräsentant wird aktualisiert, indem t_{IR} in den Attributen A_P die Werte von t_{next} übernimmt. Existiert kein solches Tupel t_{next}, wird das Tupel der Sequenz aus *IR* entfernt, und sie wird in darauf folgenden Vergleichen nicht weiter berücksichtigt.

Mit jedem Schritt (2a) kann ein Tupel aus *IR* entfernt werden. Daher wird nach jedem Vergleich mit einem Stream in (3) geprüft, ob *IR* leer ist. Wenn nicht, wird Schritt (2) wiederholt, bis *IR* für alle Streams der Gruppe überprüft wurde. Das Ergebnis der Gruppe steht in Schritt (4) fest. Es enthält maximal alle Repräsentanten der Sequenzen, die im ersten Stream vorkommen, und minimal die leere Menge. Die Ergebnistupel werden um die Gruppenattribute A_G ergänzt und ohne die Attribute A_P in R_{out} abgelegt. Der Algorithmus fährt mit der nächsten Gruppe fort, bis er alle Gruppen von R_{in} verarbeitet hat. Bezüglich der Gruppen spielt es keine Rolle, in welcher Reihenfolge sie verarbeitet werden.

Unterschiede zu Equi-Stream-Join

Trotz grundsätzlicher Übereinstimmung in der Vorgehensweise gibt es vier signifikante Unterschiede, die die Algorithmen Equi-Stream-Join und Sequi-Stream-Join voneinander unterscheiden.

- In der Vorbereitungsphase führt Sequi-Stream-Join in den Gruppen G_i eine Sortierung anstatt einer Gruppierung nach A_D durch. Die bei Equi-Stream-Join beliebige Vergleichsreihenfolge von *IR* mit den Streams ist dadurch bei Sequi-Stream-Join festgelegt.
- Bei der Initialisierung von *IR* wird keine Duplikateliminierung durchgeführt. Es wird vorausgesetzt, dass die Sequenzen in R_{in} der Definition genügen.
- Die Kandidatenmenge *IR* enthält außer den Attributen A_S, die die Sequenzen identifizieren, auch die Attribute A_P, die im Prädikat *P* verwendet werden. Die Werte der Repräsentanten in A_P werden beim Vergleichen mit den Streams aktualisiert und folgen dadurch dem Verlauf ihrer Sequenz.
- Das Prädikat *P* wird nicht auf Tupel der Streams sondern auf Tupelpaare angewandt, wobei ein Tupel aus *IR* und das andere aus dem Folgestream S_{next} stammt. Die bei Equi-Stream-Join beschriebene vorgezogene Auswertung des Prädikats *P* ist bei Sequi-Stream-Join nur für die Teilbedingungen des Prädikats möglich, die keine Sequenzbedingungen sind.

Beispiel 6.1

Gegeben sei eine Relation *WP2(Tag, Aktie, Kurs)*. Sie enthält für den Beobachtungszeitraum einer Woche von Montag bis Freitag die Schlusskurse der Automobilwerte DaimlerChrysler, Porsche, Audi und VW. Mit Hilfe von Sequi-Stream-Join sollen diejenigen Aktien ermittelt werden, deren Kurse im Beobachtungszeitraum kontinuierlich gestiegen sind.

R_{in}	Tag	Aktie	Kurs
S_1	Mo	DC	100
	Mo	Po	250
	Mo	Au	20
	Mo	VW	60
S_2	Di	Po	240
	Di	DC	102
	Di	Au	26
	Di	VW	62
S_3	Mi	VW	66
	Mi	DC	105
	Mi	Po	245
S_4	Do	Au	30
	Do	VW	68
	Do	Po	250
	Do	DC	110
S_5	Fr	VW	70
	Fr	Po	254
	Fr	DC	112

Tabelle 6.2: Eingaberelation R_{in}

IR	Aktie	Kurs
	DC	100
	Po	250
	Au	20
	VW	60
	DC	102
	Au	26
	VW	62
	DC	105
	VW	66
	DC	110
	VW	68
	DC	112
	VW	70

Tabelle 6.3: Inhalt von IR

Die Kurse jeder Akti e stellen eine Sequenz dar. Alle Aktien sollen für den gleichen Zeitraum überprüft werden. Sie können also in einer Gruppe angeordnet ($A_G = \varnothing$) und durch $A_S = \{Aktie\}$ identifiziert werden. Durch das Attribut $A_D = \{Tag\}$ legt man die zeitliche Ordnung innerha lb der Sequenzen fest. Tabelle 6.2 zeigt die Eingaberelation $R_{in} = WP2$ des Operators nach der Vorbereitungsphase. Sie besteht aus einer Gruppe, die in fünf Streams strukturiert wurde. Die Streams sind nach den Tagen aufsteigend angeordnet. Sie enthalten die Tupel der vier Sequenzen $\check{S}_k$. Im Prädikat P gibt man die Sequenzbedingung für kontinuierlich steigende Kurse an:

$$P = (ir.Kurs < next.Kurs) \quad \Rightarrow A_P = \{Kurs\}$$

Aus Prädikat P und Attribut $A_S = \{Aktie\}$ ergibt sich das zusammengesetzte Prädi kat P_J für den tupelweisen Vergleich von IR mit dem jeweiligen Folgestream:

$$P_J(t_{IR}, t_{next}) = (t_{IR}.Aktie = t_{next}.Aktie) \wedge (t_{IR}.Kurs < t_{next}.Kurs)$$

Die Kandidatenmenge IR enthält die Attribute $A_S \cup A_P = \{Aktie, Kurs\}$. Tabelle 6.3 gibt den Inhalt von IR nach der Initialisierung und nach jedem Vergleich mit den Streams S_2, S_3, S_4 und S_5 an. In ihr spiegelt sich die Arbeitsweise von Sequi -Stream-Join anschaulich wieder. Der Operator initialisiert IR mit den vier Tupeln aus Stream S_1. Es folgt der Vergleich von IR mit dem Folgestream $S_{next} = S_2$. Anhand von Prädikat P_J wird zu jedem Tupel aus IR in S_{next} nach einem Tupel gesucht, das im Attribut *Aktie* übereinstimmt. Aufgrund der Eindeutigkeit von Sequenzen darf höchstens ein solches

Tupel im Stream existieren. Der eigentliche Vergleich erfolgt durch die Auswertung der Bedingung von Prädikat P. Sie prüft, ob der Kurs gestiegen ist. Dies trifft für alle Aktien außer Porsche zu. Ihre Repräsentanten übernehmen im Attribut $A_P = \{Kurs\}$ die Werte aus S_2, wohingegen das Tupel von Porsche aus IR entfernt wird. Es folgt der Vergleich mit Stream S_3. In diesem Stream findet der Algorithmus kein Tupel für die Aktie von Audi. Ihr Repräsentant wird aus IR entfernt. Die Aktien von VW und DaimlerChrysler weisen erneut steigende Kurse auf. Ihre Repräsentanten werden wiederum im Attribut *Kurs* aktualisiert. Dasselbe ist bei den Vergleichen von IR mit den Streams S_4 und S_5 der Fall. Die Kandidatenmenge IR enthält am Schluss die Tupel dieser beiden Aktien als Ergebnis der Allquantifizierung. Da A_G leer ist, wird nur das Attribut $A_S = \{Aktie\}$ der Tupel in die Ausgaberelation R_{out} übernommen:

R_{out}	Aktie
	DC
	VW

Tabelle 6.4: Ausgaberelation R_{out}

6.2.3. Effiziente Implementierung

Für den Operator Equi-Stream-Join ist in Abschnitt 4.2.3 detailliert beschrieben, wie er effizient implementiert werden kann. Aufgrund der Verwandtschaft beider Operatoren kann die Implementierung bis auf punktuelle Anpassungen für Sequi-Stream-Join übernommen werden. Sie wird in diesem Abschnitt mit einschließlich der erforderlichen Anpassungen skizziert.

