

**Universität Stuttgart**  
**Fakultät Informatik**

Studiengang: Informatik

Prüfer: Prof. Bernhard Mitschang

Betreuer: Ralf Rantza

Beginn am: 18.07.02

Beendet am: 17.01.03

CR-Klassifikation: H 2.4

Studienarbeit Nr.: 1866

**Entwurf und Integration von  
Set Containment Division Operatoren  
in einen Anfrageprozessor**

Rudi Husser

Institut für Parallele und Verteilte Systeme (IPVS)

Abteilung Anwendersoftware (AS)

# Inhaltsverzeichnis

<b>1. Einleitung</b> .....	<b>3</b>
1.1 Gliederung der Arbeit .....	5
<b>2. Die relationale Division</b> .....	<b>7</b>
2.1 Überblick und Klassifikation der Algorithmen.....	7
2.1.1 Beschreibung der Eingabedaten .....	8
2.1.2 Formulierung der universellen Quantifikation in SQL.....	9
2.1.3 Klassifikation der Algorithmen .....	11
2.2 Merge-Sort-Division .....	12
2.3 Hash-Division .....	13
<b>3. Set Containment Join</b> .....	<b>15</b>
3.1 Algorithmen.....	15
3.1.1 Signaturen und Partitionierung.....	16
3.1.2 Partitioning Set Join (PSJ).....	18
3.1.3 Adaptive Pick-and-Sweep-Join (APSJ) .....	19
<b>4. Set Containment Division</b> .....	<b>21</b>
<b>5. Analyse der Algorithmen</b> .....	<b>23</b>
5.1 Analyse der Algorithmen für die relationale Division .....	23
5.1.1 Merge-Sort-Division .....	23
5.1.2 Hash-Division.....	23
5.2 Analyse der Algorithmen für Set Containment Join .....	24
5.2.1 Bedeutung der Formeln.....	25
<b>6. Implementierung der Algorithmen</b> .....	<b>28</b>
6.1 Die Java-Klassenbibliothek XXL.....	28
6.1.1 Das Cursor-Konzept in XXL.....	29
6.2 Die implementierten Klassen .....	30
6.2.1 Die Klasse MyFileMetaDataCursor.....	30
6.2.2 Die Klasse HashDivision.....	31
6.2.3 Die Klasse SetContainmentDivision.....	33
6.2.4 Die Klasse SetContainmentJoin .....	34
6.2.5 Die Klassen Unnest, GroupBasedNest und HashBasedNest.....	36
6.2.6 Die Klasse MyDataGenerator.....	37
<b>7. Experimente</b> .....	<b>40</b>
7.1 Experimentelle Analyse von HashDivision .....	40
7.2 Experimentelle Analyse von SetContainmentDivision .....	42
7.3 Experimentelle Analyse von SetContainmentJoin .....	45
<b>8. Zusammenfassung</b> .....	<b>50</b>
<b>Literaturverzeichnis</b> .....	<b>51</b>

# 1. Einleitung

Eine wichtige Operation in Datenbanksystemen ist die *Division* bzw. die *All-Quantifikation*. Die All-Quantifikation kommt in vielen praktischen Anwendungen wie Data Mining und OLAP zum Einsatz und ist nützlich bei Anfragen, die das Wort „jede“ oder „alle“ enthalten. Anfragen mit All-Quantifikation sind z. B.:

„Welche Bewerber besitzen **alle** erforderlichen Fähigkeiten für den neu zu besetzenden Job?“

oder

„Finde alle Transaktionen, die **alle** Elemente einer gegebenen Produktmenge enthalten.“

Wird eine Relation  $R$  durch eine Relation  $S$  dividiert, ergibt die Ergebnisrelation alle Tupel, die mit der *gesamten* Relation  $S$  in einer bestimmten Beziehung stehen. In relationalen Datenbanken wird die All-Quantifikation durch den Divisions-Operator ( $\div$ ) implementiert.

Bei der *relationalen Division* wird geprüft, ob alle Elemente einer gegebenen Menge eine *gegebene Bedingung* erfüllen. In vielen Anwendungen ist diese Bedingung ein Test, ob eine (Teil-)Menge in einer anderen Menge enthalten ist, d. h. das Quantifikations-Problem wird zu einem „*Set-Containment*“-Problem. So kann z. B. die obige Anfrage über die Bewerber folgendermaßen umformuliert werden:

„Finde die Bewerber, deren Menge von Fähigkeiten, die gegebene Menge der gesuchten Fähigkeiten enthält.“

Allerdings basiert die Division auf dem relationalen Modell. Bei diesem Modell geht man davon aus, dass alle Relationen in erster Normalform (1NF) vorliegen, d.h. alle Attribute atomare Wertebereiche besitzen und z.B. Mengen als Wertebereiche nicht erlaubt sind.

Zur Lösung des Set-Containment-Problems bei Tabellen, die nicht in 1NF sind und mehrwertige Attribute besitzen, müssen also andere Algorithmen bzw. Operatoren verwendet werden. Diese Algorithmen heißen *Set-Containment-Join-Algorithmen* (SCJ) und bilden einen Join zwischen mehrwertigen Attributen zweier Relationen, wobei die Join-Bedingung durch den Teilmengen-Operator ( $\subseteq$ ) festgelegt ist.

Der Divisions-Operator ist also nah verwandt mit SCJ und kann als Sub-Problem des Set-Containment-Problems angesehen werden, wobei die Mengen in 1NF gespeichert sind. Der Hauptunterschied zwischen beiden Operatoren liegt in der Verarbeitung des Divisors: bei der relationalen Division darf der Divisor nur aus einer einzelnen Gruppe bestehen, während bei SCJ der „Divisor“ mehrere Mengen bzw. Gruppen enthalten kann.

Um das Set-Containment-Problem auch mit Tabellen in 1NF lösen zu können, wurde eine Erweiterung der relationalen Division vorgeschlagen: die *Set-Containment-Division* (SCD). Bei SCD darf der Divisor also wie bei SCJ aus mehreren Gruppen bestehen, die Tabellen sind allerdings - wie bei der relationalen Division - in 1NF und

haben keine mehrwertigen Attribute wie sie beim SCJ erlaubt sind. Der SCD-Operator ist äquivalent zum SCJ-Operator und liefert als Ergebnis die gleiche Tabelle zurück wie SCJ, allerdings ohne die Verbundattribute.

Um die einzelnen Konzepte der relationalen Division, SCD und SCJ zu beschreiben, wird in dieser Studienarbeit das durchgehend verwendete Szenario mit Eingaberelationen aus Abb.1 benutzt, das für jeden der drei Operatoren entsprechend angepasst wird.

<b>Prüfungen /R(a, b)</b>	$\div$	<b>Vorlesung /S(c)</b>	$=$	<b>Resultat /T(a)</b>																										
<table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr><th><i>student-id</i></th><th><i>course-id</i></th></tr> </thead> <tbody> <tr><td>Alice</td><td>Compilers</td></tr> <tr><td>Alice</td><td>Theory</td></tr> <tr><td>Bob</td><td>Compilers</td></tr> <tr><td>Bob</td><td>Databases</td></tr> <tr><td>Bob</td><td>Graphics</td></tr> <tr><td>Bob</td><td>Theory</td></tr> <tr><td>Chris</td><td>Compilers</td></tr> <tr><td>Chris</td><td>Graphics</td></tr> <tr><td>Chris</td><td>Theory</td></tr> </tbody> </table>	<i>student-id</i>	<i>course-id</i>	Alice	Compilers	Alice	Theory	Bob	Compilers	Bob	Databases	Bob	Graphics	Bob	Theory	Chris	Compilers	Chris	Graphics	Chris	Theory		<table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr><th><i>course-id</i></th></tr> </thead> <tbody> <tr><td>Compilers</td></tr> <tr><td>Databases</td></tr> <tr><td>Theory</td></tr> </tbody> </table>	<i>course-id</i>	Compilers	Databases	Theory		<table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr><th><i>student-id</i></th></tr> </thead> <tbody> <tr><td>Bob</td></tr> </tbody> </table>	<i>student-id</i>	Bob
<i>student-id</i>	<i>course-id</i>																													
Alice	Compilers																													
Alice	Theory																													
Bob	Compilers																													
Bob	Databases																													
Bob	Graphics																													
Bob	Theory																													
Chris	Compilers																													
Chris	Graphics																													
Chris	Theory																													
<i>course-id</i>																														
Compilers																														
Databases																														
Theory																														
<i>student-id</i>																														
Bob																														

**Abbildung 1a) :** relationale Division, Tabellen in 1NF

<b>Prüfungen /R(a, b)</b>	<b>Vorlesung /S(c, d)</b>														
<table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr><th><i>student-id</i></th><th><i>courses</i></th></tr> </thead> <tbody> <tr><td>Alice</td><td>{Compilers, Theory}</td></tr> <tr><td>Bob</td><td>{Compilers, Databases, Graphics, Theory}</td></tr> <tr><td>Chris</td><td>{Compilers, Graphics, Theory}</td></tr> </tbody> </table>	<i>student-id</i>	<i>courses</i>	Alice	{Compilers, Theory}	Bob	{Compilers, Databases, Graphics, Theory}	Chris	{Compilers, Graphics, Theory}	<table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr><th><i>courses</i></th><th><i>program</i></th></tr> </thead> <tbody> <tr><td>{Compilers, Databases, Theory}</td><td>Systems</td></tr> <tr><td>{Compilers, Graphics}</td><td>Applications</td></tr> </tbody> </table>	<i>courses</i>	<i>program</i>	{Compilers, Databases, Theory}	Systems	{Compilers, Graphics}	Applications
<i>student-id</i>	<i>courses</i>														
Alice	{Compilers, Theory}														
Bob	{Compilers, Databases, Graphics, Theory}														
Chris	{Compilers, Graphics, Theory}														
<i>courses</i>	<i>program</i>														
{Compilers, Databases, Theory}	Systems														
{Compilers, Graphics}	Applications														

**Vorlesung courses courses Prüfungen /T(a, b, c, d)**

<i>student_id</i>	<i>courses</i>	<i>courses</i>	
Bob		{Compilers, Databases, Theory}	Systems
	{Compilers, Databases, Graphics, Theory}	{Compilers, Graphics}	
Chris	{Compilers, Graphics, Theory}	{Compilers, Graphics}	

Set-                      -Join.

**Prüfungen /R(a, b)**

<i>student-id</i>	<i>course-id</i>
Alice	Compilers
Alice	Theory
Bob	Compilers
Bob	Databases
Bob	Graphics
Bob	Theory
Chris	Compilers
Chris	Graphics
Chris	Theory

**Vorlesung /S(c, d)**

<i>student-id</i>	<i>course-id</i>
Compilers	Systems
Databases	Systems
Theory	Systems
Compilers	Applications
Graphics	Applications

**Prüfungen  $\div$  courses\_id course\_id Vorlesung /T(a, d)**

<i>student-id</i>	<i>program</i>
Bob	Systems
Bob	Systems
Chris	Applications

**Abbildung 1c) : Set Containment Division, Tabellen in 1NF**

In Abb. 2 werden die Charakteristika der drei Operatoren noch einmal zusammengefasst [1].

	<b>Set Containment Join</b>	<b>Division</b>	<b>Set Containment Division</b>
Operator und Eingaberelationen	$S(c,d) \bowtie_{c \rightarrow b} R(a,b)$	$R(a,b) \div S(c)$	$R(a,b) \div_{b \rightarrow c} S(c,d)$
linker Input/Dividend	mehrere Mengen	mehrere Gruppen	mehrere Gruppen
rechter Input/Divisor	mehrere Mengen	eine Gruppe	mehrere Gruppen
Resultat/Quotient	$T(a, b, c, d)$	$T(a)$	$T(a, d)$
Daten-Layout	nicht in 1NF	1NF	1NF

**Abbildung 2: Zusammenfassung der Operator-Charakteristika**

Das Ziel dieser Arbeit ist ein Vergleich der Algorithmen für die relationale Division, SCD und SCJ.

## 1.1 Gliederung der Arbeit

In Kapitel 2 wird zunächst die klassische relationale Division genauer beschrieben. Von den vielen in der Literatur vorgeschlagenen Algorithmen zur effizienten Implementierung der relationalen Division wird der Hash-Division-Algorithmus sowie die Merge-Sort-Division genauer diskutiert. Danach wird in Kapitel 3 der Set-Containment-Join mit den entsprechenden Algorithmen, und in Kapitel 4 die Set-Containment-Division mit seinen konkreten Algorithmen vorgestellt. Im 5. Kapitel werden anschließend die vorher beschriebenen Algorithmen analysiert und miteinander verglichen. Im nächsten Kapitel wird die Implementierung der Algorithmen die im Rahmen dieser Studienarbeit realisiert wurden, sowie die Hilfsmittel, die da-

bei zum Einsatz kamen, beschrieben. Danach werden im 7. Kapitel diese Algorithmen Leistungsmessungen unterzogen. Schließlich erfolgt im letzten Kapitel eine Zusammenfassung.

## 2. Die relationale Division

Die All-Quantifikation kommt in vielen praktischen Anwendungen zum Einsatz und wird durch den Divisions-Operator der relationalen Algebra implementiert.

Um den Divisions-Operator zu beschreiben, wird das einfache Beispiel aus Abb. 3 benutzt, welches Daten einer Abteilung in einer Universität darstellt:

**Prüfungen (Dividend)  $\div$  Vorlesung (Divisor) = Resultat (Quotient)**

<i>student-id</i>	<i>course-id</i>	<i>course-id</i>	<i>student-id</i>
Alice	Compilers	Compilers	Bob
Alice	Theory	Databases	
Bob	Compilers	Theory	
Bob	Databases		
Bob	Graphics		
Bob	Theory		
Chris	Compilers		
Chris	Graphics		
Chris	Theory		

**Abbildung 3:** relationale Division, Tabellen in 1NF

Eine Zeile der Tabelle *Vorlesung* enthält eine Vorlesung, die von einer bestimmten Fakultät momentan angeboten wird. Eine Zeile der Tabelle *Prüfungen* gibt an, dass ein Student eine Prüfung in einer bestimmten Vorlesung erfolgreich abgelegt hat. Die folgende Anfrage kann mit Hilfe des Divisions-Operators dargestellt werden:

„Welche Studenten haben die Prüfungen in **allen** Vorlesungen bestanden, die von der Fakultät angeboten werden?“

Wie man der Tabelle *Resultat* entnehmen kann, hat nur *Bob* alle Prüfungen bestanden. Zusätzlich hat *Bob* auch noch die Prüfung *Graphics* bestanden, dies hat allerdings auf das Ergebnis keinen Einfluss. Weder *Alice* noch *Chris* haben die Prüfung *Databases* erfolgreich abgelegt, deshalb sind sie auch nicht im Ergebnis enthalten.

Der Divisions-Operator benötigt zwei Tabellen als Eingabe, den *Dividenden* und den *Divisor*, und erzeugt eine Tabelle als Ausgabe, den *Quotienten*. In Abb. 3 haben *Divisor* und *Quotient* jeweils nur ein Attribut. Im Allgemeinen können sie aber eine beliebige Anzahl von Attributen besitzen. Die Attribute des Dividenden ergeben sich aber durch die disjunkte Vereinigung der Attribute des Divisors und des Quotienten. Die Attribute des Divisors werden *Divisorattribute* genannt, die Attribute des Dividenden, die nicht zum Divisor gehören, bezeichnet man als *Quotientenattribute*. Die Werte der Quotientenattribute im Dividenden ist die Menge der *Quotientenkandidaten* bzw. *Quotientengruppen*.

### 2.1 Überblick und Klassifikation der Algorithmen

Ein Anfrageprozessor eines Datenbanksystems stellt typischerweise mehrere Algorithmen für den gleichen logischen Operator zur Verfügung. Ein Optimierer muss

dann einen dieser Algorithmen auswählen, um die gegebenen Daten zu verarbeiten. Kennt der Optimierer die Struktur der Eingabedaten für den Operator, kann er den Algorithmus auswählen, der diese Struktur am besten ausnützt bzw. keine Restrukturierung der Eingabedaten durchführen muss, bevor er die Ausgabe erzeugt. Deshalb ist ein Überblick wichtig, der angibt, welche Algorithmen für welche Eingabedaten geeignet sind.

## 2.1.1 Beschreibung der Eingabedaten

In dieser Studienarbeit basiert die Klassifikation der Divisionsalgorithmen darauf, ob gewisse Attribute des Dividenden bzw. Divisors gruppiert oder gar sortiert sind. Eine Gruppierung tritt in der Anfrageverarbeitung häufig auf, da viele Datenbankoperatoren gruppierte oder sortierte Eingabedaten benötigen (z.B. Merge-Join) oder solche Ausgabedaten erzeugen (z.B. Index-Scan).

Es ist zu beachten, dass die Sortierung eine spezielle Gruppierungsoperation ist. So erfordert eine Gruppierung beispielsweise nur, dass Studenten, die eine Prüfung in der gleichen Vorlesung bestanden haben, hintereinander (in irgendeiner Reihenfolge) gespeichert werden, während die Sortierung zusätzlich noch eine bestimmte Reihenfolge (auf- oder absteigend) in jeder Gruppe vorschreibt.