Die Eingaberelation R_{in} ist nach den Attributen A_G zu gruppieren und in den Gruppen nach den Attributen A_D zu sortieren. Beide Operationen können auch in einem Schritt als Sortierung nach $A_G \cup A_D$ durchgeführt werden, wobei die Priorität der Attribute A_G zu berücksichtigen ist. Anhand der im Datenbanksystem vorhandenen Operatoren ist zu entscheiden, ob eine Gruppierung mit anschließender Sortierung in den Gruppen oder eine globale Sortierung günstiger ist. Meist werden die logischen Operatoren zum Gruppieren und Sortieren auf hash-basierte physische Operatoren abgebildet. Sie garantieren linearen Aufwand und sind am effizientesten, wenn nicht gerade ein passender Index für A_G oder A_D zur Verfügung steht.

Der in (2) spezifizierte, tupelweise Vergleich der Kandidaten aus IR mit den Streams $S_{i,j}$ der Gruppe G_i wird durch einen Hash-Join-Algorithmus implementiert. Er garantiert stets linearen Aufwand bezüglich seiner Eingabekardinalitäten $|IR|$ und $|S_{i,j}|$. Build-Relation ist die Kandidatenmenge IR. Die zugehörige Hash-Tabelle H enthält die Attribute $A_S \cup A_P$ und ein Attribut c, das einen Zähler (counter) realisiert. Sie wird mit den Tupeln des ersten Streams initialisiert, wobei der Zähler für jedes Tupel auf den Wert Eins gesetzt wird. Die Hash-Funktion hängt nur von den Sequenzattributen A_S ab.

Die Streams der Gruppe sind die Proberelationen. Die Überprüfung der Tupel aus IR erfolgt durch Proben mit dem jeweils nächsten Stream S_{next} der Gruppe. Zu jedem Tupel t_{next} des Streams wird über die Hash-Funktion das Tupel t_{IR} der zugehörigen Sequenz mit $t_{IR}.A_S = t_{next}.A_S$ gesucht. Existiert t_{IR}, wird anhand der Bedingung $t_{IR}.c = next\text{-}1$

geprüft, ob der Zähler des Kandidaten den Index des vorigen Streams enthält. Wenn ja, hat die Sequenz des Kandidaten in allen bisher geprüften Streams das Prädikat P erfüllt. Ihr Repräsentant ist gültig und das Prädikat $P(t_{IR}, t_{next})$ kann für die Sequenz ausgewertet werden. Ist P erfüllt, wird der Repräsentant der Sequenz aktualisiert. Der Zähler $t_{IR}.c$ wird auf den Index *next* des aktuellen Streams und die Belegung $t_{IR}.A_P$ auf $t_{next}.A_P$ gesetzt. Nach Proben eines Streams S_{next} kann anhand der Zähler festgestellt werden, ob ein Tupel t_{IR} ein gültiger Repräsentant ist $(t_{IR}.c = next)$. Enthält der Zähler einen Wert kleiner *next*, ist der Repräsentant ungültig.

Die Realisierung mit dem Zähler hat den Vorteil, dass die Hash-Tabelle nicht nach jedem Proben gescannt werden muss, um die ungültigen Kandidaten aus H zu löschen. Beim Proben mit dem letzten Stream S_{last} einer Gruppe wird mit dem Setzen des Zählers auf *last* das entsprechende Tupel $t_{IR}.A_S$ um die Gruppenattribute A_G ergänzt und in der Ausgaberelation R_{out} abgelegt. Dadurch entfällt ein abschließendes Scannen der Hash-Tabelle. Der Operator fährt mit der nächsten Gruppe fort.

Optimierung

Die Effizienz der angegebenen Implementierung kann durch folgende Optimierung erhöht werden. Der Algorithmus soll für eine Gruppe vorzeitig terminieren, wenn die Kandidatenmenge *IR* leer ist. Dies kann über ein Flag *IR_leer* geprüft werden. Es wird vor jedem Proben auf *wahr* gesetzt. Beim ersten Tupel, dessen Zähler erhöht wird, wird *IR_leer* auf *falsch* gesetzt. Ist *IR_leer* nach dem Proben des Streams immer noch auf *wahr* gesetzt, so sind alle Repräsentanten ungültig. Die Gruppe trägt zum Gesamtergebnis nichts bei, da *IR* bereits leer ist. Der Algorithmus fährt mit der Bearbeitung der nächsten Gruppe fort.

Die für Equi-Stream-Join in Abschnitt 4.2.3 angegebene Optimierung durch die vorgezogene Anwendung des Prädikats P kann auf den Operator Sequi-Stream-Join nicht übertragen werden. Sein Prädikat P enthält Sequenzbedingungen, die stets für Tupelpaare t_{IR}, t_{next} aus der Menge *IR* und dem Folgestream S_{next} ausgewertet werden müssen. Es könnte lediglich die Auswertung von Bedingungen aus P vorgezogen werden, die keine Sequenzbedingungen sind. Ob sich dadurch eine nennenswerte Steigerung der Performance erreichen lässt, ist fraglich. Der Fokus von Sequi-Stream-Join liegt auf Allquantifizierung in Sequenzen. Daher ist zu erwarten, dass im Prädikat P die Sequenzbedingungen überwiegen bzw. dass es nur Sequenzbedingungen enthält. Nur wenn man weiß, dass die zu bearbeitenden Anfragen im Prädikat P selektive Bedingungen enthalten, die keine Sequenzbedingungen sind, sollte man es in Betracht ziehen, die vorgezogene Auswertung dieser Bedingungen zu implementieren.

6.3. Eigenschaften des Algorithmus

Der Operator blockiert in den Schritten (0) bis (3) während der Gruppierung bzw. der Sortierung und der Bearbeitung einer Gruppe. Enthält R_{in} nur eine Gruppe, ist Sequi-Stream-Join ein blockierender Operator. Bei mehreren Gruppen wird die Blockierung

durchbrochen und tritt nur phasenweise auf. Die Ausgabe des Operators geht mit zunehmender Anzahl an Gruppen in einen kontinuierlichen Datenstrom über. In diesem Fall kann im Operatorbaum eine Pipeline zum Nachfolger aufgebaut werden.