Um eine Klassifikation durchzuführen, kann man bei der relationalen Division die Attribute des Dividenden in zwei Mengen  $D$  und  $Q$  aufteilen. Dabei entspricht  $D$  den Attributen des Divisors,  $Q$  den Attributen des Quotienten.

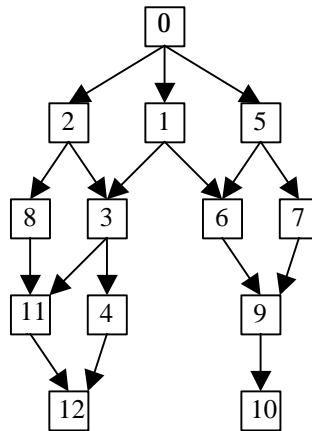
Class	Dividend		Divisor	Beschreibung der Gruppierung
	Q	D		
0	N	N	N	
1	N	N	G	
2	N	G	N	
3	N	$G_1$	$G_2$	Beliebig geordnete Gruppen in D und Divisor
4	N	$G_1$	$G_1$	Gleiche Ordnung der Gruppen in D und Divisor
5	G	N	N	
6	G	N	G	
7	$G_1$	$G_2$	N	Q Hauptordnung, D Nebenordnung
8	$G_2$	$G_1$	N	D Hauptordnung, Q Nebenordnung
9	$G_1$	$G_2$	$G_3$	Q Hauptordnung, D Nebenordnung, zufällige Ordnung in D und Divisor
10	$G_1$	$G_2$	$G_2$	Q Hauptordnung, D Nebenordnung, gleiche Ordnung in D und Divisor
11	$G_2$	$G_1$	$G_3$	D Hauptordnung, Q Nebenordnung, zufällige Ordnung in D und Divisor
12	$G_2$	$G_1$	$G_1$	D Hauptordnung, Q Nebenordnung, gleiche Ordnung in D und Divisor

**Abbildung 4:** Klassifikation von Dividend und Divisor. Attribute sind entweder gruppiert ( $G$ ) oder nicht gruppiert ( $N$ ). Es wird der gleiche (ein anderer) Index für  $G$  benutzt, wenn  $D$  und der Divisor die gleiche (eine andere) Ordnung der Gruppen haben. Ist der Dividend gruppiert in  $Q$  und  $D$ , bedeutet  $G_1$  ( $G_2$ ), dass die Attribute zuerst nach dem ersten (zweiten) Attribut gruppiert sind.

Abb. 4 zeigt alle möglichen Klassen von Eingabedaten [1]. In Klasse 10 beispielsweise ist der Dividend zuerst nach den Quotientenattributen  $Q$  gruppiert (angegeben durch  $G_1$ ), und jede Gruppe ist gruppiert nach  $D$  ( $G_2$ ). Der Divisor ist gruppiert in derselben Ordnung wie der Dividend ( $G_2$ ).

Kann ein Algorithmus Daten einer speziellen Klasse bearbeiten, wäre es nützlich zu wissen, welche zusätzlichen Klassen durch den Algorithmus abgedeckt werden. Dies

kann z. B. durch einen gerichteten azyklischen Graphen wie in Abb. 5 dargestellt werden.



**Abbildung 5:** Ein gerichteter azyklischer Graph, der die Klassifikation der Eingabedaten wie sie in Abb. 4 beschrieben ist, darstellt.

Der Wurzelknoten des Graphen ist die Klasse 0, sie benötigt keine Gruppierung von  $D$ ,  $Q$  oder dem Divisor. Jeder Algorithmus der Daten der Klasse 0 bearbeiten kann, kann Daten jeder anderen Klasse bearbeiten [1]. Ein Algorithmus der mit Daten der Klasse 3 arbeitet, kann also auch mit Daten der Klassen 4, 11 und 12 arbeiten.

### 2.1.2 Formulierung der universellen Quantifikation in SQL

Bei dem am häufigsten benutzten Ansatz zur Formulierung der universellen Quantifikation bzw. relationalen Division kommen zwei „NOT EXISTS“ – Klauseln zum Einsatz:

```

SELECT DISTINCT student_id FROM Prüfungen AS p1
WHERE NOT EXISTS (
  SELECT * FROM Vorlesung AS v
  WHERE NOT EXISTS (
    SELECT * FROM Prüfungen AS p2
    WHERE p2.student_id = p1.student_id
    AND p2.course_id = v.course_id ))).
  
```

Diese Anfrage fragt für jeden Student, ob es keine Vorlesung gibt, in dem der Student nicht erfolgreich geprüft wurde.

Diese Anfrage ist nicht nur schwer verständlich, sondern führt auch zu einer sehr schlechten Leistung [2]. Um diese SQL-Anfrage zu bearbeiten, müssen drei Schleifen durchlaufen werden.

Bei einer anderen Möglichkeit zur Formulierung der universellen Quantifikation wird die Aggregation benutzt:

```
SELECT student_id FROM Prüfungen
GROUP BY student_id
HAVING COUNT (DISTINCT course_id) = (SELECT COUNT (DISTINCT course_id)
FROM Vorlesung).
```

Bei dieser Anfrage werden alle Studenten gesucht, die genau so viele Prüfungen abgelegt haben, wie Vorlesungen angeboten werden. Die DISTINCT-Klauseln sind notwendig, um eventuell vorhandene Duplikate zu eliminieren. Alle Anfragen mit universeller Quantifikation können ersetzt werden durch Anfragen bei denen Aggregation zum Einsatz kommt [2].

Diese zwei Ansätze führen zu *skalaren* und *aggregierenden* Algorithmen, die zur Realisierung der relationalen Division eingesetzt werden. Die skalaren Algorithmen prüfen bestimmte Attributwerte (*D*) von Zeilen des Dividenden und der Tabelle des Divisors auf Übereinstimmung, während die Klasse der aggregierenden Algorithmen Zähler benutzen, um die Anzahl der Zeilen in einer Quotientengruppe des Dividenden mit der Anzahl der Zeilen des Divisors vergleichen zu können. Algorithmen, die Aggregation einsetzen, liefern allerdings nur das gleiche Ergebnis wie die skalaren Algorithmen, wenn zwei Bedingungen erfüllt sind. Erstens dürfen in den Tabellen keine Duplikate vorkommen. Deshalb werden in der zweiten SQL-Formulierung, vor der Durchführung der Division mittels DISTINCT eventuell vorhandene Duplikate eliminiert. Zweitens muss in dieser Klasse die referenzielle Integrität zwischen Dividend und Divisor sichergestellt sein, d.h. jeder Wert, der den Zähler einer Quotientengruppe inkrementiert, muss im Dividenden enthalten sein. Eine Möglichkeit, um dies zu gewährleisten, ist die Durchführung eines Semi-Join zwischen Dividend und Divisor. Dabei werden alle Zeilen des Dividenden zurückgeliefert, deren *D*-Werte im Divisor enthalten sind (siehe Abb. 6).

<b>Prüfungen (Dividend)</b>	<b>Vorlesung (Divisor) = Prüfungen (Dividend)</b>																																									
<table border="1" style="width: 100%; border-collapse: collapse;"><thead><tr><th style="border: none;"><i>student-id</i></th><th style="border: none;"><i>course-id</i></th></tr></thead><tbody><tr><td>Chris</td><td>Compilers</td></tr><tr><td>Chris</td><td>Graphics</td></tr><tr><td>Chris</td><td>Theory</td></tr><tr><td>Alice</td><td>Compilers</td></tr><tr><td>Alice</td><td>Theory</td></tr><tr><td>Bob</td><td>Compilers</td></tr><tr><td>Bob</td><td>Databases</td></tr><tr><td>Bob</td><td>Graphics</td></tr><tr><td>Bob</td><td>Theory</td></tr></tbody></table>	<i>student-id</i>	<i>course-id</i>	Chris	Compilers	Chris	Graphics	Chris	Theory	Alice	Compilers	Alice	Theory	Bob	Compilers	Bob	Databases	Bob	Graphics	Bob	Theory	<table border="1" style="width: 100%; border-collapse: collapse;"><thead><tr><th style="border: none;"><i>course-id</i></th></tr></thead><tbody><tr><td>Databases</td></tr><tr><td>Compilers</td></tr><tr><td>Theory</td></tr></tbody></table>	<i>course-id</i>	Databases	Compilers	Theory	<table border="1" style="width: 100%; border-collapse: collapse;"><thead><tr><th style="border: none;"><i>student-id</i></th><th style="border: none;"><i>course-id</i></th></tr></thead><tbody><tr><td>Chris</td><td>Compilers</td></tr><tr><td>Chris</td><td>Theory</td></tr><tr><td>Alice</td><td>Compilers</td></tr><tr><td>Alice</td><td>Theory</td></tr><tr><td>Bob</td><td>Compilers</td></tr><tr><td>Bob</td><td>Databases</td></tr><tr><td>Bob</td><td>Theory</td></tr></tbody></table>	<i>student-id</i>	<i>course-id</i>	Chris	Compilers	Chris	Theory	Alice	Compilers	Alice	Theory	Bob	Compilers	Bob	Databases	Bob	Theory
<i>student-id</i>	<i>course-id</i>																																									
Chris	Compilers																																									
Chris	Graphics																																									
Chris	Theory																																									
Alice	Compilers																																									
Alice	Theory																																									
Bob	Compilers																																									
Bob	Databases																																									
Bob	Graphics																																									
Bob	Theory																																									
<i>course-id</i>																																										
Databases																																										
Compilers																																										
Theory																																										
<i>student-id</i>	<i>course-id</i>																																									
Chris	Compilers																																									
Chris	Theory																																									
Alice	Compilers																																									
Alice	Theory																																									
Bob	Compilers																																									
Bob	Databases																																									
Bob	Theory																																									

**Abbildung 6:** Semi-Join *Prüfungen* *Vorlesung*, um für die Eingaben der Division referenzielle Integrität zu gewährleisten.

Da bei aggregierenden Algorithmen Duplikate und die Erhaltung der referenziellen Integrität besonders beachtet werden muss, können bei diesen Algorithmen Fehler leichter auftreten als bei den skalaren Algorithmen.

### 2.1.3 Klassifikation der Algorithmen

Abb. 7 zeigt einen Überblick über mehrere Algorithmen, die zur Implementierung der relationalen Division verwendet werden können.

Divisions-Algorithmus	Algorithmen - Klasse	Daten- Klasse	Dividend S		Divisor T
			Q	D	
Nested-Loops-Counting-Division	Aggregierend	0	N	N	N
Nested-Loops-Division	Skalar				
Hash-Division	Skalar				
Transposed Hash-Division	Skalar				
Hash-Division für Divisorgruppen	Aggregierend	2	N	G	N
Transposed Hash-Div. für Divisorgruppen	Aggregierend				
Stream-Join-Division	Aggregierend				
Merge-Count-Division	Aggregierend	5	G	N	N
Hash-Division für Quotientengruppen	Skalar				
Transposed Hash-Div. für Quotientengruppen	Skalar				
Merge-Group-Division	Skalar	10	G <sub>1</sub>	G <sub>2</sub>	G <sub>2</sub>
Merge-Sort-Division	Skalar			S <sub>2</sub>	S <sub>2</sub>

**Abbildung 7:** Überblick über die Divisions-Algorithmen.

Für jeden Algorithmus wird seine Klasse sowie die Klasse der Eingabe bezüglich bestimmter Attribute, welche der Algorithmus handhaben kann, angegeben. Die Eingabedaten sind entweder nicht gruppiert (*N*), gruppiert (*G*) oder sortiert (*S*). Klasse 10 ist zuerst gruppiert nach *Q*, dargestellt durch *G*<sub>1</sub>, und jede Quotientengruppe ist gruppiert (*G*<sub>2</sub>) oder sortiert (*S*<sub>2</sub>) nach *D* in derselben Reihenfolge wie der Divisor. Der Überblick zeigt, dass es vier Hauptklassen von Eingabedaten gibt, die jeweils von mehreren Algorithmen abgedeckt werden:

- Klasse 0, die keinerlei Gruppierung voraussetzt,
- Klasse 2, die eine Gruppierung des Dividenden nach *D* voraussetzt,
- Klasse 5, die eine Gruppierung des Dividenden nach *Q* voraussetzt, und
- Klasse 10, eine Spezialisierung von Klasse 5, bei der zusätzlich für jede Quotientengruppe die Zeilen von *D* und die Zeilen des Divisors in derselben Reihenfolge sind.

Alle aggregierenden Algorithmen haben gemeinsam, dass in der ersten Phase der Divisor einmal gelesen wird, um die Anzahl der Zeilen zu zählen. Jeder dieser Algorithmen benutzt anschließend unterschiedliche Datenstrukturen um die Anzahl der Zeilen in einer Quotientengruppe zu speichern.

Im Rahmen dieser Studienarbeit werden in den nächsten Abschnitten die beiden skalaren Algorithmen Merge-Sort-Division und Hash-Division zur Implementierung des Divisions-Operators und für den Vergleich mit den Algorithmen für SCD und SCJ detaillierter beschrieben.

## 2.2 Merge-Sort-Division

Der Merge-Sort-Division-Algorithmus (MSD) setzt eine

- Sortierung des Divisors und
- eine Gruppierung des Dividenden nach  $Q$ , sowie für jede Quotientengruppe eine Sortierung nach  $D$  in derselben Reihenfolge (auf- oder absteigend) wie der Divisor

voraus. Bei diesen Eingabedaten handelt es sich also um einen Spezialfall der Datenklasse 10, wobei  $D$  und der Divisor sortiert und nicht nur gruppiert sind.

Bei der Bearbeitung einer einzelnen Quotientengruppe wird ein Merge-Join durchgeführt, die Bearbeitung aller Gruppen ähnelt einem Nested-Loops-Join. Im Folgenden wird die Verarbeitung der Zeilen einer einzelnen Gruppe mit aufsteigender Sortierung skizziert [1]. Zuerst wird die erste Zeile des Dividenden und des Divisors betrachtet. Falls der  $D$ -Wert der aktuellen Dividendenzeile mit dem Wert der aktuellen Divisorzeile übereinstimmt, fährt man mit der nächsten Zeile in beiden Tabellen fort. Ist der Wert von  $D$  größer als der Wert des Divisors, springt man zur nächsten Quotientengruppe, ist  $D$  kleiner als der Wert des Divisors, wird die Bearbeitung mit der nächsten Zeile in der Quotientengruppe und der aktuellen Zeile des Divisors fortgesetzt. Sind nun keine Zeilen mehr in der Gruppe aber noch mindestens eine Zeile im Divisor vorhanden, springt man zur nächsten Quotientengruppe. Sind allerdings im Divisor keine Zeilen mehr übrig, ist die Quotientengruppe ein Quotient und wird der Ausgabe hinzugefügt.

Der Merge-Sort-Ansatz ist ähnlich zum naiven Sort-Based-Algorithmus beschrieben in [2]. Beim naiven Sort-Based-Algorithmus werden die Daten vor dem Merge-Schritt im Gegensatz zu MSD explizit sortiert. Noch schwerer wiegt außerdem die Tatsache, dass bei diesem Ansatz der Dividend nicht nur nach  $Q$  gruppiert, sondern sogar unnötigerweise sortiert wird.

Abb. 8 zeigt, welche Zeilen zwischen Dividend und Divisor übereinstimmen. Dabei ist zu beachten, dass die Daten im Dividenden nach dem Attribut *student\_id* in einer zufälligen Reihenfolge gruppiert und nicht sortiert sind. Wäre beispielsweise eine Sortierung in aufsteigender Reihenfolge erforderlich, müsste die Gruppe *Alice* vor *Bob* und diese wiederum vor *Chris* erscheinen.

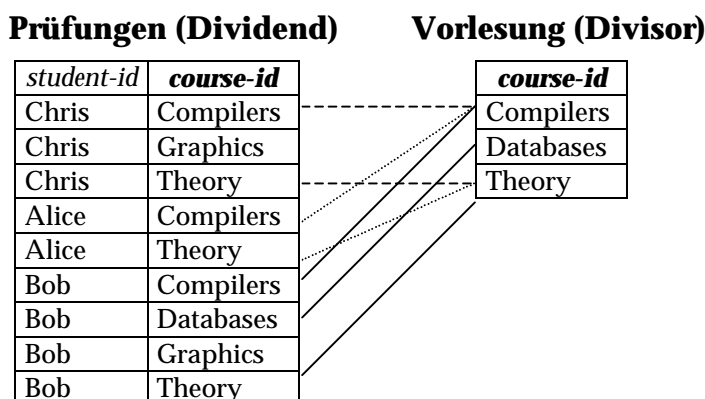


Abbildung 8: Merge-Sort-Division (MSD)

## 2.3 Hash-Division

Dieser Algorithmus benutzt zwei Hashtabellen, eine für den Divisor und eine für den Quotienten [2]. Die erste Hashtabelle wird Divisor- und die zweite Quotienten-Hashtabelle genannt (siehe Abb. 9).