Sequi-Stream-Join ist als unärer Operator nicht in der Lage, Sequenz en direkt miteinander zu vergleichen bzw. so zu verarbeiten, wie dies in der im nächsten Kapitel beschriebenen Anfragesprache SRQL möglich ist. Sein Einsatz beschränkt sich auf die Durchführung von Allquantifizierung innerhalb von Sequenzen. Er ist ein Ana lyse-werkzeug. Bestimmte Vergleiche von Sequenzen sind dadurch möglich, dass die zu vergleichenden Sequenzen durch einen Join zu einem Eingabestrom verbunden und dem Operator übergeben werden. Dieses Vorgehen erlaubt es Sequi -Stream-Join, wertebasierte Seq uenzvergleiche durchzuführen. Positionsbasierte Sequenzvergleiche kann der Operator nicht durchführen. Im Joinprädikat können nur Attributwerte der zu verarbeitenden Tupel verglichen werden, nicht jedoch die Positionen, die die Tupel innerhalb ihrer Herkun ftsrelationen hätten, wenn diese sortiert würden. Für die positionsbasierte Verarbeitung von Sequenzen müssen zwei Voraussetzungen gegeben sein: Sequenzen müssen als logische Datenstruktur existieren und sie müssen mit einem Positionsattribut versehen sein. In einem relationalen Schema ist beides nicht gegeben.

Sequi-Stream-Join ist ein Operator, der Sequenzen analysieren kann. Da seine Umgebung, das relationale Datenbanksystem, keine Sequenzen kennt, ergeben sich Einschränkungen hinsichtlich seiner Einsetz barkeit. Dies spiegelt sich in dem Umstand wieder, dass er die Sequenzen, die er untersuchen soll, selbst durch Sortierung aufbauen muss. Im Kapitel 7 wird die Anfragesprache SRQL vorgestellt. Sie ist eine Erweiterung von SQL und ermöglicht das Erzeugen und Verarbeiten von Sequenzen.

6.3.1. Aufwandsabschätzung

Die Aufwandsabschätzung ermöglicht es, die Effizienz von Sequi -Stream-Join zu bewerten. Sie ist die Grundlage für kostenbasierte Optimierer bei der Bewertung von Ausführungsplänen, die den Operator Sequi -Stream-Join enthalten. Die Abschätzung gilt unter der Annahme, dass die Eingaberelation R_{in} sowie die Ergebnisrelation R_{out} im Datenbankpuffer gehalten werden können. Der Aufwand zum Aufbau einer Hash -Tabelle beträgt $k_H\, n$ für eine Relation der Kardinalität n, wobei $k_H \geq 1$. Die Konstante k_H bringt den Aufwand zum Ausdruck, der durch die Kollisions - bzw. Überlaufbehandlung für die Buckets der Hash -Tabelle entsteht. Sie beträgt bei gut gewählten Hash -Funktionen ca. 1,2. Das Proben ei ner Relation der Kardinalität n_2 mit der Hash -Tabelle verursacht einen Aufwand von $k_H\, n_2$.

Sei $n_{in} = |R_{in}|$ die Kardinalität der Eingaberelation und $n_S = |R_{in}.A_G \cup A_D|$ die Summe der Kardinalitäten der Divisoren aller Gruppen. Es gilt $n_S \leq n_{in}$ und es gibt insge samt genau n_S Streams. Bei Gleichverteilung beträgt die Kardinalität eines Streams n_{in}/n_S. Sie entspricht der Kardinalität von IR. Die Eingaberelation wird in n_G Gruppen G_i partitioniert. Die Gruppierung der Eingaberelation R_{in} nach den Attributen A_G und d ie anschließende Sortierung nach den Attributen A_D wird von einem hash -basierten Algorithmus als eine physische Sortierung nach $A_G \cup A_D$ vorgenommen. Der Aufwand von Sequi-Stream-Join zur Realisierung von Allquantifizierung in Sequenzen beträgt:

(1) Sortieren der Eingaberelation R_{in}:	$k_H\, n_{in}$
(2) Aufbau der n_G Hash-Tabellen von IR:	$(k_H\, n_{in} / n_S)\, n_G$
(3) Proben der übrigen $(n_S - n_G)$ Streams mit IR: $(k_H\, n_{in} / n_S)\, (n_S - n_G) =$	$k_H\, n_{in} - (k_H\, n_{in} / n_S)\, n_G$
Gesamtaufwand:	$2\, k_H\, n_{in}$
mit $k_H = 1{,}2$:	$2{,}4\, n_{in}$

Der Aufwand von Sequi-Stream-Join als One-Pass Algorithmus ist linear. Er stimmt mit dem Aufwand überein, der für Equi -Stream-Join in Abschnitt 4.3.1 im One -Pass Szenario bei relationaler Division und unmittelbarer Anwendbarkeit angegeben ist.

6.4. Mustererkennung in Genomdatenbanken

Sequi-Stream-Join kann entgegen der Darstellung in [3] nicht zur Mustererkennung in Genomdatenbanken eingesetzt werden. Der Operator wird dort auf die Eingaberelation $R_{in} = IR$ angewendet. (Die Relation heißt zufällig *IR*. Die Kandidatenmenge *IR* ist an dieser Stelle nicht gemeint.) Er wird mit den Attributmengen $A_G = \{patternkey\}$, $A_S = \emptyset$ und $A_D = \{letter\}$ aufgerufen. Die Sequenz, die durch die Sortierung nach *letter* aufgebaut wird, genügt d er Definition von Sequenzen nicht. Sie enthält zu Belegungen von $A_D \cup A_S$ mehrfache Einträge und ist daher nicht eindeutig. Für *letter*=„*A*“ enthält sie beispielsweise die Werte 1 und 5. Bei der Verarbeitung einer solchen Sequenz mit Sequi-Stream-Join führen die Duplikate zu einer unklaren Semantik des Operators.

Entsprechend der algorithmischen Beschreibung, aus der heraus die Idee des Operators Stream-Join entwickelt wurde, repräsentiert jedes Tupel der Kandidatenmenge *IR* eine Sequenz. Auf Basis von tupelweis en Vergleichen mit dem Folgestream kann man entscheiden, was mit dem jeweiligen Tupel aus *IR* geschehen soll: Entweder es verbleibt in *IR* oder es wird entfernt. Als vorteilhafte Konsequenz für die Komplexität und Effizienz des Algorithmus ergibt sich, dass $|IR|$ im Worst Case konstant bleibt und im Average Case monoton fällt. All dies trifft auf das Szenario der Mustererkennung in Genomdatenbanken nicht zu.

Bei der Mustererkennung werden Gensequenzen daraufhin untersucht, ob und wo sie eine bestimmte Teilsequ enz enthalten. Die gesuchte Teilsequenz wird durch einen Ausdruck beschrieben, der als Muster bezeichnet wird. Das Muster kann an keiner, einer oder mehreren Stellen in der Sequenz vorkommen. Laut [3] sollen zu einem gesuchten Muster die Endpositionen ausgegeben werden, an denen es in der Sequenz enthalten ist. Es wird allerdings keine Aussage darüber gemacht, wie Stream -Join verfahren soll, wenn ein Muster bezüglich einer Endposition mehrfach in einer Gensequenz enthalten ist. Ein Beispiel dafür ist das zweifache Vorkommen des Musters A*B*C in der Gensequenz ABBC. Abgesehen davon kann nicht einmal das weniger harte Existenzproblem, also die Frage, ob das Muster in der Gensequenz enthalten ist, mit Stream-Join gelöst werden.

In Anlehnung an das Szenario aus [3] betrachte man die Gensequenz ABBCCXD, in der das Muster A*B*C*D gesucht wird. Die maximal erlaubte Anzahl an Zwischen - buchstaben, die durch das Symbol * im gesuchten Muster repräsentiert werden , betrage Eins. Die Gensequenz ist in der Relation *GS(pos, letter)* abgelegt. Tabelle 6.5 stellt die Relation nach dem Positionsattribut *pos* sortiert dar. Anhand der in [3] beschriebenen Vorgehensweise soll das Muster unter Einsatz des Operators Stream -Join in der Gensequenz gesucht werden.

Der natürliche Gleichverbund von *GS* und dem Muster über das gemeinsame Attribut *letter* liefert die Eingaberelation R_{in} für Stream -Join. Sie enthält zu jedem Buchstaben des Musters einen Stream, der wiederum zu jeder Position, an der der Buchstabe in der Gensequenz vorkommt, ein Tupel enthält. Tabelle 6.6 zeigt die Eingaberelation R_{in} nach der Strukturierung in sortiert angeordnete Streams. Sie enthält vier Streams S_1, S_2, S_3 und S_4. Um zu verdeutlichen, dass es innerhalb der Streams keine bestimmte Ordnung gibt, kommt im dritten Stream das Tupel *(C,5)* vor dem Tupel *(C,4)*. Zum Buchstaben *X* gibt es keinen Stream in R_{in}, da *X* in dem gesuchten Muster nicht vorkommt.

GS	pos	letter
	1	A
	2	B
	3	B
	4	C
	5	C
	6	X
	7	D

Tabelle 6.5: Gensequenz GS

R_{in}	letter	pos
S_1	A	1
S_2	B	2
	B	3
S_3	C	5
	C	4
S_4	D	7

Tabelle 6.6: Eingaberelation R_{in}

Stream-Join initialisiert *IR* mit dem Inhalt des ersten Streams und vergleicht seine Tupel anschließend mit den Tupeln des jeweils folgenden Streams S_{next} anhand des Prädikats P_J. Da die Menge A_S der Sequenzattribute lee r ist, ist $P_J = P$. Die korrekte Sequenz - bedingung in *P* für maximal einen erlaubten Zwischenbuchstaben * lautet in der hier verwendeten und in der in [3] verwendeten Schreibweise:

$$ir.pos < next.pos \leq ir.pos + 2$$

$$pos.old < pos.new \leq pos.old + 2$$

Schon beim Vergleich von *IR* mit dem zweiten Stream zeigt sich, dass der Operator Stream-Join zur Mustererkennung nicht geeignet ist. Beide Tupel aus S_2 erfüllen das Prädikat *P*. Durch welches Tupel ist *(A,1)* zu ersetzen? Man betrachte zunächs t die Lösung der Existenzfrage: Wir wissen, dass das gesuchte Muster genau einmal in *GS* enthalten ist. Die zugehörigen Tupel sind *(A,1)*, *(B,3)*, *(C,5)* und *(D,7)*. Wenn man das Tupel t_{IR} aus *IR* stets durch das erste Tupel t_{next} des Folgestreams S_{next} ersetzt, das $P(t_{IR}, t_{next})$ erfüllt, definiert man einen nicht-deterministischen Algorithmus. Er kann das richtige Ergebnis liefern oder auch nicht. In diesem Fall würde er das Muster in der Sequenz nicht finden und ein falsches Ergebnis liefern. Aufgrund der fehle nden Ordnung innerhalb der Streams kann sowohl *(B,2)* als auch *(B,3)* das erste Tupel des zweiten Streams sein. Das Ergebnis hängt damit von der zufälligen Anordnung der

Tupel innerhalb der Streams ab. Wendet man Stream -Join auf Sequenzen an, die der Definition nicht genügen und Duplikate enthalten, liefert der Operator nicht - deterministische Ergebnisse. Er kann zur Mustererkennung nicht eingesetzt werden.

Um einen deterministischen Algorithmus zu erhalten, muss man *IR* um ein Tupel erweitern und *(A,1)* durch b eide Tupel *(B,2)* und *(B,3)* ersetzen. Beim Vergleich mit dem dritten Stream wird nochmals ein Tupel hinzugefügt. *(B,2)* wird durch *(C,4)* ersetzt und *(B,3)* durch die beiden Tupel *(C,5)* und *(C,4)*. Das Teilmuster A*B*C kommt dreimal in *GS* vor. Daher entscheidet sich erst im Vergleich mit dem letzten Stream, welches der drei Vorkommen sich zum gesamten Muster vervollständigt. Nur *(C,5)* kann durch *(D,7)* ersetzt werden. Dieser Algorithmus terminiert mit dem korrekten Ergebnis.

In realen Anwendungen hat man es mit Gensequenzen zu tun, deren Länge von wenigen hundert bis zu 2,4 Mio. Zeichen im Falle des längsten Gens (Dystrophin) reichen kann. Das durchschnittliche Protein kodierende Gen hat eine Länge von 27.000 Zeichen. Den Genen liegt das Alphabet *{A, C, G, T}* der vier Nukleinsäuren zugrunde. Unter der Annahme der Gleichverteilung ergeben sich vier Streams mit einer durchschnittlichen Anzahl von 6750 Tupeln pro Stream. Wird für die maximal erlaubte Anzahl an Zwischenbuchstaben * ein höherer Wert als Eins angegeben, nimmt die kombina - torische Vielfalt beim tupelweisen Vergleichen von *IR* mit dem Folgestream S_{next} weiter zu. Dies kann bei dem deterministischen Algorithmus ohne weiteres dazu führen, dass sich die Anzahl der Tupel in *IR* durch die Verarbeitung des nächsten Streams verdoppelt. Bei realen Anwendungen geht es also nicht darum, dem Zwischenergebnis *IR* bei Bedarf ein paar zusätzliche Tupel hinzuzufügen, sondern es seiner Größen - ordnung entsprechend zu erweitern.

Bei dem deterministischen Algorithmus handelt es s ich nicht um eine weitere Stream - Join Variante. Er hat grundsätzlich andere Eigenschaften als die Algorithmen von equivalence und sequence Stream -Join bzw. als der in [3] angegebene Algorithmus für Stream-Join und gehört nicht zum Konzept Stream-Join.

7. Integration von Sequi-Stream-Join in SRQL

Dieses Kapitel stellt die Anfragesprache SRQL vor und beschreibt, welche Arten von Sequenzverarbeitung SRQL und Sequi -Stream-Join ermöglichen. Es integriert beide Konzepte, um ihre Stärken zu kombinieren und eine effiziente Verarbeitung von SRQL-Anfragen zu erreichen, die Allquantifizierung in Sequenzen enthalten.