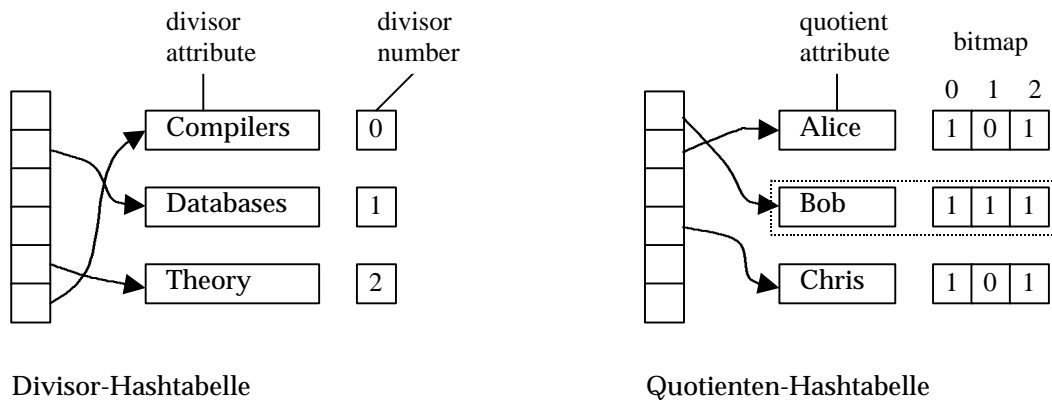


Abbildung 9: Hash-Division (HD)

Die Divisor-Hashtabelle enthält die Zeilen des Divisors. Jede dieser Zeilen besitzt einen Integer-Wert, genannt *divisor number*, der zusammen mit jeder Zeile gespeichert wird. In der Dividend-Hashtabelle wird jeder Quotientenkandidat zusammen mit einer Bitmap, die jeweils ein Bit pro Divisortupel besitzen, abgespeichert. Die Eingaben für die Hash-Division in Abb. 9 sind die gleichen wie in Abb. 3, wobei die Eingabe nicht sortiert sein muss, da bei HD für die Eingabedaten Klasse 0 erlaubt ist. In der ersten Phase der Hash-Division wird beim Scannen des Divisors die Divisor-Hashtabelle gebildet. Die Hashfunktion übernimmt die Attribute als Parameter und weist jeder Zeile des Divisors einen Hashbucket in der Hashtabelle zu. Um Duplikate im Divisor zu eliminieren, wird eine Zeile des Divisors nur dann gespeichert, wenn sie nicht schon in der Hashtabelle vorhanden ist. Außerdem wird jeder Zeile in der Hashtabelle eine eindeutige Divisornummer zugewiesen. Dieser Zähler wird mit *null* initialisiert und anschließend für jede Zeile, die in der Hashtabelle gespeichert wird, inkrementiert. Die Divisornummer wird als Index für die Bitmaps der Quotienten-Hashtabelle verwendet. Am Ende dieser Phase ist die Divisor-Hashtabelle vollständig aufgebaut (siehe linker Teil in Abb. 9).

In der zweiten Phase des Hash-Division-Algorithmus wird beim Scannen des Dividenden die Quotienten-Hashtabelle erzeugt. Bei jeder Zeile des Dividenden wird mit Hilfe der beim Aufbau der Divisor-Hashtabelle eingesetzten Hashfunktion geprüft, ob sein *D*-Wert in der Divisor-Hashtabelle enthalten ist. Falls kein übereinstimmendes Divisortupel existiert, wird dieses Tupel des Dividenden sofort verworfen. Ist der *D*-Wert aber in der Divisor-Hashtabelle enthalten, wird die entsprechende Divisornummer vermerkt und die Zeile des Dividenden als Quotientenkandidat betrachtet. Als nächstes wird durch Anwendung der Hashfunktion der Quotienten-Hashtabelle auf die Quotientenattribute des Tupels festgestellt, ob sich dieser Kandidat schon in der Quotienten-Hashtabelle befindet. Falls nein, wird eine neue Zeile mit dem Quotientenkandidaten und einer Bitmap mit einem Bit pro Divisortupel in

die Quotienten-Hashtabelle eingefügt. Bei der Bitmap wird das Bit, dessen Position der vorher vermerkten Divisornummer entspricht, auf 1 und der Rest auf 0 gesetzt. Ist der Kandidat jedoch schon in der Quotienten-Hashtabelle enthalten, muss nur ein einzelnes Bit in der Bitmap gesetzt werden. Da nur Quotientenkandidaten eingefügt werden, die noch nicht in der Tabelle enthalten sind, werden Duplikate eliminiert.

In der letzten Phase des Algorithmus werden schließlich alle Quotientenkandidaten, deren Bitmaps nur Einsen enthalten, ausgegeben.

Abb. 9 zeigt den Inhalt der Hashtabellen zu dem Zeitpunkt, nachdem alle Zeilen des Divisors und Dividenden von Abb. 3 eingelesen worden sind. Da nur *Bobs* Bitmap keine Nullen enthält, ist er der einzige Quotient.

### 3. Set Containment Join

Ein Set Containment Join ist ein Join zwischen mengenwertigen Attributen zweier Relationen, wobei die Join-Bedingung durch den Teilmengenoperator ( $\subseteq$ ) festgelegt ist [3].

Diese Joins sind beim relationalen Datenmodell nicht möglich, da hier die Attributwerte atomar sein müssen [4]. Bei erweiterten relationalen oder objekt-orientierten Datenmodellen sind allerdings mengenwertige Attribute erlaubt, und man kann sich viele interessante Anfragen vorstellen, die einen Join basierend auf Mengenvergleichen durchführen.

<b>Prüfungen</b>		<b>Vorlesung</b>	
<i>student-id</i>	<i>courses</i>	<i>courses</i>	<i>program</i>
Alice	{Compilers, Theory}	{Compilers, Databases, Theory}	Systems
Bob	{Compilers, Databases, Graphics, Theory}	{Compilers, Graphics}	Applications
Chris	{Compilers, Graphics, Theory}		

<b>Vorlesung</b>	<i>courses</i>	<i>courses</i>	<b>Prüfungen</b>
<i>student_id</i>	<i>courses</i>	<i>courses</i>	<i>program</i>
Bob	{Compilers, Databases, Graphics, Theory}	{Compilers, Databases, Theory}	Systems
Bob	{Compilers, Databases, Graphics, Theory}	{Compilers, Graphics}	Applications
Chris	{Compilers, Graphics, Theory}	{Compilers, Graphics}	Applications

**Abbildung 10:** Beispiel für ein Set Containment Join. Die Relationen sind nicht in 1NF und enthalten mengenwertige Attribute

Abb. 10 zeigt ein Beispiel für die Berechnung eines Set Containment Join, abgeleitet aus dem Szenario aus Abb. 3. Lediglich die Tabelle *Vorlesung* wurde um das Attribut *program*, welches die erforderliche Kombination an Vorlesungen für eine Vertiefungslinie angibt, erweitert. *Bob* hat alle Voraussetzungen erfüllt um *Systems* und *Applications* vertiefen zu können, *Chris* kann nur *Applications* vertiefen.

Allgemein lässt sich also ein Set Containment Join zwischen zwei mengenwertigen Attributen  $b$  und  $c$  zweier Relationen  $R(a, b)$  und  $S(c, d)$  folgendermaßen ausdrücken:

$$S \bowtie_{c \subseteq b} R = \{t \mid t \in S \times R \wedge c \subseteq b\}.$$

#### 3.1 Algorithmen

Die in der Literatur vorgeschlagenen Algorithmen zur Berechnung von Set Containment Joins basieren auf Signaturen und Partitionierung. Dabei haben alle Algorithmen gemeinsam, dass die Daten nicht in 1NF sind. Im nächsten Abschnitt werden die Prinzipien der Signaturen und Partitionierung eingeführt, anschließend werden die Set-Containment-Join-Algorithmen Partitioning Set Join und Adaptive Pick-and-Sweep-Join detailliert beschrieben.

### 3.1.1 Signaturen und Partitionierung

Zur Beschreibung der Prinzipien und Algorithmen beim Set Containment Join wird im folgenden ein einfaches Beispiel verwendet (siehe Abb. 11).

Relation S	Relation R
$a = \{2, 9\}$ $b = \{8, 18\}$ $c = \{1, 3\}$	$A = \{2, 4, 9\}$ $B = \{1, 3, 4\}$ $C = \{3, 8, 18\}$ $D = \{3, 4, 7\}$

**Abbildung 11:** Zwei Beispiel-Relationen mit mengenwertigen Attributen

In Abb. 11 [3] hat jede Relation eine Spalte mit einer Menge von Integer-Werten als Attributwert, wobei in  $S$  drei Tupel mit den Mengen  $\{2, 9\}$ ,  $\{8, 18\}$ ,  $\{1, 3\}$  und in  $R$  vier Tupel mit den Mengen  $\{2, 4, 9\}$ ,  $\{1, 3, 4\}$ ,  $\{3, 8, 18\}$  und  $\{3, 4, 7\}$  enthalten sind. Um die Mengen in  $R$  und  $S$  leichter referenzieren zu können, werden sie mit den Buchstaben  $a, b, c$  bzw.  $A, B, C, D$  bezeichnet. Um nun einen Set Containment Join  $S \bowtie R$  zu berechnen, müssen alle Tupel  $(s, r) \in S \times R$  mit  $s \supseteq r$  gefunden werden. In unserem Beispiel ist  $S \bowtie R = \{(a, A), (b, C), (c, B)\}$ .

Natürlich kann man  $S \bowtie R$  immer berechnen, indem man jedes Tupel im Kreuzprodukt  $S \times R$  nach der Teilmengen-Bedingung überprüft. Dieser Ansatz würde in unserem Beispiel  $|S| * |R| = 3 * 4 = 12$  Mengenvergleiche erfordern. Bei großen Relationen wäre dieser Ansatz mit  $|S| * |R|$  Vergleichen allerdings sehr zeitaufwendig, da bei jedem Vergleich Elemente beider Mengen zu einem großen Teil durchlaufen und verglichen werden müssen, um festzustellen, ob eine Menge Teilmenge der anderen Menge ist. Falls die Relationen nicht in den Hauptspeicher passen, würden bei der Bildung des Kreuzprodukts außerdem enorme E/A-Kosten auftreten.

Um einen Set Containment Join effizient berechnen zu können, wurden zwei wichtige Techniken vorgeschlagen [3]: Signaturen und Partitionierung.

Hinter der Verwendung von Signaturen steckt die Idee, dass teure Mengenvergleiche durch billigere Vergleiche von Signaturen ersetzt werden sollen. Eine Signatur wird dabei durch einen Hash-Wert über den Inhalt der Menge gebildet. Zur Veranschaulichung wird das einfache Beispiel aus Abb. 12, welches sich auf die Daten aus Abb. 11 bezieht, betrachtet. In diesem Beispiel wird die Signatur jeder Menge aus den Relationen  $R$  und  $S$  durch einen Vektor mit 4 Bits dargestellt. Jedes Mengenelement  $j$  setzt das Bit an Position  $(j \bmod 4)$  im Vektor auf 1. So wird beispielsweise für Menge  $b = \{8, 18\}$  Bit 0 ( $8 \bmod 4$ ) und Bit 2 ( $18 \bmod 4$ ) im Vektor gesetzt und man erhält  $sig(\{8, 18\}) = 1010$ .

x	S	sig(x)
a		0110
b		1010
c		0101

y	R	sig(y)
A		1110
B		1101
C		1011
D		1001

**Abbildung 12:** 4-Bit Signaturen der Mengen in  $R$  und  $S$

Sei  $b$  die „Bit-Teilmenge“, dann ist  $sig(x) \leq b \ sig(y)$  für jedes Paar von Mengen  $x, y$  mit  $x \subseteq y$  gültig. Dadurch können viele Mengenvergleiche durch das Testen auf Bit-Teilmenge vermieden werden. Ist beispielsweise  $sig(c) \not\leq b \ sig(A)$ , dann kann  $c$  keine Teilmenge von  $A$  sein. Ob eine Signatur der Menge  $x$  Bit-Teilmenge einer anderen Signatur der Menge  $y$  ist, kann sehr effizient berechnet werden, indem man prüft, ob  $sig(x) \& \< sig(y) = 0$  ist, wobei  $\&$  und  $\<$  die bitweisen AND- und NOT- Operatoren sind. In unserem Beispiel müssen nach 12 Signaturvergleichen bei nur noch 4 Mengenpaaren geprüft werden, ob die eine Menge Teilmenge der anderen ist:  $(a, A)$ ,  $(b, A)$ ,  $(b, C)$  und  $(c, B)$ . Von diesen Mengen wird  $(b, A)$  als falsches Zwischenergebnis verworfen.

Die Benutzung von Signaturen reduziert die Anzahl der benötigten Mengenvergleiche deutlich, allerdings sind immer noch  $|S| * |R|$  Signaturvergleiche durchzuführen. Zur Reduzierung der Signaturvergleiche und Verbesserung der Leistung, wird die Technik der Partitionierung eingesetzt. Dabei wird eine Join-Anfrage  $S \bowtie R$  in  $k$  kleinere Sub-Joins  $S_1 \bowtie R_1, \dots, S_k \bowtie R_k$  aufgeteilt, so dass gilt:

$S \bowtie R = \bigcup_k S_k \bowtie R_k$ . Eine *Partitions-Funktion*  $\delta$  weist jedes Tupel von  $S$  einer oder mehreren Partitionen  $S_1, \dots, S_k$  zu, und jeden Tupel von  $R$  einer oder mehreren Partitionen  $R_1, \dots, R_k$ .

Für unser Beispiel aus Abb. 11 wird  $\delta$  so gewählt, dass gilt  $\delta(a) = \delta(b) = \delta(A) = \delta(B) = \{1\}$ ,  $\delta(c) = \delta(C) = \{2\}$ , und  $\delta(D) = \{1, 2\}$ .  $S$  wird also partitioniert in  $S_1 = \{a, b\}$ ,  $S_2 = \{c\}$ ,  $R$  wird partitioniert in  $R_1 = \{A, B, C\}$ ,  $R_2 = \{C, D\}$ . Der Join  $S \bowtie R$  wird also aufgeteilt in  $(S_1 \bowtie R_1) \cup (S_2 \bowtie R_2)$ . Die Berechnung von  $S_1 \bowtie R_1 = \{a, b\} \times \{A, B, C\}$  und  $S_2 \bowtie R_2 = \{c\} \times \{C, D\}$  erfordert nur  $2 * 3 + 1 * 2 = 8$  Signaturvergleiche, d.h. durch die Partitionierung wurde die Anzahl der Vergleiche von 12 auf 8 reduziert. Das Verhältnis  $8/12$  wird als *Vergleichsfaktor (comparison factor)* bezeichnet. Der Vergleichsfaktor liegt immer zwischen 0 und 1.

Neben der Reduzierung der Signaturvergleiche hilft die Partitionierung auch bei der Behandlung großer Relationen  $R$  und  $S$ , die nicht in den Hauptspeicher passen, indem die Partitionen  $R_1, \dots, R_k$  und  $S_1, \dots, S_k$  auf Platte gespeichert werden. Um die E/A-Kosten beim Zurückschreiben auf Platte und Einlesen in den Hauptspeicher zu minimieren, enthalten die Partitionen typischerweise nur die Signaturen und die entsprechenden *Tupel-Identifizier*. In unserem Beispiel werden  $|\{a, b\}| + |\{c\}| = 3$  Signaturen von  $S_{1,2}$  und  $|\{A, B, C\}| + |\{C, D\}| = 5$  Signaturen von  $R_{1,2}$  temporär auf Platte gespeichert. Das Verhältnis zwischen der Gesamtzahl der Signaturen, die auf Platte geschrieben werden, und der Gesamtzahl der Tupel in  $R$  und  $S$ , wird *Replikationsfaktor (replication factor)* genannt. In unserem Beispiel beträgt der Replikationsfaktor

$3+5/3+4 = 8/7$ . Der optimale Replikationsfaktor, der bei einem partitionsbasierten Join erreicht werden kann ist 1.

Die größte Herausforderung bei einer effektiven Partitionierung ist die Konstruktion einer Partitionsfunktion  $\delta$ , die den Vergleichs- und Replikationsfaktor minimiert. Natürlich muss  $\delta$  korrekt sein, d. h. die Funktion muss sicherstellen, dass alle Tupel, die miteinander gejoint werden sollen, auch gefunden werden.

### 3.1.2 Partitioning Set Join (PSJ)

Um den Partitioning Set Join (PSJ) zu beschreiben, wird wiederum das Beispiel aus dem vorigen Kapitel verwendet. Weiter nehmen wir an, dass  $R$  und  $S$  aus Abb. 11 in  $k = 8$  Partitionen aufgeteilt werden sollen. Die Partitionsnummer jeder Menge aus  $S$  wird festgelegt durch ein einzelnes, zufällig ausgewähltes Element der Menge. Wählt man z. B. aus der Menge  $a = \{2, 9\} \subseteq S$ , das Element 9 zufällig aus, wird  $a$  einer der Partitionen  $0, 1, \dots, 7$  zugewiesen, indem der Elementwert modulo  $k = 8$  genommen wird, d. h.  $a$  wird der Partition mit Index  $(9 \bmod 8) = 1$ , also  $S_1$  zugewiesen. Ist 18 das zufällig ausgewählte Element der Menge  $b = \{8, 18\}$ , dann wird  $b$  Partitionsnummer  $2 = (18 \bmod 8)$  zugewiesen. Die Menge  $c$  fällt beim zufällig ausgewählten Element 3 schließlich in die Partition  $S_3$ . Nun wird das gleiche Verfahren bei der Relation  $R$  angewandt, allerdings werden *alle* Elemente jeder Menge betrachtet, um die Partitionszugehörigkeit festzustellen. Folglich wird Menge  $A = \{2, 4, 9\}$  den Partitionen  $R_2, R_4$  und  $R_1$  zugewiesen,  $B$  den Partitionen  $R_1, R_3$  und  $R_4$  usw. Abb. 13 zeigt die komplette Zuweisung der Relationen  $R$  und  $S$ .

Partition	$S_i$	$R_i$
0		C
1	a	A B
2	b	A C
3	c	B C D
4		A B D
5		
6		
7		D

Abbildung 13: Partitionierung mit PSJ

Sind beide Relationen partitioniert, wurden also alle Signaturen und zugehörigen Tupel-Identifizier in Dateien geschrieben, wird jedes der  $k$  Paare von Partitionen von Platte gelesen und unabhängig von den anderen Paaren gejoint. Wird z. B. ein Join

der Partitionen  $S_3$  und  $R_3$  durchgeführt, wird die Signatur der Menge  $c$  aus  $S_3$  gelesen und verglichen mit den Signaturen der Mengen  $B$ ,  $C$ , und  $D$ , die in  $R_3$  gespeichert sind. Bei der Berechnung von  $S_3 \cap R_3$  müssen also  $1 * 3 = 3$  Signaturvergleiche durchgeführt werden. Die Gesamtzahl der erforderlichen Signaturvergleiche in unserem Beispiel ist  $0 + 2 + 2 + 3 + 0 + 0 + 0 + 0 = 7$ , wobei insgesamt 15 Signaturen in Dateien geschrieben müssen. Folglich erhalten wir für dieses Beispiel einen Vergleichsfaktor von  $7/12 \approx 0,58$  und einen Replikationsfaktor von  $15/3+4 = 2,14$ .

### 3.1.3 Adaptive Pick-and-Sweep-Join (APSJ)

Der Adaptive Pick-and-Sweep-Join (APSJ) erweitert den PSJ Algorithmus. Zur Beschreibung von APSJ wird weiterhin das Beispiel aus Abb. 11 und  $k = 8$  Partitionen angenommen. Außerdem wird vorausgesetzt, dass es  $k - 1 = 7$  Boolean-Hash-Funktionen  $h_1, \dots, h_7$  gibt, denen eine Menge von Integer-Werten als Eingabe übergeben wird, und die 0 oder 1 als Ausgabe liefern. Die Funktion  $h_i(x)$  wird definiert als  $h_i(x) = 1$  iff  $x: (e \bmod 9) = i$  für  $i = 1, \dots, 7$ , d. h. die Funktion mit Index  $i$  „feuert“ für Menge  $s$  nur dann, wenn  $s$  ein Element enthält, welches Modulo 9 genommen  $i$  ergibt. Jede dieser Funktionen ist im folgenden Sinne monoton: immer wenn  $h_i$  für eine gegebene Menge  $x$  feuert, feuert  $h_i$  garantiert auch für eine Obermenge von  $x$ . Wenn wir beispielsweise die Menge  $c = \{1, 3\}$  betrachten, dann liefert  $(1 \bmod 9) = 1$  und  $(3 \bmod 9) = 3$ , d. h.  $h_1(c) = h_3(c) = 1$ . Bei der Menge  $B = \{1, 3, 4\}$  feuert zusätzlich zu  $h_1$  und  $h_3$  noch die Funktion  $h_4$ , da  $(4 \bmod 9) = 4$  ergibt. Abb. 14 zeigt die Werte, die alle sieben Boolean-Hash-Funktionen für die Mengen  $a$ ,  $b$ ,  $c$  und  $A$ ,  $B$ ,  $C$ ,  $D$  annehmen.

x	S	<u>h<sub>1</sub></u>	<u>h<sub>2</sub></u>	h <sub>3</sub>	h <sub>4</sub>	h <sub>5</sub>	h <sub>6</sub>	h <sub>7</sub>
a		0	<u>1</u>	0	0	0	0	0
b		0	0	0	0	0	0	0
c		<u>1</u>	0	1	0	0	0	0

y	R	h <sub>1</sub>	h <sub>2</sub>	h <sub>3</sub>	h <sub>4</sub>	h <sub>5</sub>	h <sub>6</sub>	h <sub>7</sub>
A		0	1	0	1	0	0	0
B		1	0	1	1	0	0	0
C		0	0	1	0	0	0	0
D		<u>0</u>	0	1	1	0	0	1

Abbildung 14: Boolean-Hash-Funktionen

APSJ kann jede monotone Boolean-Hash-Funktion benutzen, nicht nur die in unserem Beispiel benutzten modulo-basierten.

Bei der Benutzung dieser  $k - 1 = 7$  Funktionen, werden unsere Tabellen den  $k = 8$  Partitionen folgendermaßen zugeteilt. Für jede Menge  $s \in S$  werden die Indizes  $j$  der Hash-Funktionen, für die  $h_j(s) = 1$  gilt, betrachtet. Aus dieser Indexmenge wird ein Index  $i$  zufällig ausgewählt, und die Menge  $s$  der Partition  $S_i$  zugewiesen. Ist die Indexmenge leer, hat also keine Hash-Funktion für  $s$  gefeuert, wird  $s$  der Partition  $S_0$  zugeteilt. So kann man z. B. bei der Menge  $c$  zwischen Index 1 und Index 3 auswählen. Wird Index 1 zufällig ausgewählt, fällt  $c$  in Partition  $S_1$  (die ausgewählten Indizes sind in Abb. 14 unterstrichen). Da bei der Menge  $b$  keine Hash-Funktion feuert, wird sie in Partition  $S_0$  platziert. Die Mengen  $r \in R$  werden wiederum *allen* Partitionen  $R_j$  mit  $h_j(r) = 1$  zugewiesen, d.h. man nimmt für die Auswahl der Partitionen die Indizes aller feuernden Funktionen. Zusätzlich befindet sich jede Menge  $r$  noch in der Parti-

tion  $R_0$ . Die Menge  $A$  wird also neben den Partitionen  $R_2$  und  $R_4$  zusätzlich noch der Partition  $R_0$  zugewiesen. Abb. 15 zeigt die komplette Zuweisung mittels APSJ für unsere Beispieltabellen.

Partition	$S_i$	$R_i$
0	b	A B C D
1	c	A B
2	a	A
3		B C D
4		A B D
5		
6		
7		D

**Abbildung 15:** Partitionierung mit APSJ

Zusammen mit den Partitionen  $R_0$  und  $S_0$  erzeugen die  $k - 1$  Hash-Funktionen  $k$  Partitionen. Dies erlaubt uns eine korrekte Partitionierung, selbst wenn  $S$  leere Mengen oder Mengen, für die kein  $h_i$  feuert, enthält. Insgesamt werden  $4 + 1 + 1 + 0 + 0 + 0 + 0 + 0 = 6$  Mengenvergleiche durchgeführt und 16 Signaturen auf Platte gespeichert. Man erhält also in unserem Beispiel einen Vergleichsfaktor von  $6/12 = 0,5$  und einen Replikationsfaktor von  $16/3+4 = 2,14$ .

Sowohl PSJ als auch APSJ können durch eine variierende Anzahl von Partitionen an unterschiedliche Leistungsanforderungen angepasst werden. Werden z . B. mehr Partitionen benutzt, sind weniger Vergleiche notwendig. Allerdings steigt die Anzahl der Replikate. Außerdem hat man bei APSJ im Gegensatz zu PSJ mit den Boolean-Hash-Funktionen eine weitere „Tuningmöglichkeit“. Feuern alle Hash-Funktionen mit einer hohen Wahrscheinlichkeit, wird jede Partition  $R_i$  fast die gesamte Tabelle  $R$  enthalten, d. h. Joins werden teuer. Feuern die Funktionen allerdings mit einer sehr geringen Wahrscheinlichkeit, entspricht  $S_0$  ungefähr  $S$  und man erhält einen Join  $S_0 \bowtie R_0 = S \bowtie R$  [3]. Um die Arbeit zu minimieren, müssen folglich Funktionen ausgewählt werden, die mit mittlerer Wahrscheinlichkeit feuern.

## 4. Set Containment Division

Die Set Containment Division (SCD) ist eine Erweiterung der relationalen Division und ermöglicht die Berechnung des Set-Containment-Problems bei Tabellen in 1NF. Bei SCD darf der Divisor analog zu den Mengen beim SCJ aus mehreren Gruppen bestehen, die Tabellen enthalten allerdings - wie bei der relationalen Division - nur atomare Attribute.

Es gibt also unterschiedliche Möglichkeiten, um Mengen zu speichern. Bei SCJ spricht man von einer „Nested“-Darstellung der Mengen, bei SCD von einer „Unnested“-Darstellung (siehe Abb. 16).

Unnested	Nested								
<table border="1" style="border-collapse: collapse; width: 100%;"> <tr><td style="padding: 2px 10px;">Chris</td><td style="padding: 2px 10px;">Compilers</td></tr> <tr><td style="padding: 2px 10px;">Chris</td><td style="padding: 2px 10px;">Graphics</td></tr> <tr><td style="padding: 2px 10px;">Chris</td><td style="padding: 2px 10px;">Theory</td></tr> </table>	Chris	Compilers	Chris	Graphics	Chris	Theory	<table border="1" style="border-collapse: collapse; width: 100%;"> <tr> <td style="padding: 2px 10px;">Chris</td> <td style="padding: 2px 10px;">{Compilers, Graphics, Theory}</td> </tr> </table>	Chris	{Compilers, Graphics, Theory}
Chris	Compilers								
Chris	Graphics								
Chris	Theory								
Chris	{Compilers, Graphics, Theory}								

**Abbildung 16:** Darstellung von mengenwertigen Attributen

Die Attributwerte werden als mehrere Werte gespeichert: Bei der Nested-Darstellung werden die Werte in einer bestimmten Datenstruktur als ein Attribut gespeichert, die Unnested-Darstellung speichert sie als mehrere Tupel. In Abb. 16 wird als Beispiel ein einzelnes Tupel der Relation *Prüfung(student\_id, courses)* dargestellt, wobei *student\_id* ein atomares und *courses* ein mengenwertiges Attribut ist, d.h. *Chris* hat die Prüfungen der Vorlesungen *Compilers*, *Graphics* und *Theory* erfolgreich abgelegt. Nur die Unnested-Darstellung im linken Teil der Abbildung ist in 1NF.

Der SCD-Operator ist äquivalent zum SCJ-Operator und liefert als Ergebnis die gleiche Tabelle zurück wie SCJ, allerdings ohne die Verbundattribute. Abb. 17 zeigt die Berechnung einer Set Containment Division basierend auf den gleichen Daten wie in Abb. 10 allerdings mit Daten in 1NF.

Prüfungen /R(a, b)	Vorlesung /S(c, d)																																
<table border="1" style="border-collapse: collapse; width: 100%;"> <thead> <tr><th style="padding: 2px 10px;"><i>student-id</i></th><th style="padding: 2px 10px;"><i>course-id</i></th></tr> </thead> <tbody> <tr><td style="padding: 2px 10px;">Alice</td><td style="padding: 2px 10px;">Compilers</td></tr> <tr><td style="padding: 2px 10px;">Alice</td><td style="padding: 2px 10px;">Theory</td></tr> <tr><td style="padding: 2px 10px;">Bob</td><td style="padding: 2px 10px;">Compilers</td></tr> <tr><td style="padding: 2px 10px;">Bob</td><td style="padding: 2px 10px;">Databases</td></tr> <tr><td style="padding: 2px 10px;">Bob</td><td style="padding: 2px 10px;">Graphics</td></tr> <tr><td style="padding: 2px 10px;">Bob</td><td style="padding: 2px 10px;">Theory</td></tr> <tr><td style="padding: 2px 10px;">Chris</td><td style="padding: 2px 10px;">Compilers</td></tr> <tr><td style="padding: 2px 10px;">Chris</td><td style="padding: 2px 10px;">Graphics</td></tr> <tr><td style="padding: 2px 10px;">Chris</td><td style="padding: 2px 10px;">Theory</td></tr> </tbody> </table>	<i>student-id</i>	<i>course-id</i>	Alice	Compilers	Alice	Theory	Bob	Compilers	Bob	Databases	Bob	Graphics	Bob	Theory	Chris	Compilers	Chris	Graphics	Chris	Theory	<table border="1" style="border-collapse: collapse; width: 100%;"> <thead> <tr><th style="padding: 2px 10px;"><i>student-id</i></th><th style="padding: 2px 10px;"><i>course-id</i></th></tr> </thead> <tbody> <tr><td style="padding: 2px 10px;">Compilers</td><td style="padding: 2px 10px;">Systems</td></tr> <tr><td style="padding: 2px 10px;">Databases</td><td style="padding: 2px 10px;">Systems</td></tr> <tr><td style="padding: 2px 10px;">Theory</td><td style="padding: 2px 10px;">Systems</td></tr> <tr><td style="padding: 2px 10px;">Compilers</td><td style="padding: 2px 10px;">Applications</td></tr> <tr><td style="padding: 2px 10px;">Graphics</td><td style="padding: 2px 10px;">Applications</td></tr> </tbody> </table>	<i>student-id</i>	<i>course-id</i>	Compilers	Systems	Databases	Systems	Theory	Systems	Compilers	Applications	Graphics	Applications
<i>student-id</i>	<i>course-id</i>																																
Alice	Compilers																																
Alice	Theory																																
Bob	Compilers																																
Bob	Databases																																
Bob	Graphics																																
Bob	Theory																																
Chris	Compilers																																
Chris	Graphics																																
Chris	Theory																																
<i>student-id</i>	<i>course-id</i>																																
Compilers	Systems																																
Databases	Systems																																
Theory	Systems																																
Compilers	Applications																																
Graphics	Applications																																

**Prüfungen**  $\div$  **courses\_id** **course\_id** **Vorlesung** /T(a, d)

<i>student-id</i>	<i>program</i>
Bob	Systems
Bob	Systems
Chris	Applications

**Abbildung 17:** Set Containment Division, Tabellen in 1NF

Formal kann die Set Containment Division basierend auf zwei Relationen  $R(a, b)$  und  $S(c, d)$  mittels der relationalen Division folgendermaßen ausgedrückt werden [1].

$$T(a, d) = R \div_{b \supseteq c} S = \bigcup_{x \in p_d(S)} ((R \div_{p_c(S_{d=x})}) \times (x))$$

Bei diesem Ansatz werden mehrere Divisionen durchgeführt. Bei jeder Division wird der gesamte Dividend  $R$  durch die Tupel von  $S$  dividiert, die zur selben Gruppe gehören. Es gibt also so viele Divisionen wie es unterschiedliche Werte von  $S.d$  gibt.

Bei unserer Implementierung wird in einer ersten Phase zuerst der Divisor gelesen, um die unterschiedlichen Werte (Gruppen) von  $d$  in einem Vektor zu speichern. Danach wird der Vektor in einer for-Schleife durchlaufen und in jedem Schritt eine Selektion, Projektion, relationale Division und ein Join durchgeführt.

Bei der Selektion werden die Tupel von  $S$  ausgewählt, deren  $d$ -Wert mit dem Wert übereinstimmt, der sich am aktuellen Index des Vektors befindet. Anschließend wird auf den resultierenden Tupeln eine Projektion durchgeführt um die Verbundattribute des Divisors zu eliminieren. Danach ist eine relationale Division nötig. Die relationale Division kann durch einen beliebigen Algorithmus realisiert sein, in unserem Fall können die Merge-Sort- oder Hash-Division zum Einsatz kommen. Schließlich muss mit allen Ergebnistupeln der Division ein Join mit dem Wert des Vektors durchgeführt werden, der schon bei der Selektion herangezogen wurde. Die daraus resultierenden Tupel sind das Ergebnis einer Set Containment Division.

## 5. Analyse der Algorithmen

In diesem Kapitel werden zuerst die Merge-Sort- und Hash-Division als Algorithmen für die relationale Division, und anschließend Partitioning Set Join und Adaptive Pick-and-Sweep Join als Algorithmen für den Set Containment Join analysiert.

### 5.1 Analyse der Algorithmen für die relationale Division

Bei den folgenden Betrachtungen wird angenommen, dass die Eingabedaten der Division aus einer Dividentabelle  $S(Q, D)$  und einer Divisortabelle  $T(D)$  bestehen. Dabei handelt es sich bei  $Q$  um die Menge der Quotientenattribute und bei  $D$  um die Menge der Divisorattribute, wie in Kapitel 2.1.1 definiert.