7.1. Sequenzverarbeitung in SRQL

SRQL heißt Sorted Relational Query Language. Sie wird in [4] beschrieben und stellt eine einfache aber mächtige Erweiterung von SQL dar, um Sequenzen verarbeiten zu können. Grundlage ist die Relationenalgebra einschließlich Gruppierung und definiertem NULL-Wert. Sie wird um die vier neuen Operatoren Sequenz ψ, Shift δ, Shift-All Λ und Fensteraggregat ω erweitert. Mathematisch betrachtet würde die alleinige Erweiterung um den Sequenzoperator ausreichen, da die anderen drei Operatoren in der so erweiterten Algebra durch äquivalente Ausdrücke beschrieben werden können. Der Sequenzoperator enthält also den gesamten Zuwachs an Mächtigkeit von SRQL im Vergleich zu SQL. Er stellt den Kern der Erweiterung dar.

Eine Sequenz ist in SRQL definiert als eine Relation R versehen mit Gruppierungs- und Sortierattributen A_G und A_D. Beide Attributmengen können leer sein. Sequenzen werden durch den Sequenzoperator $\psi(A_G, A_D, R)$ aufgebaut. Er geht in der gleichen Weise wie Sequi-Stream-Join vor, indem er zuerst nach A_G gruppiert und dann innerhalb der Gruppen nach A_D sortiert. In der Terminologie von SRQL werden die Gruppen von R als Sequenzen und ganz R als zusammengesetzte Sequenz bezeichnet. Duplikate sind in SRQL bezüglich der Attribute $A_G \cup A_D$ erlaubt. Man könnte angesichts der grundlegenden Übereinstimmung vermuten, dass beide Ansätze etwa gleich mächtig sind und

ihre Ergebnisse lediglich über verschiedene Wege erreichen. SRQL stellt allerdings einen weitaus mächtigeren Ansatz dar.

Der Grund dafür wurde bereits im Kapitel 6 erwähnt. Ein relationales Datenbanksystem verfügt mit dem Operator Sequi-Stream-Join zwar über einen Operator, der Sequenzen aufbauen und durch Allquantifizierung analysieren kann. Das Datenbanksystem selbst verfügt aber über keine Datenstruktur für Sequenzen. Es verfügt nur über die Datenstruktur der Relation. Ein Operator kann seinem nachfolgenden Operator daher nur Relationen, nicht aber Sequenzen übergeben. Aus diesem Grund muss Sequi-Stream-Join die von ihm zu untersuchenden Sequenzen selbst aufbauen. An den nachfolgenden Operator kann er lediglich das Ergebnis der Allquantifizierung, nicht aber die aufgebauten Sequenzen zur Weiterverarbeitung übergeben. Eine mehrstufige Verarbeitung von Sequenzen durch aufeinanderfolgende Stream-Joins ist nicht möglich. SRQL unterliegt diesen Einschränkungen dagegen nicht. Es definiert eine Datenstruktur für Sequenzen. Sequenzen müssen zwar auch erst durch den Operator ψ aufgebaut werden, können dann aber als solche von Operator zu Operator übergeben werden. Dadurch ist es möglich, mehrstufige und aufeinander aufbauende Verarbeitungsschritte durchzuführen und die so modifizierten Sequenzen auszugeben.

Hinsichtlich eines einzelnen Verarbeitungsschrittes stellt SRQL umfassendere Möglichkeiten zur Kombination und Verarbeitung von Sequenzen bereit. Es ermöglicht wertebasierte und positionsbasierte Verarbeitung von Sequenzen. Bei wertebasierter Sequenzverarbeitung werden die Tupel einer bzw. mehrerer Sequenzen durch ein Prädikat miteinander kombiniert, das die Werte der Tupel in Attributen aus A_D miteinander vergleicht. Bei positionsbasierter Verarbeitung werden die Tupel dagegen anhand eines Prädikates kombiniert, das die Positionen der Tupel in ihren Herkunftssequenzen miteinander vergleicht. Da Sequi-Stream-Join ein unärer Operator ist, müssen die Tupel einer oder mehrerer Sequenzen im Vorfeld durch eine Join-Operation kombiniert werden, damit sie der Operator überhaupt erst miteinander vergleichen kann. Die Joinbedingung kann die Tupel nur wertebasiert kombinieren, weswegen Sequi-Stream-Join nur wertebasierte Sequenzvergleiche durchführen kann.

Beim Aufbau einer Sequenz ermittelt der Sequenzoperator ψ für jedes Tupel eine Ordinalzahl. Sie gibt die Position des Tupels innerhalb der Sequenz bezüglich der Sortierung nach A_D an. Jede in der Sequenz vorhandene Belegung von A_D definiert eine Position, so dass es keine leeren bzw. nicht besetzten Positionen gibt. Die Positionen werden nach dem Sortieren beginnend mit dem Wert 1 aufsteigend durchnummeriert. Gibt es in einer Sequenz zu einer Belegung von A_D mehrere Tupel, wird ihnen dieselbe Ordinalzahl zugewiesen. Die Ordinalzahlen werden als Integer-Werte unter dem speziellen Attribut *ordinal* bzw. *ord* abgelegt. Dieses Attribut wird nicht in der Datenbank gespeichert und kann auch nicht in der Ergebnisrelation ausgegeben werden. Es wird beim Aufbau einer Sequenz vom Operator ψ angelegt und existiert so lange wie die zugehörige Sequenz.

In SRQL können die Restriktion und der Join-Operator auf das Positionsattribut zugreifen und Sequenzen positionsbasiert verarbeiten. In der zugrunde liegenden Algebra müssen Join und Left-Join im Gegensatz zur Relationenalgebra von SQL daher als elementare Operatoren behandelt werden, die in ihrem Joinprädikat das Positionsattribut enthalten können. Dadurch ist es z. B. möglich, für die Anwendung von Aggregatfunktionen positionsbasierte, bewegliche Fenster zu definieren (Operator ω) und die in [4] angegebenen Anfragen über den Zusammenhang von Erdbeben und Vulkanausbrüchen zu verarbeiten.

Bei einer zusammengesetzten Sequenz, deren einzelne Sequenzen denselben Verlauf in den Attributen A_D aufweisen, bilden bei Sequi-Stream-Join genau diejenigen Tupel einen Stream, die in SRQL dieselbe Ordinalzahl erhalten. (Eine Relation, die alle Schlusskurse der Dax-Werte des Jahres 2000 enthält, wäre ein Beispiel für eine solche zusammengesetzte Sequenz.) Sequi-Stream-Join ist jedoch nicht in der Lage, die Tupel der Streams über ein spezielles Positionsattribut anzusprechen. Er kann lediglich die Tupel von unmittelbar aufeinanderfolgenden Streams über das Sequenzattribut miteinander vergleichen. Eine Verknüpfung der Tupel zum Aufbau einer neuen Sequenz ist nicht möglich. In SRQL können Tupel dagegen nicht nur miteinander verglichen sondern auch miteinander verknüpft und ausgegeben werden. Die Verknüpfung kann dabei wertebasiert oder mit Hilfe des Attributs *ord* positionsbasiert erfolgen. Die Mächtigkeit von SRQL schließt Sequi-Stream-Join ein. Das folgende Anfrageschema drückt die von Sequi-Stream-Join realisierte Allquantifizierung allgemein in SRQL aus:

```
select IR.A_G∪A_S
from   R_in group by A_G∪A_S sequence by A_D as IR
where  P(IR, shiftall(IR, 1))
having count(*) + 1 = (select count(distinct R.A_D)
                       from   R_in as R
                       where R.A_G = IR.A_G)
```

Es wird vorausgesetzt, dass es bezüglich der Attribute $A_G \cup A_S \cup A_D$ keine Duplikate in der Eingaberelation R_{in} gibt. Die FROM-Klausel der äußeren Anfrage legt fest, wie die Sequenzen aufgebaut werden. Alle Tupel, die in den Attributen $A_G \cup A_S$ übereinstimmen, bilden eine nach A_D sortierte Sequenz. Die WHERE-Klausel realisiert in Verbindung mit dem Operator ShiftAll die Überprüfung der Sequenzbedingungen von Prädikat *P*. ShiftAll(*IR*, 1) erzeugt zu jeder Sequenz Paare von unmittelbar aufeinanderfolgenden Tupeln. Diejenigen Tupel einer Sequenz, die zusammen mit ihrem Folgetupel das Prädikat *P* erfüllen, verbleiben in dem von der WHERE-Klausel erzeugten, temporären Zwischenergebnis *Temp*.

Sequi-Stream-Join realisiert eine Allquantifizierung und gibt für eine Sequenz nur dann ein Ergebnistupel aus, wenn sie die Bedingung *P* für alle Paare unmittelbar aufeinanderfolgender Streams ihrer Gruppe erfüllt. Die HAVING-Klausel realisiert diese Allquantifizierung durch Aggregieren und Zählen. Die Anzahl der Tupel einer Sequenz in *Temp* wird mit der Anzahl der Streams der zugehörigen, durch die Attribute A_G definierten Gruppe verglichen. Dabei wird zu der in *Temp* für die Sequenz ermittelten Anzahl Eins dazuaddiert, da das letzte Tupel kein Folgetupel besitzt und daher *P* nicht erfüllen kann. Hat die Restriktion der WHERE-Klausel mindestens ein Tupel aus der Sequenz entfernt oder besaß die Sequenz von Anfang an weniger Tupel als die Anzahl der zugehörigen Streams, erfüllt sie HAVING-Klausel nicht. Die Anfrage gibt zu jeder Sequenz, die die HAVING-Klausel und damit Allquantifizierung erfüllt, ein Ergebnistupel mit den Attributen $A_G \cup A_S$ aus.

Beispiel 7.2

Um das Anfrageschema zur Beschreibung von Sequi-Stream-Join in SRQL zu verdeutlichen, wird es auf die Anfrage aus Beispiel 6.1 angewandt. Aus der Relation *WP2(Tag, Aktie, Kurs)* sind diejenigen Automobilwerte zu ermitteln, deren Kurse im

Beobachtungszeitraum Montag bis Freitag kontinuierlich gestiegen sind. Der Operator Sequi-Stream-Join wird mit den Attributmengen $A_G = \varnothing$, $A_S = \{Aktie\}$ und $A_D = \{Tag\}$ aufgerufen. Das Prädikat *P* lautet:

P = (ir.Kurs < next.Kurs)

Die Anfrage in SRQL ergibt sich durch Einsetzen der Attributmengen und des Prädikats *P* in das Anfrageschema. Sie fällt in diesem Beispiel etwas einfacher aus, da die Gruppenattribute A_G leer sind:

```
select  IR.Aktie
from    Rin  group by  Aktie  sequence by  Tag  as  IR
where   IR.Kurs < shiftall(IR, 1).Kurs
having count(*) + 1 = (select  count(distinct R.Tag)
                       from    Rin  as  R)
```

IR	Tag	Aktie	Kurs	ord
	Mo	Po	250	1
	Di	Po	240	2
	Mi	Po	245	3
	Do	Po	250	4
	Fr	Po	254	5
	Mo	VW	60	1
	Di	VW	62	2
	Mi	VW	66	3
	Do	VW	68	4
	Fr	VW	70	5
	Mo	DC	100	1
	Di	DC	102	2
	Mi	DC	105	3
	Do	DC	110	4
	Fr	DC	112	5
	Mo	Au	20	1
	Di	Au	26	2
	Do	Au	30	3

Tabelle 7.1: Eingaberelation IR

	Tag_1	$Kurs_1$
	Di	240
	Mi	245
	Do	250
	Fr	254
	NULL	NULL
	Di	62
	Mi	66
	Do	68
	Fr	70
	NULL	NULL
	Di	102
	Mi	105
	Do	110
	Fr	112
	NULL	NULL
	Di	26
	Do	30
	NULL	NULL

Tabelle 7.2: $ShiftAll_1$ (IR)

Tabelle 7.1 zeigt die zusammengesetzte Sequenz $IR = \psi(Aktie, Tag, R_{in})$, die aus der Anwendung des Sequenzoperators ψ auf die Eingaberelation R_{in} hervorgeht. R_{in} wird nach Attribut *Aktie* in Sequenzen gruppiert; innerhalb der Sequenzen wird nach Attribut *Tag* sortiert. Jedes Tupel wird mit dem Attribut *ord* versehen, das seine Position innerhalb der Sequenz angibt.

Tabelle 7.2 enthält die zwei zusätzlichen Spalten, die der Operator ShiftAll $\Lambda_1(IR)$ erzeugt. Die komplette Ergebnisrelation IR_1 von ShiftAll ergibt sich, indem man die Relation IR um diese beiden Spalten ergänzt. Die Anfrage wird in SRQL folgendermaßen ausgewertet:

(1) Übertrage den ShiftAll-Ausdruck in die FROM-Klausel

(2) Werte alle Ausdrücke der FROM-Klausel aus. Die Sequenz $IR = \psi(Aktie, Tag, R_{in})$ wird erzeugt und zur Sequenz $IR_1 = \Lambda_1(IR)$ erweitert.

(3) Werte die WHERE-Klausel aus. Die Restriktion entfernt diejenigen Tupel aus IR_1, die die Sequenzbedingung P nicht erfüllen. Tabelle 7.3 zeigt die Sequenz IR_1 nach Auswertung der WHERE-Klausel.

(4) Werte die HAVING-Klausel aus. Nur zu denjenigen Sequenzen, die P für alle aufeinander folgenden Tage erfüllen, wird ein Ergebnistupel produziert. Es enthält den Namen der zugehörigen Aktie.