Die Analyse der Algorithmen beschränkt sich auf die Angabe der Komplexität des *schlechtesten* und *typischen* Fall jeweils für die Laufzeit und Speicherbedarf in  $O$ -Notation, basierend auf der Größe (Anzahl der Zeilen) des Dividenten  $|S|$  und der Größe des Divisors  $|T|$ . Die E/A-Kosten werden hier nicht betrachtet. Außerdem wird bei der Bestimmung der Komplexität der Algorithmen die Anzahl der unterschiedlichen Werte der Quotientenattribute von  $Q$  im Dividenten  $|Q|$ , benutzt. Für  $Q$  und  $S$  gilt immer  $|Q| \leq |S|$ , und im *schlechtesten* Fall  $|Q| = |S|$ , d.h. jede Zeile in  $S$  ist ein potenzieller Quotient. Zur Ableitung von Formeln der *typischen* Zeit- und Speicherkomplexität wird  $|S| \gg |T|$  angenommen, d.h. es gibt sehr viele Quotientenkandidaten bzw. die Anzahl der Zeilen eines typischen Quotientenkandidaten ist viel größer als die Anzahl der Zeilen des Divisors. Diese Situation wird als der *typische* Fall betrachtet, da die relationale Division als Berechnung einer *Menge* von Ergebnistupeln definiert wird, wobei diese Menge in der realen Welt von beträchtlicher Größe ist, und eine große Ergebnismenge nur auftritt, wenn der Divident erheblich mehr Zeilen als der Divisor enthält [1].

#### 5.1.1 Merge-Sort-Division

Bei der Merge-Sort-Division ist der schlechteste Fall bei der Zeitkomplexität  $O(|S| + |Q| \times |T|) = O(|S| + |S| \times |T|) = O(|S| \times |T|)$ , da der Divident genau einmal gelesen wird, und vom Divisor so viele Zeilen gelesen werden wie die Anzahl der Quotientenkandidaten multipliziert mit der Anzahl der Zeilen des Divisors.

Die typische Zeitkomplexität ist  $O(|S| \times |T|)$ . Beim Speicherbedarf ist sowohl der schlechteste als auch der typische Fall jeweils  $O(1)$ , da nur eine konstante Anzahl kleiner Datenstrukturen (je eine Zeile von  $S$  und  $T$ ) im Speicher gehalten werden müssen.

#### 5.1.2 Hash-Division

Beim Hash-Division-Algorithmus werden der Divident und der Divisor genau einmal gelesen. Da Hashtabellen mit einer nahezu konstanten Zugriffszeit zum Einsatz kommen, beträgt die Zeitkomplexität bei diesem Algorithmus im schlechtesten und typischen Fall  $O(|S| + |T|)$  bzw.  $O(|S|)$ . Die Speicherkomplexität besteht aus  $O(|T|)$  um die Divisor-Hashtabelle zu speichern, plus  $O(|Q| \times |T|)$  für die Quotienten-Hashtabelle. Die Größe der Bitmap in der Quotienten-Hashtabelle ist proportio-

nal zu  $|S|$ . Da im schlechtesten Fall  $|Q| = |S|$  gilt, ist die Speicherkomplexität im schlechtesten und typischen Fall  $O(|S| \times |T|)$ .

## 5.2 Analyse der Algorithmen für Set Containment Join

Bei der Analyse der Set-Containment-Join-Algorithmen werden die Vergleichs- und Replikationsfaktoren als Effizienzmaß verwendet. Der Vergleichsfaktor ist das Verhältnis zwischen der aktuellen Anzahl von Signaturvergleichen und  $|S| \cdot |R|$ , der Replikationsfaktor ist das Verhältnis aus der Anzahl der Signaturen, die in einer Datei gespeichert sind und  $|S| + |R|$  (siehe auch Kapitel 3.1.1). Der Vergleichsfaktor reflektiert die CPU-Auslastung, der Replikationsfaktor den E/A-Aufwand bei der Partitionierung.

Um den Vergleichs- und Replikationsfaktor zu bestimmen, werden der abgeleitete Parameter  $\tilde{e} = \frac{q_R}{q_S}$ , der das Verhältnis der Mengenkardinalitäten angibt, und der Parameter  $\tilde{n} = \frac{|R|}{|S|}$ , der das Verhältnis der Tabellengrößen angibt, benutzt. In Abb. 18 sind alle Variablen, die bei der Analyse verwendet werden, zusammengefasst.

$ R ,  S $	Kardinalitäten der Tabellen, $S$ $c$ $b$ $R$
$\tilde{n}$	Verhältnis der Tabellenkardinalitäten, $\tilde{n} = \frac{ R }{ S }$
$\tilde{e}_R, \tilde{e}_S$	Mengkardinalitäten in $R$ und $S$
$\tilde{e}$	Verhältnis der Mengenkardinalitäten, $\tilde{e} = \frac{q_R}{q_S}$
$k$	Anzahl der Partitionen

**Abbildung 18:** Bei der Analyse der Algorithmen benutzte Variablen

In [3] wurden die optimalen Vergleichs- und Replikationsfaktoren hergeleitet (siehe Abb. 19):

Algorithmus	Vergleichs- und Replikationsfaktor
PSJ	$comp_{PSJ} = 1 - \left(1 - \frac{1}{k}\right)^{q_R}$ $repl_{PSJ} = \frac{1}{1+r} + \frac{r}{1+r} k \left(1 - \left(1 - \frac{1}{k}\right)^{q_R}\right)$
APSJ	$comp_{APSJ} = 1 - \frac{k-1}{k-1+l} \cdot \left(\frac{l}{k-1+l}\right)^{\frac{1}{k-1}}$ $repl_{APSJ} = \frac{1}{1+r} + \frac{r}{1+r} \cdot \left(k - (k-1) \left(\frac{l}{k-1+l}\right)^{\frac{1}{k-1}}\right)$

**Abbildung 19:** Zusammenfassung der Vergleichs- und Replikationsfaktoren von PSJ und APSJ

Man erkennt aus Abb. 19, dass der Vergleichs- und Replikationsfaktor von PSJ direkt von der Mengenkardinalität  $\tilde{e}_R$  abhängt, während die Formeln für APSJ nur abhängig sind vom Verhältnis  $\tilde{e} = \frac{q_R}{q_S}$ . Aus dieser Beobachtung lässt sich folgern, dass APSJ

große Mengen effizienter handhaben sollte als PSJ. Im nächsten Abschnitt wollen wir diese Vermutung genauer betrachten.

### 5.2.1 Bedeutung der Formeln

**Vergleichsfaktor:** Zuerst werden wir zeigen, dass der Vergleichsfaktor bei einer steigenden Zahl von Partitionen sinkt. In Abb. 20 wird  $comp_{PSJ}$  und  $comp_{APSJ}$  für die Mengenkardinalitäten  $e_R = e_S = 10$ ,  $e_R = e_S = 100$ , und  $e_R = e_S = 1000$  dargestellt [3]. Da APSJ nur vom Verhältnis  $\bar{e}$  der Mengenkardinalitäten abhängt, und  $\bar{e} = 1$  in allen drei Fällen gilt, wird bei diesem Algorithmus aus drei Kurven eine.

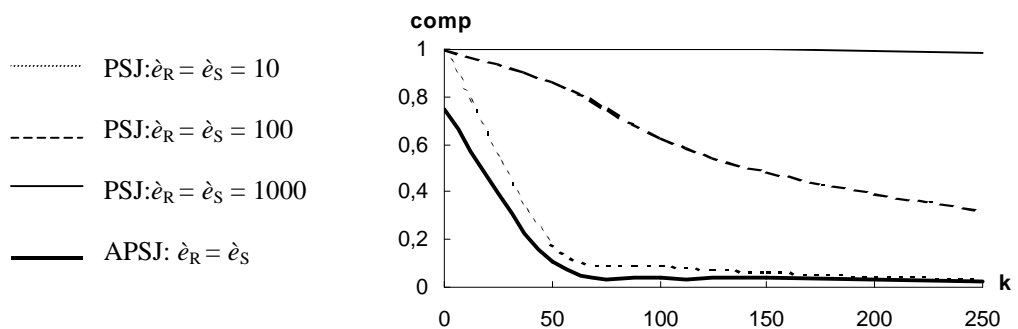


Abbildung 20: Vergleichsfaktor vs. k

Wie man sieht, ist PSJ bei großen Mengenkardinalitäten wie beispielsweise  $e_R = e_S = 1000$  nicht effizient.

Abb. 21 zeigt, wie der Vergleichsfaktor mit wachsenden Kardinalitäten der Mengen in S steigt. Die Kardinalität der Mengen in S und die Anzahl der Partitionen ist konstant mit  $e_S = 100$  und  $k = 128$ , die Mengenkardinalitäten in R variieren von  $e_R = 10$  bis  $e_R = 1000$ , d.h.  $\bar{e}$  hat einen Wert von 0,1 bis 10.

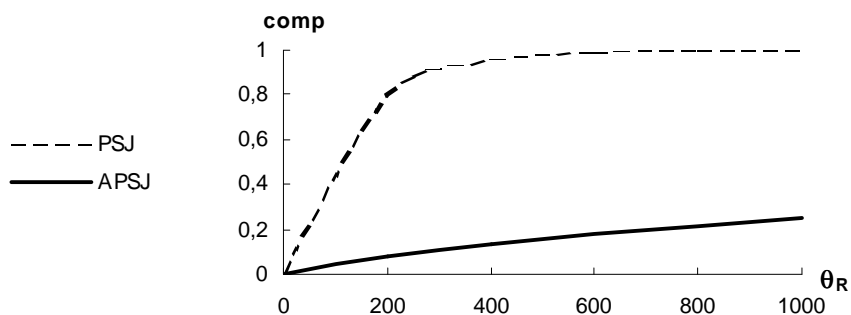
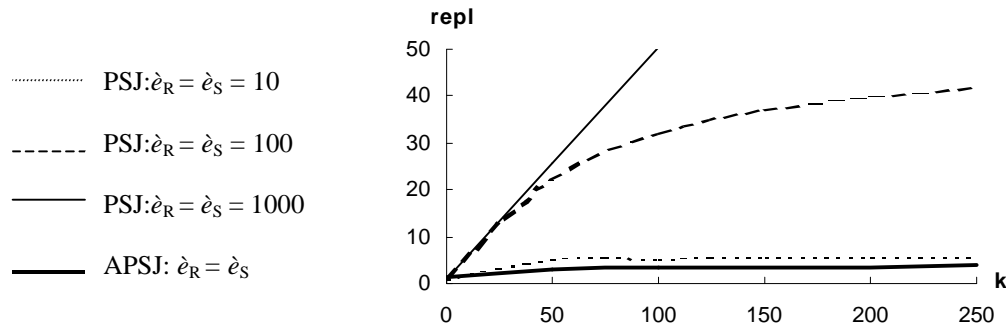


Abbildung 21: Vergleichsfaktor vs.  $e_R$

Wie man in der Abbildung erkennt, bleiben beim Anwachsen des Verhältnisses der Kardinalitäten die Werte von  $comp_{APSJ}$  immer unter den Werten von  $comp_{PSJ}$ .

**Replikationsfaktor:** Der Replikationsfaktor ist abhängig vom Verhältnis  $\tilde{n}$  der Tabellenkardinalitäten. Zuerst wird der Fall  $|S| = |R|$ , d.h.  $\tilde{n} = 1$  betrachtet. Abb. 22 zeigt

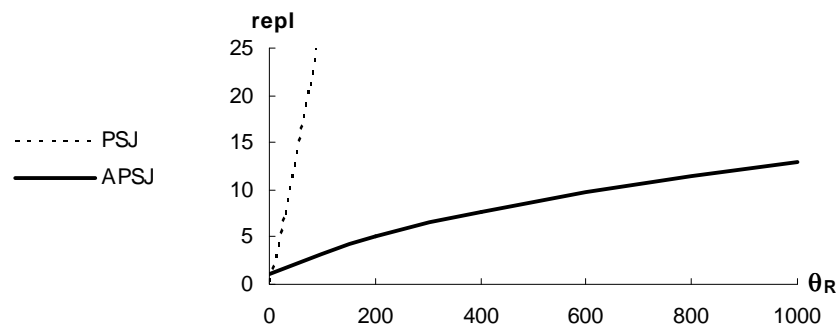
das Wachstum der Replikationsfaktoren  $repl_{APSJ}$  und  $repl_{PSJ}$  bei einer steigenden Anzahl von Partitionen für die Fälle  $\hat{e}_R = \hat{e}_S = 10$ ,  $\hat{e}_R = \hat{e}_S = 100$ , und  $\hat{e}_R = \hat{e}_S = 1000$ . Da  $\tilde{e}$  in allen drei Fällen den Wert 1 hat, und  $repl_{APSJ}$  neben dem Verhältnis  $\tilde{n}$  nur noch vom Verhältnis der Mengenkardinalitäten abhängt, ergibt sich für APSJ - genau so wie in Abb. 20 - eine einzige Kurve.



**Abbildung 22:** Replikationsfaktor vs.  $k$

Selbst bei kleinen Mengenkardinalitäten wie  $\hat{e}_R = \hat{e}_S = 10$  ergibt sich für  $repl_{APSJ}$  ein besserer Wert als für  $repl_{PSJ}$ . Bei größeren Mengen, z.B.  $\hat{e}_R = \hat{e}_S = 100$  und  $k = 128$  muss PSJ  $35 * (|S| + |R|)$  Signaturen auf Platte schreiben, das sind 10-mal mehr Daten, die gespeichert werden müssen, als bei APSJ.

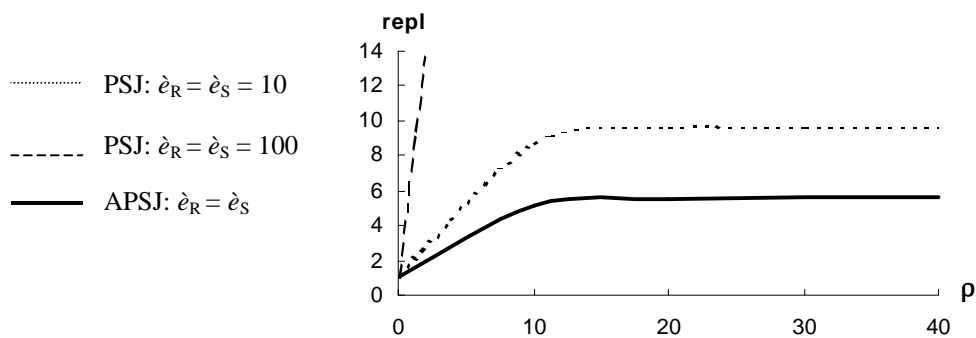
Die Auswirkungen des Verhältnisses der Mengenkardinalitäten auf den Replikationsfaktor wird in Abb. 23 dargestellt. Wieder werden die Variablen  $k = 128$  und  $\hat{e}_S = 100$  festgelegt,  $\hat{e}_R$  wird wieder variiert von 10 bis 1000.



**Abbildung 23:** Replikationsfaktor vs.  $\hat{e}_S$  (für  $\tilde{n} = 1$ )

Auch hier ergeben sich für den Replikationsfaktor bei APSJ deutliche Vorteile gegenüber PSJ.

Abb. 24 zeigt das Verhalten der Replikationsfaktoren wenn die Relation  $R$  größer als die Relation  $S$  ist ( $|S| < |R|$ ). Die Anzahl der Partitionen bleibt konstant bei  $k = 128$ .



**Abbildung 24:** Replikationsfaktor vs.  $\tilde{n}$  (für  $\hat{e} = 1$ )

Sowohl  $repl_{PSJ}$  als auch  $repl_{APSJ}$  steigen mit wachsendem  $\tilde{n}$ . Ähnlich wie in den anderen Schaubildern ergeben sich aber bei APSJ günstigere Werte für den Replikationsfaktor als bei PSJ.

Die Analysen in diesem Abschnitt deuten also darauf hin, dass bei beiden Algorithmen der Vergleichsfaktor (und damit der CPU-Aufwand) mit wachsendem  $k$  sinkt, während der Replikationsfaktor (und damit der E/A-Aufwand) mit wachsendem  $k$  steigt. Folglich gibt es eine optimale Anzahl von Partitionen  $k$ , welche die Gesamtlauzeit für jeden Algorithmus minimiert [3]. Weiter deuten die Analysen darauf hin, dass PSJ der am besten geeignete Algorithmus bei sehr kleinen Mengenkardinalitäten (unter 10 Elementen) ist, während APSJ die erste Wahl bei steigenden Mengenkardinalitäten ist.

## 6. Implementierung der Algorithmen

Die Implementierung erfolgte in der Programmiersprache Java. Eine der wichtigsten Vorteile von Java ist die Plattformunabhängigkeit des erzeugten (Byte-) Codes, der unter allen Windows-Versionen, Macintosh, UNIX usw. lauffähig ist. Die unterschiedlichen Systeme müssen nur einen Java-Interpreter (Java Virtual Machine) bereitstellen, um die Bytecode-Anweisungen zu interpretieren. Da der Interpreter den Code zeilenweise übersetzt und nicht auf einmal vor der Ausführung des Programms, kommt es allerdings zu langsameren Ausführungszeiten als bei Compilern. Ein weiterer Vorteil von Java ist seine Objektorientierung, d.h. jedes Element ist ein Objekt und alle Objekte sind Instanzen einer Klasse. Damit lassen sich komplexere Programme leichter entwickeln als bei der strukturierten Programmierung. Die Syntax von Java ist an C++ angelehnt, allerdings ist sie einfacher zu erlernen, da beispielsweise keine Zeiger verwendet werden.

### 6.1 Die Java-Klassenbibliothek XXL

Die heutigen Datenbank-Management-Systeme (DBMS) sind häufig zu unflexibel, um sich schnell genug an die Bedürfnisse der Anfragebearbeitung anzupassen [5]. Sie stellen deshalb Mechanismen bereit, um neue Funktionalität in das System einzubauen, und erlauben, dass Benutzer ihre eigene Funktionalität implementieren und in das verwendete DBMS integrieren.

Die Java-Bibliothek XXL (eXtensible and fleXible Library) wurde von der Universität Marburg entwickelt und kann bei solchen Szenarien eingesetzt werden. XXL ist einfach zu benutzen, plattformunabhängig und unterstützt die Implementierung neuer Anfragefunktionalität [6]. Die Bibliothek besteht aus den folgenden Komponenten:

1. Das Cursor-Package stellt eine Algebra der wichtigsten Anfrageoperatoren bereit. Die Operatoren der Algebra erfordern, dass die Ein- und Ausgabe einer Iterator-Schnittstelle genügt. Um den Import externer Datenquellen z.B. über JDBC zu unterstützen, enthält XXL eine erweiterte Algebra basierend auf der Java-Schnittstelle *ResultSet*.
2. XXL stellt eine vielfältige Infrastruktur externer Datenstrukturen, die bei der Implementierung neuer Datenbankfunktionalität hilfreich sind, zur Verfügung. Zusätzlich bietet XXL einen sehr flexiblen Puffer-Mechanismus an.
3. In XXL wurden als Framework zahlreiche Indexstrukturen implementiert, wobei viele Implementierungen (R-Baum, M-Baum, X-Baum) schon existieren, und die Implementierung neuer Strukturen stark vereinfacht wird. So wurden beispielsweise schon verschiedenen Typen von Operationen auf Indizes effizient implementiert.

Im nächsten Abschnitt wird das Cursor-Konzept in XXL genauer beschrieben.

## 6.1.1 Das Cursor-Konzept in XXL

Jeder Cursor wird einer Anfrage zugewiesen, d.h. er enthält die Datenmenge, die sich bei dieser Anfrage als Resultat ergibt. Ein Cursor kann dann auf die Ergebnismenge der ihm zugeordneten Anfrage zeilenweise zugreifen, und ermöglicht so die weitere Verwendung der einzelnen Ergebnisobjekte.

Die in XXL verfügbaren Cursor sind unabhängig von den spezifischen Strukturen und der Speicherrepräsentation der Objekte. XXL bietet eine Cursor-Algebra an, die geeignet ist, komplexe Anfragen in einer einfachen Art und Weise zu definieren.

Cursor sind nah verwandt mit Iteratoren, die schon in der Java API vorhanden sind. Um Kompatibilität mit der Java API zu gewährleisten, wird in XXL die Iterator-Funktionalität der Java API angepasst. Die Schnittstelle *Iterator* besteht in der Java API aus den folgenden drei Methoden:

```
interface Iterator {
    boolean hasNext();
    Object next();
    void remove();
}
```

Die Methode *hasNext* prüft, ob die zugrundeliegende Datenmenge noch ein Objekt enthält, während *next* auf das nächste Objekt der Menge zugreift. Eine Klasse, die die Schnittstelle *Iterator* realisiert, muss diese zwei Methoden implementieren. Die dritte Methode ist optional, d.h. sie muss nicht zwingend realisiert werden. Ist die Methode *remove* implementiert, wird bei ihrem Aufruf das letzte Objekt, welches der Iterator zurückgeliefert hat, aus der zugrundeliegenden Datenmenge entfernt.

Die Schnittstelle *Cursor* in XXL erweitert die Funktionalität eines Iterators mit den folgenden Methoden:

```
interface Cursor extends Iterator {
    Object peek();
    void update(Object o);
    void reset();
    void close();
}
```

Die Methode *peek* zeigt das nächste Objekt der Iteration ohne den Zustand der Iteration zu ändern. Durch den Aufruf von *reset* beginnt die Iteration noch mal von vorne. Mit der Methode *close* werden Ressourcen, die ein Cursor möglicherweise belegt hat (z.B. Netzwerkverbindungen), wieder freigegeben.

Die Absicht bei der Definition der Cursor-Algebra in XXL bestand darin, dass die unterschiedlichen Operatoren der Algebra einen Cursor als Eingabe erhalten und einen Cursor als Ausgabe zurückliefern. Zusätzlich enthält die Algebra eine Menge von *Eingabe-Operatoren*, deren Eingaben nicht der *Cursor*-Schnittstelle entsprechen. Eine komplexe Anfrage wird dann durch einen Operatorbaum dargestellt, wobei die Knoten des Baums speziellen Operatoren entsprechen. Die Blattknoten sind die Eingabe-Operatoren, die internen Knoten des Operatorbaums die *Prozess-Operatoren*.

In XXL gibt es viele unterschiedliche Eingabe-Operatoren, die spezielle Datenquellen umwandeln (Wrapper). Die beiden folgenden sind zwei der wichtigsten:

- *IteratorCursor*: java.util.Iterator    Cursor
- *ResultSetCursor*: java.sql.ResultSet    Cursor

Der erste Operator wandelt einen Iterator in einen Cursor um. Dies ist wichtig, falls die zu bearbeitende Datenquelle nur die *Iterator*-Schnittstelle der Java API implementiert.

Der zweite Operator kommt bei der Bearbeitung von Datenquellen zum Einsatz, die der Java-Schnittstelle *ResultSet* entsprechen. Dieser Operator kann besonders nützlich beim Import von Tabellen aus relationalen Datenbanken sein. Die Schnittstelle *ResultSet* enthält einen Verweis auf die aktuelle Zeile in einer Tabelle. Die *next*-Methode dieser Schnittstelle bewegt den Verweis zur nächsten Zeile. Da diese Methode den Wert *false* zurückliefert, wenn keine Zeilen mehr in der Ergebnismenge enthalten sind, kann sie in einer *while*-Schleife benutzt werden, um über die Ergebnismenge zu iterieren.

Prozess-Operatoren benötigen mindestens einen Cursor (z.B. erzeugt durch die Umwandlung eines Iterators in einen Cursor durch den Eingabe-Operator *IteratorCursor*) als Eingabe. Ein Beispiel für einen Prozess-Operator ist die Klasse *Aggregator*, die für eine Eingabe eine Aggregation vornimmt (z.B. Summe oder Durchschnitt).

## 6.2 Die implementierten Klassen

In diesem Abschnitt werden alle wichtigen Klassen beschrieben, die im Rahmen dieser Studienarbeit in Java implementiert wurden, um die Datenbankoperatoren zu realisieren.

### 6.2.1 Die Klasse *MyFileMetaDataCursor*

Die meisten der in XXL schon enthaltenen Operatoren wie auch die von uns im Rahmen der Studienarbeit implementierten Operatoren (z.B. *HashDivision*, siehe nächstes Kapitel) benötigen Eingaben vom Typ *MetaDataCursor*.

*MetaDataCursor* ist eine Schnittstelle, die von der *Cursor*-Schnittstelle erbt, aber *Cursor* realisiert, die im Gegensatz zur normalen *Cursor*-Schnittstelle Metadaten bereitstellen. Der Hauptunterschied zwischen einem *Cursor* und einem *MetaDataCursor* liegt darin, dass ein *MetaDataCursor* noch zusätzlich über die Methode *getMetaData()* verfügt, die Metadaten über die Elemente zurückliefert, welche der *Cursor* ausliefert. Um nun Dateien als Eingabedaten beispielsweise als Dividend oder Divisor verwenden zu können, müssen diese in einen *MetaDataCursor* umgewandelt werden, damit sie dem Konstruktor einer Klasse z.B. *HashDivision* (siehe nächstes Kapitel) als Parameter übergeben werden können. Dies ermöglicht die Klasse *MyFileMetaDataCursor*, welche die Schnittstelle *MetaDataCursor* implementiert und von der Klasse *xxl.relationalNEU.FileMetaDataCursor* abgeleitet wurde, indem sie für eine gegebene Datei einen *MetaDataCursor* bereitstellt. Bei *FileMetaDataCursor* muss die Datei in der ersten Zeile alle durch ein Begrenzungszeichen getrennte Spaltennamen enthalten.

Der Default-Wert für das Begrenzungszeichen ist das Tabulatorzeichen (`\t`). Die zweite Zeile muss alle Spaltentypen (ebenfalls durch das Begrenzungszeichen getrennt) enthalten, wobei bei *FileMetaDataCursor* nur die Typen *Number* und *Varchar* erlaubt sind. Die dritte Zeile ist leer. Anschließend repräsentieren alle folgenden Zeilen jeweils ein Tupel der Relation, wobei zwischen jedem Attributwert des Tupels das Begrenzungszeichen stehen muss und Nullwerte per Default mit `#` dargestellt werden. Beispiel für eine Datei:

```
student_id  NAME      course_id
Number     Varchar   varChar

1    Alice  Compilers
1    Alice  Theory
2    Bob    Compilers
2    Bob    Databases
2    Bob    Graphics
2    Bob    Theory
3    Chris  Compilers
3    Chris  Graphics
3    Chris  Theory
4    #      Test
```

Durch den Aufruf eines Konstruktors von *FileMetaDataCursor* mit dem Dateinamen als Parameter, kann die spezifizierte Datei als *MetaDataCursor* verwendet werden. Die Klasse *MyFileMetaDataCursor* wurde nun so erweitert, dass Eingabedateien die eine Nested-Darstellung haben, auch als *MetaDataCursor* verwendet werden können, um als Eingaberelationen der Klasse *SetContainmentJoin* zum Einsatz zu kommen. Deshalb wurde als weiterer Spaltentyp *Array* zugelassen, so dass die oben dargestellte Datei auch in der folgenden Nested-Darstellung vorliegen und als *MetaDataCursor* verwendet werden kann:

```
student_id  NAME      course_id
Number     Varchar   Array

1    Alice  [Compilers, Theory]
2    Bob    [Compilers, Databases, Graphics, Theory]
3    Chris  [Compilers, Graphics, Theory]
```

Im Zuge der Entwicklung von *MyFileMetaDataCursor* musste auch die XXL-Klasse *MetaData* zu *MyMetaData* weiterentwickelt werden, um für den Spaltentyp *Array* die richtigen Metadaten zurückzuliefern.

## 6.2.2 Die Klasse HashDivision

Die Klasse *HashDivision* stellt einen Divisions-Operator der relationalen Algebra zur Verfügung. Sie erbt von der abstrakten Klasse *AbstractCursor* und implementiert die Schnittstelle *MetaDataCursor*.

*AbstractCursor* implementiert die Methoden der *Cursor*-Schnittstelle und enthält zusätzlich noch einige (z.T. abstrakte) Methoden, die nützlich sind, wenn man Objekte

erzeugen will, die anschließend durch den Cursor zurückgegeben werden sollen. Für diese Fälle wird die abstrakte Methode `computeNext()` bereitgestellt, die jede Klasse, die von `AbstractCursor` erbt, implementiert, um das nächste Resultat zu erhalten.

Bei der Klasse `HashDivision` handelt es sich also um einen Cursor, der das nächste Element einer Hash-Division mittels `computeNext()` zurückgibt, und außerdem noch Metadaten (z.B. Name oder Typ der Attribute) über seine Elemente mit Hilfe der Methode `getMetaData()` bereitstellt. Die Klasse besitzt einen Konstruktor, der einen Dividenden und einen Divisor, die beide vom Typ `MetaDataCursor` sind, als Eingabe erwartet, und nur die Methode `initialize`, mit dem gleichen Dividenden und Divisor als Parameter, aufruft.

Mit der Ausführung von `initialize (dividend, divisor)` erfolgt dann die eigentliche Berechnung der Hash-Division basierend auf der Beschreibung des Hash-Division-Algorithmus in Kapitel 2.3. Die resultierenden Tupel werden dann mit `computeNext()` als Ergebnis zurückgegeben. Die verwendeten Hashtabellen für den Divisor und Quotienten werden mittels der Klasse `java.util.HashMap` der Java API gebildet. Das folgende Codefragment zeigt die Bildung der Divisor-Hashtabelle:

```

HashMap divisorHashTable = new HashMap();
int divisorCount = 0; // Number of divisor rows.
while (divisor.hasNext()) {

    hashCodeArrayTuple tuple = new hashCodeArrayTuple(Tuples.getObjectArray ((Tu-
ple) divisor.next()), (ResultSetMetaData) divisor.getMetaData());

    if (!divisorHashTable.containsKey(tuple)) {
        divisorHashTable.put(tuple, new Integer(divisorCount));
        divisorCount++;
    }
}

```

Die Tupel, die in den Hashtabellen zusammen mit der *divisor number* bzw. der Bitmap abgelegt werden, sind vom Typ `HashCodeArrayTuple`. Diese Klasse erweitert die XXL-Klasse `ArrayTuple` lediglich um die Methode `hashCode` zur Berechnung eines Hash-Werts für einen Tupel. Mit der Klasse `ArrayTuple` erzeugt man ein Tupel, indem man den Konstruktor der Klasse mit einem Feld von Objekten und Metadaten, die vom Datentyp `java.sql.ResultSetMetaData` sind, aufruft. Das Objekt-Array enthält in jedem Feld eine Spalte des Tupels, die Metadaten enthalten die Informationen über die einzelnen Spalten. Die Methode `getObjectArray(Tuple tuple)` der Klasse `xxl.relationalNEU.Tuples` wandelt ein Tupel wieder in ein Object-Array um. Nach dem Anlegen der Hashtabellen werden mittels des Operators *Projection* der sich im Package `xxl.relationalNEU` befindet, die Quotientenattribute  $Q$  und Divisorattribute  $D$  des Dividenden bestimmt. Anschließend wird in einer while-Schleife der Dividend gelesen, und für jede Zeile geprüft, ob die Quotientenattribute dieser Zeile einen Quotientenkandidaten bilden. Bei der Bitmap, die zusammen mit jedem Quotientenkandidaten in der Quotienten-Hashtabelle gespeichert wird, handelt es sich um ein Boolean-Array, dessen Werte wie in Kapitel 2.3 beschrieben auf *true* oder *false* gesetzt werden müssen. Schließlich werden alle Quotientenkandidaten, deren Bitmaps nur *true* enthalten in einem Vektor vom Typ `java.util.Vector` gespeichert und bilden das Ergebnis der Hash-Division.

Neben dem Konstruktor, den Methoden *initialize*, *computeNext* und *getMetaData* enthält die Klasse *HashDivision* noch die Methode *reset()*, die den Cursor auf den Anfangszustand zurücksetzt, so dass der Aufrufer die Ergebnismenge des Cursors noch einmal durchlaufen kann.

### 6.2.3 Die Klasse *SetContainmentDivision*

Die Klasse *SetContainmentDivision* realisiert die Set-Containment-Division. Sie erweitert wie *HashDivision* die abstrakte Klasse *AbstractCursor* und implementiert *MetaDataCursor*. Dem Konstruktor müssen neben den Parametern für den Dividenden und Divisor, die beide vom Typ *MetaDataCursor* sind, zusätzlich noch der Divisionsoperator übergeben werden. Bei unserer Implementierung wurden als Divisionsoperator *HashDivision* und *MySortBasedDivision* zugelassen. Die Klasse *MySortBasedDivision* ist ähnlich zur Klasse *SortBasedDivision* aus dem Package *xxl.relationalNEU*, allerdings ist bei *MySortBasedDivision* ein Reset des Cursors möglich, während bei *SortBasedDivision* der Aufruf von *reset()* zu Fehlern in der Programmausführung führte. *MySortBasedDivision* realisiert ebenso wie *HashDivision* einen Divisions-Operator der relationalen Algebra, erwartet aber im Gegensatz zur Hash-Division, dass die Eingabedaten (Dividend und Divisor) sortiert sind und keine Duplikate enthalten.