IR_1	Tag	Aktie	Kurs	ord	Tag_1	$Kurs_1$
	Di	Po	240	2	Mi	245
	Mi	Po	245	3	Do	250
	Do	Po	250	4	Fr	254
	Mo	VW	60	1	Di	62
	Di	VW	62	2	Mi	66
	Mi	VW	66	3	Do	68
	Do	VW	68	4	Fr	70
	Mo	DC	100	1	Di	102
	Di	DC	102	2	Mi	105
	Mi	DC	105	3	Do	110
	Do	DC	110	4	Fr	112
	Mo	Au	20	1	Di	26
	Di	Au	26	2	Do	30

Tabelle 7.3: Zwischenergebnis nach Auswertung der WHERE-Klausel

Das Ergebnis der Anfrage sind die Aktien VW und DaimlerChrysler. Ihre Sequenzen enthalten nach Anwendung der WHERE-Klausel drei Tupel. Sie erfüllen also für jeden der vier Streams *Mo, Di, Mi, Do* das Prädikat P. Bei Porsche erfüllt das erste Tupel der Sequenz das Prädikat P nicht. Sie enthält beim Auswerten der HAVING-Klausel nur zwei Tupel und wird nicht ausgegeben. Die Sequenz von Audi ist von Anfang an unvollständig und erreicht die erforderliche Anzahl an Tupeln ebenfalls nicht.

Möchte man die Beschreibung von Sequi-Stream-Join in SRQL auf allgemeinstem Niveau nachvollziehen, ist das Beispiel um ein Attribut für die Menge A_G zu erweitern. Fügt man zur Eingaberelation R_{in} das Attribut *Woche* hinzu, kann man wochenweise nach monoton steigenden Kursen suchen. Mit der Attributmenge $A_G = \{Woche\}$ ergibt sich die Anfrage:

```
select IR.Woche, IR.Aktie
from   R_in group by Woche, Aktie sequence by Tag as IR
where  IR.Kurs < shiftall(IR, 1).Kurs
having count(*) + 1 = (select count(distinct R.Tag)
                       from   R_in as R)
                       where  R.Woche = IR.Woche)
```

7.2. Verwendung von Sequi-Stream-Join in SRQL

SRQL stellt einen mächtigen Ansatz zur Verarbeitung von Sequenzen dar. Ihre sprachliche Ausdruckskraft schließt den Operator Sequi -Stream-Join ein. Die von ihm realisierte Allquantifizierung in Sequenzen kann durch ein allgemeines Anfrageschema in SRQL formuliert werden. Es enthält den Aufbau der Sequenzen und umschreibt die Allquantifizierung durch Aggregieren und Zählen.

Equi-Stream-Join bearbeitet Anfragen mit Allquantifizierung effizienter als alle anderen bislang bekannten Ansätze. Der Algorithmus von Sequi -Stream-Join geht in der gleichen Weise vor, um Allquantifizierung in Sequenzen zu bearbeiten. Er führt vergleichbare Allquantifizierungen mit derselben Effizienz aus und unterscheidet sich im Aufwand nicht von Equi -Stream-Join. In einem relation ales Datenbanksystem, das zur Verarbeitung von Sequenzen erweitert wurde und über die Anfragesprache SRQL verfügt, kann Sequi -Stream-Join implementiert werden, um Allquantifizierung in Sequenzen effizient zu verarbeiten. Dem Benutzer des Datenbanksystems b leibt der Operator verborgen. Es ist, wie im Falle von Equi-Stream-Join, die Aufgabe des Parsers bzw. des Optimierers, Anfragen zu erkennen, die eine Allquantifizierung in Sequenzen beschreiben. Hat er eine solche Anfrage erkannt, ermittelt er die Aufrufpa rameter R_{in}, A_G, A_S, A_D, und P für den Operator und generiert einen Ausführungsplan für die Anfrage, die den Operator Sequi-Stream-Join enthält.

Die Schwierigkeiten, die mit der Erkennung von Allquantifizierung in Anfragen verbunden sind, wurden bereits in Kapitel 5 erwähnt. Sie treten beim Parsen von SRQL-Anfragen in gleicher Weise auf, da die Anfragesprache SRQL auf SQL basiert und ebenfalls kein Schlüsselwort für den Allquantor enthält. Um diese Schwierigkeiten zu vermeiden, sollte eine Implementierung von Sequi -Stream-Join stets mit einer Erweiterung der Anfragesprache SRQL um das Schlüsselwort FOR ALL einhergehen.

Beispiel 7.2

Um abschließend die Einsatzgebiete von equivalence und sequence Stream -Join einander gegenüberzustellen, wird eine SRQL -Anfrage angegeben, die sowohl eine „gewöhnliche“ Allquantifizierung als auch eine Allquantifizierung in Sequenzen enthält. Gegeben sei das Schema der Relation *WP*:

WP (Branche, Index, Aktie, Tag, Kurs)

Sie ent hält Wertpapiere von verschiedenen Branchen, z. B. Einzelhandel, Telekom - munikation, Information Technology, Automobilindustrie, etc. Zu jedem Börsentag werden die Schlusskurse der Aktien in der Relation notiert. Außerdem gibt es zu jeder Branche einen Branchenindex, dessen Tageswert im Attribut *Index* abgelegt wird.

Durch eine Anfrage sollen diejenigen Branchen ermittelt werden, deren Aktien im Beobachtungszeitraum der zehn Börsentagen 120 bis 129 alle dem Index gefolgt sind. Ein benutzerdefiniertes Prädikat *follows(r1, r2, w1, w2)* ermittelt für die Kurse von zwei aufeinanderfolgenden Tagen, ob die Aktie *w* dem Verlauf des Referenzkurses *r* innerhalb einer vorgegebenen Bandbreite folgt. Die Anfrage wird der Übersichtlichkeit wegen in eine Unteranfrage *Q1* und eine Hauptanfrage geteilt:

```
select  Q1.Aktie
from    (select *
         from   WP
         where Tag ≥ 120  and  Tag ≤ 129)
        group by  Aktie  sequence by  Tag  as  Q1
where  follows(Q1.Index, shiftall(Q1, 1).Index, Q1.Kurs, shiftall(Q1, 1). Kurs)
having count(*) + 1 = 10
```

Die Unteranfrage ist in SRQL entsprechend dem Anfrageschema für Sequi -Stream-Join formuliert. Sie führt die Allquantifizierung in den Sequenzen der Aktienkurse durch. Ihre Ausgaberelation enthält diejenigen Aktien, die im Beobachtungszeitraum ihrem Branchenindex gefolgt sind. In der HAVING-Klausel ist keine weitere Unteranfrage erforderlich, um die Tage des Beobachtungszeitraumes zu zählen. Er ist für alle Aktien der gleiche (A_G ist leer) und umfasst zehn Tage. Die Hauptanfrage lautet:

```
select  WP.Branche
from    WP  as  IR
where  for all (select distinct WP1.Aktie
                from   WP  as  WP1
                where WP1.Branche = WP.Branche)
               (WP1.Aktie  in  Q1)
```

Die Anfrage ermittelt die Aktien zu jeder Branche und prüft, ob sie alle in der von der Unteranfrage erzeugten Relation *Q1* enthalte n sind. Ist dies der Fall, dann sind alle Aktien dem Branchenindex gefolgt und die Branche wird ausgegeben. Die Allquanti - fizierung der Hauptanfrage hat mit Sequenzen nichts zu tun und kann von Equi -Stream-Join bearbeitet werden. Die Allquantifizierung der Unteranfrage kann dagegen nur von Sequi-Stream-Join bearbeitet werden. Der kombinierte Einsatz beider Stream -Join Operatoren garantiert eine effiziente Ausführung dieser Anfrage.