Der Konstruktor von *SetContainmentDivision* ruft ähnlich wie bei *HashDivision* lediglich die Methode *initialize* auf. In *initialize* werden zunächst mit Hilfe des Operators *Projection* die Attribute bzw. Spalten des Divisors bestimmt, die zu *S.c* und *S.d* (siehe Kapitel 1) gehören. Anschließend werden in einem Vektor (*tuplesD*) die unterschiedlichen Werte von *S.d* gespeichert. Schließlich wird die Set-Containment-Division entsprechend der Definition aus Kapitel 4 durchgeführt, wobei nacheinander die Operationen Selektion, Projektion, Division und Bildung des kartesischen Produkts mit Hilfe der entsprechenden implementierten Operatoren durchzuführen sind, so oft, wie es unterschiedliche Werte von *S.d* gibt. Das folgende Codestück enthält diesen Teil der Implementierung:

```

    for (int i = 0; i < tuplesD.size(); i++) {
        divisor.reset();
        //
        // tupleComparator is used for the selection
        //
        final Comparator tupleComparator = Tuples.getTupleComparator(columnIndicesD,
columnIndicesTuple);
        //
        // Selection
        //
        Selection selection = new Selection(divisor, new Predicate () {
            public boolean invoke(Object o) {
                return (tupleComparator.compare(o,
                (HashCodeArrayTuple) tuplesD.get(vectorIndex)) == 0);
            }
        });
        //
        // Projection
        //
        Projection projection = new Projection (selection, columnIndicesC, HashCodeArray-
Tuple.FACTORY_METHOD);
        //
        // Division

```

```

//
if (divisionOperator == HASH_DIVISION)
    division = new HashDivision(dividend, projection);
else if (divisionOperator == SORT_BASED_DIVISION)
    division = new MySortBasedDivision(dividend, projection, HashCodeArrayTu-
ple.FACTORY_METHOD);
else
    throw new Exception("unknown division operator");
//
// Cartesian product
//
join = new NestedLoopsJoin(division, new MySingleObjectCursor (
    (HashCodeArrayTuple)tuplesD.get(vectorIndex)), null,
    HashCodeArrayTuple.FACTORY_METHOD,
    NestedLoopsJoin.CARTESIAN_PRODUCT);
dividend.reset();

```

Nach Durchführung der Join-Operation werden die Ergebnistupel dieses Cursors in einem Vektor gespeichert und anschließend mit *computeNext()* einzeln als Ergebnisse der gesamten Set-Containment-Division zurückgeliefert.

In der Klasse *SetContainmentDivision* werden neben dem Konstruktor, den Methoden *initialize* und *computeNext* - wie bei *HashDivision* - noch die Methoden *reset()* und *getMetaData()* implementiert.

## 6.2.4 Die Klasse SetContainmentJoin

Die Klasse *SetContainmentJoin* realisiert den **Adaptive Pick-and-Sweep-Join** zur Berechnung eines Set Containment Join. Sie erweitert wie die in den beiden vorangegangenen Abschnitten beschriebenen Klassen die abstrakte Klasse *AbstractCursor* und implementiert *MetaDataCursor*. Als Konstanten werden die Signaturlänge mit dem Wert 4 und die Anzahl der Partitionen mit dem Wert 8 festgelegt. Dem Konstruktor müssen die beiden Eingaberelationen (beides *MetaDataCursor*), sowie jeweils deren durchschnittliche Mengenkardinalität vom Datentyp *double* als Parameter übergeben werden.

Bei den Eingaberelationen sind wir von dem Szenario ausgegangen, dass es sich bei den Mengenelementen der Tupel ebenfalls um Tupel handelt, die wiederum aus genau einem Attribut bestehen. Die Tupel der Eingaberelationen haben also beispielsweise die Form:

TID	Item
1	[Item: 3, Item: 5, Item: 12, Item: 7]
2	[Item:2, Item: 72, Item: 1, Item: 18]
...	...

Diese Eingabedaten können mit Hilfe der Klasse *ArrayToNest* erzeugt werden. Als Eingabe von *ArrayToNest* wird ein *MetaDataCursor* erwartet, der Daten in Nested-Darstellung enthält, so wie sie *MyFileMetaDataCursor* bereitstellt. Der Konstruktor von *ArrayToNest* erhält also einen *MyFileMetaDataCursor* als Parameter, um dessen Tupel wie z.B.

TID: 1 Item: [4, 6, 13, 8]

in Tupel der Form

TID: 1 Item:[Item: 4, Item: 6, Item: 13, Item: 8]

umzuwandeln, wobei als Metadaten für die Tupel in der Menge, die Metadaten der Menge (*Item*) verwendet werden. Bei dieser Umwandlung wird das ursprüngliche Array durch einen Vektor, dessen Elemente Tupel sind, ersetzt.

Der Konstruktor von *SetContainmentJoin* ruft so wie bei *HashDivision* und *SetContainmentDivision* lediglich die Methode *initialize* auf. Zunächst wird in *initialize* die „Modulo“-Zahl, die bei der Berechnung des Partitionsindex verwendet wird (in Kapitel 3.1.3 hat sie den Wert 9), wie in [3] angegeben, berechnet:

```
final int bitStringModule = (int)Math.round(1.0 / (1.0 - Math.pow(lambda / (lambda + numPartitions - 1.0), 1.0 / (numPartitions - 1.0) / avgSetCardinality1)));
```

Anschließend werden die Partitionen, die bei APSJ verwendet werden und vom Typ *xxl.io.BufferedRandomAccessFile* sind, initialisiert und in einem Array gespeichert (*partitionsR* bzw. *partitionsS*). Danach werden die Eingaberelationen gelesen und jeweils für jedes Tupel die Signatur (verwendete Datenstruktur: Boolean-Array) und der Index einer Partition, der das Tupel zugewiesen wird, berechnet. Da bei der Eingaberelation *S* (siehe Kapitel 3.1.3) aus der Indexmenge, die für die Zuteilung der Partitionen herangezogen wird, ein Index zufällig ausgewählt wird, ist bei unserer Implementierung der erste Index, der sich für diese Indexmenge qualifiziert, für die Partitionszuteilung zuständig.

In XXL wird eine *Converter*-Schnittstelle zur Verfügung gestellt, um ein Objekt als Strom binärer Daten speichern zu können. Zur Konvertierung von Tupeln wurde die Klasse *ArrayTupleConverter* implementiert. Um die E/A-Kosten zu verringern, werden in dieser Implementierung solche Konverter (z.B. *converterR*) eingesetzt, und die Tupel in der zugehörigen Partition als ein binären Datenstrom gespeichert. Der folgende Code zeigt diese Schritte bei der Bearbeitung der ersten Eingaberelation:

```
while (cursor1.hasNext()) {
    //
    //Build the signature for each tuple, and assign it to a randomly chosen
    //partition(here: the assignment depends only on the first element
    //for which the partitionIndex < numPartitions - 1 is true).
    //
    hashCodeArrayTuple currentTuple = (hashCodeArrayTuple)cursor1.next();
    Object[] currentArray = Tuples.getObjectArray(currentTuple);
    Vector currentSet = (Vector)currentArray[lengthCursor1 - 1];
    boolean notPartitionNull = false;
    int partitionIndex = 0;
    boolean[] signatureArray = new boolean[signatureLength];
    for (int i = 0; i < currentSet.size(); i++) {
        hashCodeArrayTuple setElement = (hashCodeArrayTuple)currentSet.get(i);
        int value = ((Integer)Tuples.getObjectArray(setElement)[0]).intValue();
        signatureArray[value % signatureLength] = true;
        if (!notPartitionNull) {
            partitionIndex = value % bitStringModule;
            if (partitionIndex < numPartitions - 1) {
                partitionIndex++;
            }
        }
    }
}
```

```

        converterR.write(partitionsR[partitionIndex], currentTuple);
        notPartitionNull = true;
    }
}
}

```

Anschließend wird jede Signatur mit der schon vorhandenen Klasse *xxl.io.BooleanArrayConverter* konvertiert, und in der gleichen Partition wie das zugehörige Tupel gespeichert. Schließlich werden die Spaltennamen und -typen der ersten Eingaberelation jeweils in String-Arrays gespeichert, um mit *computeNext()* die resultierenden Tupel eines Set Containment Join bilden zu können.

Für die zweite Eingaberelation sind entsprechende Schritte durchzuführen und der Code ist von kleineren Ausnahmen, wie z.B. der Partitionszuteilung eines Tupel (siehe Kapitel 3.1.3), abgesehen identisch.

Die Implementierung der *computeNext*-Methode zur Berechnung des nächsten Ergebnistupel ist bei *SetContainmentJoin* wesentlich aufwendiger als bei den Klassen *HashDivision* und *SetContainmentDivision*. So sind mehrere Variablen notwendig um den aktuellen Zustand bei der Berechnung des Joins zu speichern (z.B. aktuelle Partitionen, Position in Partitionen, ...). Für jedes Tupel in einer Partition *S* muss seine Signatur mit den Signaturen aller Tupel, die sich in der entsprechenden *R*-Partition befinden, verglichen werden. Um zu prüfen, ob eine Signatur Bit-Teilmenge einer anderen ist (siehe Kapitel 3.1.1), werden ihre Werte in der Klasse *BitSet* der Java API gespeichert, da diese Datenstruktur eine einfache Berechnung der bitweisen NOT- und AND-Operatoren erlaubt. Ist die Signatur eines Tupels aus *S* Bit-Teilmenge einer Signatur aus *R*, muss schließlich noch geprüft werden, ob die Menge des Tupels aus *S* Teilmenge der Menge des Tupels aus *R* ist. Falls ja, wird mit Hilfe der in *initialize* berechneten und in den String-Arrays gespeicherten Metadaten ein neues Ergebnistupel gebildet.

Außerdem werden in *SetContainmentJoin* noch die Methoden *reset()* und *getMetaData()*, sowie *hasNext()* und *next()* (siehe Kapitel 6.1.1) implementiert.

## 6.2.5 Die Klassen *Unnest*, *GroupBasedNest* und *HashBasedNest*

Die Klasse *Unnest* ermöglicht die Umwandlung von Daten in Nested-Darstellung in Daten in Unnested-Darstellung, d.h. in Daten die sich in 1NF befinden. Diese Klasse ist nützlich, wenn die Laufzeit von *SetContainmentJoin* mit der Laufzeit von *SetContainmentDivision* verglichen werden soll, wobei dieselben Daten, allerdings in unterschiedlicher Darstellung zum Einsatz kommen. Man kann also einen *MyFileMetaDataCursor*, der Nested-Daten enthält, dem Konstruktor von *SetContainmentJoin* als Eingaberelation übergeben, und denselben *MyFileMetaDataCursor* der Klasse *Unnest* als Parameter übergeben, um einen *MetaDataCursor* zu erhalten, der die gleichen Daten enthält, wie die Eingabe von *SetContainmentJoin*, aber nun als Eingabe für *SetContainmentDivision* verwendet werden kann.

Die umgekehrte Richtung bei der Umwandlung der Datendarstellung ist mit den beiden Klassen *GroupBasedNest* und *HashBasedNest* möglich. Während *GroupBasedNest* einen Cursor mit Daten erwartet, bei denen eine Gruppierung auf den Mengenattributen vorliegt, ist bei *HashBasedNest* keine Gruppierung des Eingabecursors notwendig.

## 6.2.6 Die Klasse MyDataGenerator

Um Daten bzw. Relationen mit verschiedenen Eigenschaften erzeugen und bei den Experimenten, die im 7. Kapitel beschrieben werden, einsetzen zu können, wurde die Klasse *MyDataGenerator* entwickelt.

Der Konstruktor dieser Klasse wird mit mehreren Parametern aufgerufen:

```
public MyDataGenerator(int numGroups, int setCardinality, int startValue, int
    numItems, String elementValues, String layout, boolean duplicates,
    String type, String data)
```

Im folgenden werden die Parameter kurz erläutert:

- *numGroups*: dieser Parameter legt die Anzahl der Gruppen (Transaktionen) in einer Tabelle fest.
- *setCardinality*: legt die Kardinalität der mengenwertigen Attribute in der Tabelle fest. *setCardinality* multipliziert mit *numGroups* ergibt die Gesamtzahl der Zeilen in einer Tabelle.
- *startValue*: gibt den Startwert der mengenwertigen Attribute an, falls keine Zufallszahlen verwendet werden.
- *numItems*: legt den Bereich fest, aus dem Zufallszahlen generiert werden. Ist *numItems* =0, werden keine Zufallszahlen verwendet.
- *elementValues*: hat *elementValues* den Wert *same*, haben alle Mengen den gleichen Wert, andernfalls sind die Mengen unterschiedlich.
- *layout*: wird für *layout* der Wert *unnest* angegeben, werden die Mengen in 1NF dargestellt, ansonsten liegen die Mengen in der Nest-Darstellung vor.
- *duplicates*: bei *true* dürfen Duplikate in den Mengen vorkommen, bei *false* nicht.
- *type*: damit wird die Verwendung der erzeugten Daten festgelegt. Dabei sind die Werte *dividend*, *divisor* und *hd-divisor* möglich. Mit *dividend* wird ein Dividend für die Klassen *HashDivision*, *SetContainmentDivision* und *SetContainmentJoin* erzeugt, mit *divisor* ein Divisor für *SetContainmentDivision* und *SetContainmentJoin*, und mit *hd\_divisor* ein Divisor für *HashDivision*.
- *data*: ermöglicht die Angabe des Dateinamens für die erzeugten Daten.

Bei denen durch *MyDataGenerator* erzeugten Daten bestehen beim Dividend  $R(a, b)$  und beim Divisor  $S(c, d)$  bzw.  $S(c)$   $R.a$ ,  $R.b$ ,  $S.c$  und  $S.d$  aus jeweils einem Attribut, d.h. die Dividenden aus zwei Spalten und der Divisor aus zwei bzw. (bei *HashDivision*) aus einer Spalte. Außerdem sind alle Spalten vom Typ *Integer* mit Ausnahme des mengenwertigen Attributs, das im Fall *layout* = *nest* vom Typ *Array* ist, wobei der Inhalt des Arrays auch aus *Integer*-Zahlen besteht. Natürlich können unsere implementierten Algorithmen auch mit Daten, die wie in Abb. 25 eine beliebige Anzahl von Attributen besitzen, umgehen:

student_id Number	name Varchar	age Number	course_id Varchar	dozent Varchar
1	Alice	22	Compilers	Tyson
1	Alice	22	Theory	Chavez
2	Bob	27	Compilers	Tyson
2	Bob	27	Databases	Hearns
2	Bob	27	Graphics	Leonard
2	Bob	27	Theory	Chavez
3	Chris	19	Compilers	Tyson
3	Chris	19	Graphics	Leonard
3	Chris	19	Theory	Hearns

course_id Varchar	dozent Varchar	program Varchar
Compilers	Tyson	Systems
Databases	Hearns	Systems
Theory	Chavez	Systems
Compilers	Tyson	Applications
Graphics	Leonard	Applications

**Dividend**

**Divisor**

student_id Number	name Varchar	age Number	program Varchar
2	Bob	27	Systems
2	Bob	27	Applications
3	Chris	19	Applications

**Quotient**

**Abbildung 25:** Beispiel für andere Eingabedaten als durch von *MyDataGenerator* erzeugte Daten

Abb. 25 zeigt einen möglichen Dividenden und Divisor als Eingabedaten für die Klasse *SetContainmentJoin*. Dabei besitzt der Dividend drei Quotienten- (*R.a*) und zwei Divisorattribute (*R.b*), beim Divisor besteht *S.c* aus zwei und *S.d* aus einer Spalte. Die Tabelle *Quotient* stellt das durch *SetContainmentDivision* errechnete Ergebnis dar.

Werden bei *MyDataGenerator* keine Zufallszahlen verwendet (*numItems* = 0), hat der Wert von *duplicates* keine Auswirkungen auf die erzeugten Daten, da keine Duplikate generiert werden können. Außerdem sind bei *numItems* = 0 alle Mengen entweder disjunkt oder gleich. Beim Einsatz von Zufallszahlen ist der Wert von *startValue* unerheblich. Bei der Erzeugung eines Divisors für eine Hash-Division (*type* = *hd\_divisor*), ist die Angabe von *setCardinality*, *elementValues* und *layout* ohne Bedeutung, da *numGroups* und *setCardinality* hier die gleiche Bedeutung haben, nur eine Menge verwendet wird, und bei der Hash-Division der Divisor immer in der Unnest-Darstellung vorliegt.

Beispiele für erzeugte Daten sind (für *setCardinality* = 3):

A		B		C		E		F	
Tid	Item	Tid	Item	Tid	Item	Tid	Item	Item	Tid
1	1	7	1	1	[1, 2, 3]	1	12	13	1
1	2	7	2	2	[4, 5, 6]	1	2	21	1
1	3	7	3	3	[7, 8, 9]	1	73	21	1
2	4	8	1	4	[10, 11, 12]	2	41	6	2
2	5	8	2	5	[13, 14, 15]	2	15	13	2
2	6	8	3	...	...	2	8	48	2
3	7	9	1	D		3	27	7	3
3	8	9	2			3	18	75	3
3	9	9	3	Item	Tid	3	12	7	3
4	10	10	1	[47, 100, 8]	1	4	10	10	4
4	11	10	2	[47, 100, 8]	2	4	81	81	4
4	12	10	3	[47, 100, 8]	3	4	2	2	4
5	13	11	1	[47, 100, 8]	4	5	13	76	5
...	...	...	...	[47, 100, 8]	5	...	...	...	...