Schlusswort

Mit equivalence und sequence Stream -Join stehen zwei neue Algori thmen zur Verfügung, die es relationalen Datenbanksystemen ermöglichen, Allquantifizierung in Anfragen effizienter als bisher auszuführen. Datenbanksysteme mit der Anfragesprache

SQL können jede Allquantifizierung optimal ausführen, wenn die Operatoren Incremental Hash -Division und Equi -Stream-Join implementiert sind sowie die Anfragesprache um ein Schlüsselwort für den Allquantor erweitert wird. Die Operatoren selbst bleiben dem Benutzer verborgen . Sie werden vom Parser bzw. vom Optimierer in die Ausführungspläne der Anfragen eingebaut.

Relationale Datenbanksysteme, die über die Anfragesprache SRQL zur Verarbeitung von Sequenzen verfügen, werden durch den Operator Sequi -Stream-Join in die Lage versetzt, Allquantifizierung in Sequenzen effizient ausführen. Auch hier bleibt der Operator selbst dem Benutzer verborgen.

8. Anhang

Die folgende Tabelle enthält den Dividend *S* zu Beispiel 3.2. Er besteht aus 115 Tupeln. Die von dem Join zu einem Tupel der Relation *R = Abg2* erzeugten Tupel sind in *S* durch die horizontal verlaufenden Linien voneinander getrennt.

S	Sname	Studium	Pnr
	Tina	I	1
	Tina	I	2
	Tina	I	3
	Tina	I	4
	Tina	M	5
	Tina	M	6
	Tina	M	7
	Tina	B	9
	Mark	I	2
	Mark	M	5
	Mark	M	6
	Mark	M	7
	Mark	B	8
	Mark	B	9
	Mark	B	10
	Mark	I	3
	Mark	M	5
	Mark	M	6
	Mark	M	7
	Mark	B	8
	Mark	B	9
	Mark	B	10

S	Sname	Studium	Pnr
	Max	I	1
	Max	I	2
	Max	I	3
	Max	I	4
	Max	M	5
	Max	B	8
	Max	B	9
	Max	B	10
	Alex	M	5
	Alex	M	6
	Alex	M	7
	Alex	B	8
	Alex	B	9
	Alex	B	10
	Alex	I	1
	Alex	M	5
	Alex	M	6
	Alex	M	7
	Alex	B	8
	Alex	B	9
	Alex	B	10

S	Sname	Studium	Pnr
	Tina	I	1
	Tina	I	2
	Tina	I	3
	Tina	I	4
	Tina	M	5
	Tina	M	6
	Tina	M	7
	Tina	B	8
	Alex	I	3
	Alex	M	5
	Alex	M	6
	Alex	M	7
	Alex	B	8
	Alex	B	9
	Alex	B	10
	Mark	I	1
	Mark	M	5
	Mark	M	6
	Mark	M	7
	Mark	B	8
	Mark	B	9
	Mark	B	10
	Max	I	1
	Max	I	2
	Max	I	3
	Max	I	4
	Max	M	6
	Max	B	8
	Max	B	9
	Max	B	10
	Tina	I	1
	Tina	I	2
	Tina	I	3
	Tina	I	4
	Tina	M	5
	Tina	M	6
	Tina	M	7

S	Sname	Studium	Pnr
	Mark	I	4
	Mark	M	5
	Mark	M	6
	Mark	M	7
	Mark	B	8
	Mark	B	9
	Mark	B	10
	Alex	I	2
	Alex	M	5
	Alex	M	6
	Alex	M	7
	Alex	B	8
	Alex	B	9
	Alex	B	10
	Tina	I	1
	Tina	I	2
	Tina	I	3
	Tina	I	4
	Tina	M	5
	Tina	M	6
	Tina	M	7
	Tina	B	10
	Alex	M	5
	Alex	M	6
	Alex	M	7
	Alex	B	8
	Alex	B	9
	Alex	B	10
	Mark	I	1
	Mark	M	5
	Mark	M	6
	Mark	M	7
	Mark	B	8
	Mark	B	9
	Mark	B	10

Tabelle 8.1: Dividend S

Literaturverzeichnis

Stream-Join:

[1] C. Nippl, R. Rantzau and B. Mitschang. StreamJoin: A generic database app roach to support the class of stream -oriented applications. In *Proceedings of the IDEAS Conference,* pages 83-91, Yokohama, Japan, September 2000.

[2] R. Rantzau, B. Mitschang and L. Shapiro. Sequence analysis with the Stream -Join database operator: Definition s, applications and query optimization. Research Paper. *Universität Stuttgart, Fakultät Informatik, Abteilung Anwendersoftware,* 2001.

[3] Clara Nippl. Providing efficient, extensible and adaptive intra -query parallelism for advanced applications. Doktorarbeit. *Technische Universität München, Institut für Informatik,* 2000.

Sequenzverarbeitung:

[4] R. Ramakrishnan, D. Donjerkovic, A. Ranganathan, K. Beyer and M. Krishnaprasad. SRQL: Sorted Relational Query Language. In *IEEE Proceedings of the 10th SSDBM Conference,* pages 84-95, Capri, Italy, 1998.

[5] P. Seshadri, M. Livny and R. Ramakrishnan. The design and implementation of a sequence database system. In *Proceedings of the 22 nd VLDB Conference,* pages 99-110, Bombay, India, 1996.

[6] P. Seshadri, M. Livny and R. Ramakrishnan . Sequence query processing. In *ACM Proceedings of the SIGMOD Conference,* pages 430-441, Minneapolis, USA, May 1994.

[7] Praveen Seshadri. Management of sequence data. Ph. D. Thesis. *University of Wisconsin, Dept. of Computer Science,* 1996.

Allquantifizierung:

[8] G. Graefe and R. Cole. Fast algorithms for universal quantification in large databases. In *ACM Proceedings of the TODS Conference,* Vol. 20, No. 2, pages 187-236, June 1995.

[9] J. Claussen, A. Kemper, G. Moerkotte and K. Peithner. Optimizing queries with universal quantification in object -oriented and object -relational databases. In *Proceedings of the 23rd VLDB Conference,* pages 286-295, Athens, Greece, 1997.

[10] P. Hsu and D. Parker. Improving SQL with generalized quantifiers. In *IEEE Proceedings of the 11th ICDE Conference,* pages 298-305, Taipeh, Taiwan, 1995.

[11] P. Gulutzan and T. Pelzer. SQL-99 Complete, Really. R&D Books, 1999.

[12] Peter Lesky sen. Mathematische Grundlagen, zur Vorlesung im WS 91/92. *Universität Stuttgart, Mathematisches Institut A,* 1991.

Datenbank-Algorithmen:

[13] H. Garcia Molina, J. Ullman and J. Widom. Database System Implementation. *Stanford University, Department of Comuter Science*, Prentice Hall, 2000.

Erklärung zur Diplomarbeit

Hiermit erkläre ich, dass ich diese Diplomarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Stuttgart, den 23. Juli 2001

(Bernd Watzal)