**Abbildung 26:** Beispiele für durch *MyDataGenerator* erzeugt Daten

Bei den Daten aus Tabelle A, B und C handelt es sich um Dividenden, wobei keine Zufallszahlen verwendet werden. In Tabelle A sind alle Mengen disjunkt, in Tabelle B alle Mengen gleich (*elementValues* = *same*). Außerdem wurde in B *startValue* = 7 gewählt. Tabelle C ist ein Dividend mit dem Wert *nest* für den Parameter *layout*. Bei Tabelle E handelt es sich ebenfalls um einen Dividenden, diesmal allerdings mit Zufallszahlen, d.h. *numItems* = 0 (hier: *numItems* = 81), sowie *duplicates* = *false* und *elementValues* = *same*. Die Tabellen D und F sind Divisoren für Set-Containment-Division und -Join (*type* = *divisor*). Dabei haben wir bei D eine Nested-Darstellung, Zufallszahlen und *elementValues* = *same*, der Wert von *duplicates* ist hier nicht ableitbar. In Tabelle F sind allerdings Duplikate erlaubt (*duplicates* = *true*). Ein Divisor für *HashDivision* besteht nur aus einer Spalte und könnte z.B. wie die Tabellen A und E ohne die erste bzw. B und F ohne die zweite Spalte aussehen.

## 7. Experimente

Die Experimente für die Klasse *HashDivision*, *SetContainmentDivision* und *SetContainmentJoin* wurden auf einem 350MHz Pentium-Prozessor und Windows 98 als Betriebssystem ausgeführt. An Hauptspeicher standen 64 MByte zur Verfügung. Die Eingabedaten für die Operanden wurden durch die Klasse *MyDataGenerator* erzeugt, wobei die Tupel aus zwei Attribute bestehen und Transaktionen darstellen sollen. Das erste Attribut repräsentiert den Transaktions-Identifier (*Tid*) und das zweite (Mengen-) Attribut den Inhalt (*Items*) der Transaktion. Es wurden keine Zufallszahlen verwendet, d.h. die Transaktionen besitzen alle die gleiche Menge oder die Mengen sind alle disjunkt zueinander. Der erste Fall (gleiche Mengen) ist gut um das Laufzeitverhalten der Algorithmen bei keinen oder vielen Ergebnistupeln zu finden, der zweite Fall überprüft das Laufzeitverhalten bei keinen oder wenigen Ergebnistupeln.

### 7.1 Experimentelle Analyse von HashDivision

Bei *HashDivision* waren bei die Erzeugung des Dividenden und Divisors durch die Klasse *MyDataGenerator* die folgenden Parameter immer identisch: *startValue* = 1, *numItems* = 0, *layout* = *unnest* und *duplicates* = *false*. Da keine Zufallszahlen verwendet wurden (*numItems* = 0), können keine Duplikate auftreten, sodass man *duplicates* auch auf *true* setzen könnte, ohne am Ergebnis etwas zu ändern. Der Parameter *type* erhielt beim Dividenden den Wert *dividend* und beim Divisor den Wert *hd\_divisor*. Für die Mengenkardinalitäten des Dividenden ( $\hat{e}_R$ ) wurden die Werte 10 und 100 gewählt, für die Mengenkardinalitäten des Divisors ( $\hat{e}_S$ ) die Werte 1, 5, 10, 50, 100. Von diesen Kardinalitäten wurden alle möglichen Kombinationen mit  $\hat{e}_R$   $\hat{e}_S$  gebildet, wobei jeweils noch die unterschiedlichen Fälle *elementValues* = *same* (Mengen identisch) oder *elementValues* *same* (Mengen unterschiedlich) betrachtet wurden. Bei *elementValues* *same*, ergab sich für  $\hat{e}_R = 10$  bei  $\hat{e}_S = 1, 5$  und 10, sowie für  $\hat{e}_R = 100$  bei  $\hat{e}_S = 1, 5, 10, 50$  und 100 (Auszug dieser Daten siehe Abb. 27) die Kurve aus Abb. 28 (nächste Seite).

Tid	Item
1	1
1	2
...	...
1	9
1	10
2	11
2	12
...	...
2	19
2	20
3	21
...	...
1000	999
1000	1000

Item
1
2
3
4
5
6
7
8
9
10

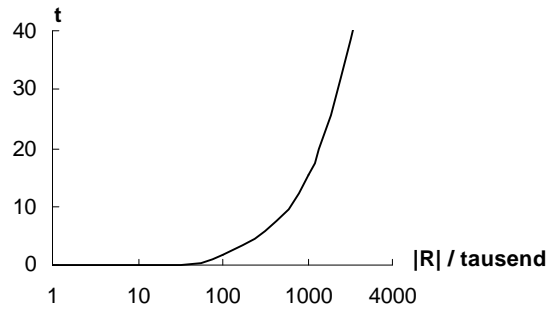
Tid	Item
1	1
1	2
...	...
1	99
1	100
2	11
2	12
...	...
2	199
2	200
3	201
...	...
100000	99999
100000	100000

Item
1
2
3
4
5
...
47
48
49
50

**Abbildung 27 a):** Dividend bei  $\hat{e}_R = 10$ , im Fall  $|R| = 10000$ , Divisor für  $\hat{e}_S = 10$

**Abbildung 27 b):** Dividend bei  $\hat{e}_R = 100$ , im Fall  $|R| = 10000000$ , Divisor für  $\hat{e}_S = 50$

$ R $	$t$
100	0s
1000	0s
10000	0,16s
100000	1,54s
1000000	15,38s
4000000	64,30s



**Abbildung 28:** Laufzeit von HashDivision für  $\epsilon_R = 10$  bzw.  $\epsilon_R = 100$  und unterschiedlichen Mengen

Sind die Mengen also disjunkt, haben die Mengenkardinalitäten von  $S$  und  $R$  keine Auswirkungen auf die Laufzeit von *HashDivision*, da jedes Mal die gleiche Anzahl von Ergebnistupeln zurückgeliefert wird (hier immer ein Ergebnistupel).

Die Laufzeit von *HashDivision* ist ungefähr proportional zu  $|R|$ , so dass sich eine Kurve mit linearer Steigung ergibt (man beachte die Skalierung der  $|R|$ -Achse in Abb. 28).

Bei *elementValues = same*, werden Dividenden der folgenden Art verwendet (Divisoren bleiben unverändert):

Tid	Item
1	1
1	2
...	...
1	9
1	10
2	1
2	2
...	...
2	9
2	10
3	1
...	...
1000	1
...	...
1000	10

Item
1
2
3
4
5
6
7
8
9
10

**Abbildung 29:** Dividend im Fall  $\epsilon_R = 10$  und  $|R| = 10000$

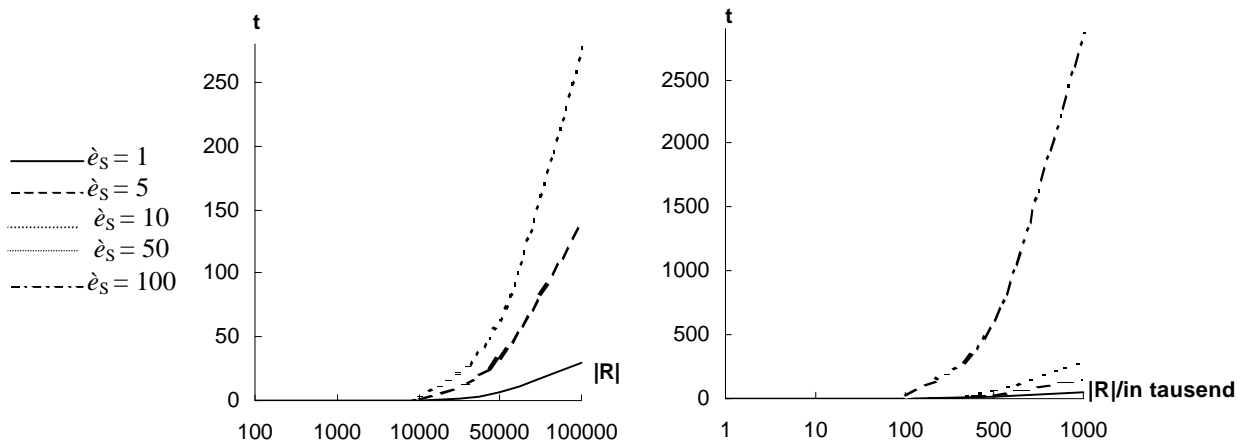
Für  $\epsilon_R = 10$  bei  $\epsilon_S = 1, 5$  und  $10$ , und für  $\epsilon_R = 100$  bei  $\epsilon_S = 1, 5, 10, 50$  und  $100$  ergaben sich folgende Werte und Kurven:

$ R $	100	1000	10000	50000	100000
$\epsilon_S = 1$	0s	0s	0,42s	6,98s	29,75s
$\epsilon_S = 5$	0s	0s	1,38s	32,35s	139,64s
$\epsilon_S = 10$	0s	0s	2,75s	62,75s	276,47s

**Abbildung 30:** Messwerte für  $\epsilon_R = 10$

$ R $	1000	10000	100000	500000	1000000
$\dot{e}_S = 1$	0s	0,17s	1,70s	13,24s	43,48s
$\dot{e}_S = 5$	0s	0,17s	2,64s	38,28s	155,50s
$\dot{e}_S = 10$	0s	0,19s	3,83s	68,10s	302,66s
$\dot{e}_S = 50$	0s	0,29s	14,60s	301,10s	1428s
$\dot{e}_S = 100$	0s	0,43s	25,17s	609,95s	2857,5s

**Abbildung 31:** Messwerte für  $\dot{e}_R = 100$



**Abbildung 32a):** Laufzeit bei  $\dot{e}_R = 10$

**Abbildung 32b):** Laufzeit bei  $\dot{e}_R = 100$

Aus diesen Messwerten kann abgeleitet werden (vor allem bei den Werten die sich bei größerem  $|R|$  ergeben), dass die Laufzeit von *HashDivision* bei einer Vergrößerung von  $|R|$  um den Faktor 5 bzw. 10, 25-mal bzw. 100-mal länger wird, d.h.  $t$  ist proportional zu  $|R|^2$ . Außerdem erkennt man, dass *HashDivision* bei doppelter Mengenkardinalität von  $S$  doppelt so viel Zeit benötigt. Folglich ergibt sich, dass  $t$  proportional zu  $|R|^2$  und  $\dot{e}_S$  ist.

Der Einfluss von  $\dot{e}_R$  auf  $t$  ist geringer. So ergeben sich in Abb. 32b) mit 10-mal größerer Mengenkardinalität von  $R$  bei größerem  $|R|$  eine Erhöhung der Laufzeit um 10 – 20%.

## 7.2 Experimentelle Analyse von SetContainmentDivision

Bei *SetContainmentDivision* wurden bei der Erzeugung des Dividenden und Divisors durch die Klasse *MyDataGenerator* die gleichen Parameter wie bei *HashDivision* verwendet. Einzige Ausnahme war der Parameter *type* der beim Divisor auf den Wert *divisor* gesetzt wurde. Außerdem entspricht beim Divisor *numGroups* nicht mehr *set-Cardinality*, sodass ein weiterer Parameter betrachtet werden musste. Für die Anzahl der Gruppen im Divisor wurden im Rahmen dieser Studienarbeit für die Experimente der Fall *numGroups* = 10 angenommen. Der Wert von *elementValues* ist beim Divisor immer ungleich *same*, da die Verwendung identischer Mengen im Divisor keinen Sinn machen würde. Zusätzlich kommt beim Konstruktor der Klasse *SetContainmentDivision* noch der Parameter für den Divisionsoperator hinzu, sodass bei den

Messungen noch zwischen der Verwendung von *HashDivision* und *MySortBasedDivision* als Divisionsoperator unterschieden werden musste. Ist beim Dividenten *elementValues* *same*, wurden z.B. folgende Daten verwendet (siehe Abb. 33):

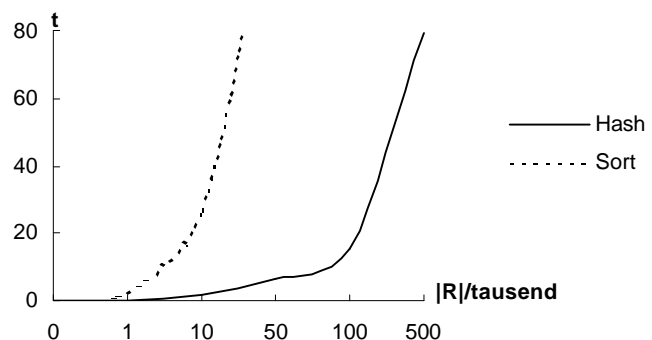
Tid	Item	Item	Tid	Tid	Item	Item	Tid
1	1	1	1	1	1	1	1
1	2	2	1	1	2	2	1
...	...	3	1	...	...	...	...
1	9	4	1	1	99	100	1
1	10	5	1	1	100	101	2
2	11	6	2	2	101	102	2
2	12	7	2	2	102	...	...
...	...	8	2	...	...	200	2
2	19	9	2	2	199	201	3
2	20	10	2	2	200	...	...
3	21	11	3	3	201	901	10
...	...	...	...	...	...	...	...
1000	999	49	10	100000	99999	999	10
1000	1000	50	10	100000	100000	1000	10

**Abbildung 33 a):** Divident bei  $\epsilon_R = 10$ , im Fall  $|R| = 10000$ , Divisor für  $\epsilon_S = 5$  und  $|S| = 50$

**Abbildung 33 b):** Divident bei  $\epsilon_R = 100$ , im Fall  $|R| = 10000000$ , Divisor für  $\epsilon_S = 100$

Für  $\epsilon_R = 10$  mit  $\epsilon_S = 1, 5$  und  $10$ , sowie für  $\epsilon_R = 100$  mit  $\epsilon_S = 1, 5, 10, 50$  und  $100$  ergeben sich folgende Messergebnisse:

$ R $	$t_{hash}$	$t_{sort}$
100	0s	0s
1000	0,176s	2,64s
10000	1,59s	26,8s
50000	6,60s	133,3s
100000	15,61s	259,38s
500000	79,15s	1287s



**Abbildung 34:** Laufzeit von *SetContainmentDivision* für  $\epsilon_R = 10$  bzw.  $\epsilon_R = 100$  und unterschiedlichen Mengen bei Verwendung von *HashDivision* bzw. *MySortBasedDivision*

Bei unterschiedlichen Mengen hat die Mengenkardinalität von *S* keine Auswirkungen auf die Laufzeit von *SetContainmentDivision*, da trotz gleicher Anzahl von erzeugten Ergebnistupeln (hier immer zehn Ergebnistupel) *t* bei steigendem  $\epsilon_S$  konstant bleibt. Da die Laufzeit von *SetContainmentDivision* beim Anwachsen der Relation *R* mit dem entsprechenden Faktor steigt, kann man davon ausgehen, dass *t* proportional zu  $|R|$  ist. Erstaunlich ist, dass die Laufzeit von *SetContainmentDivision* bei Verwendung von *MySortBasedDivision* ziemlich genau 15-mal länger als bei der Verwendung von *HashDivision* ist.

Vergleicht man Abb. 34 mit den Messergebnissen für *HashDivision* bei ungleichen Mengen (Abb. 28), so fällt auf, dass die Werte für *SetContainmentDivision* ca. 10-mal

höher sind. Da bei *SetContainmentDivision*  $numGroups = 10$  gewählt wurde, so dass 10 Divisionen durchzuführen sind, und bei einer Hash-Division immer nur eine Gruppe bzw. Menge vorhanden ist, kann dieses Ergebnis auch nicht sonderlich überraschen. Folglich ergibt sich die Laufzeit bei *SetContainmentDivision* bei Verwendung des Divisionsoperators *HashDivision* aus der Anzahl der Gruppen im Divisor ( $numGroups$ ) multipliziert mit der Laufzeit, die *HashDivision* mit demselben Dividenden und einer Menge als Divisor benötigen würde.

Wird beim Dividenden der Parameter *elementValues* auf den Wert *same* gesetzt, erhält man folgende Daten (Divisoren bleiben unverändert wie in Abb. 33):

Tid	Item	Item	Tid
1	1	1	1
1	2	2	1
...	...	3	1
1	9	4	1
1	10	5	1
2	1	6	2
2	2	7	2
...	...	8	2
2	9	9	2
2	10	10	2
3	1	11	3
...	...	...	...
1000	1	49	10
...	...	50	10
1000	10		

**Abbildung 35:** Dividend im Fall  $\epsilon_R = 10$  und  $|R| = 10000$

Bei  $\epsilon_R = 10$  ergaben sich für  $\epsilon_S = 1, 5$  und  $10$  jeweils ungefähr gleiche Laufzeiten:

$ R $	100	1000	10000	50000
$t_{hash}$	0s	0,27s	3,82s	71,42s
$t_{sort}$	0s	2,8s	27,9s	139,34s

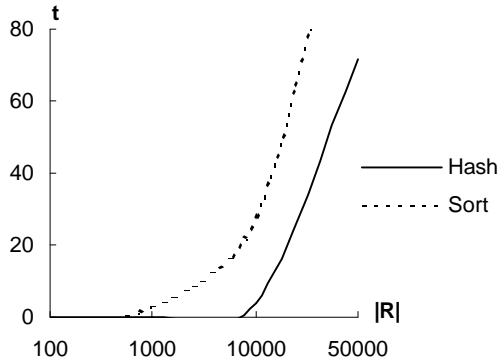
**Abbildung 36:** Messwerte für  $\epsilon_R = 10$

Bei  $\epsilon_R = 100$  wurden bei der Verwendung von *HashDivision* folgende Werte für  $\epsilon_S = 1, 5, 10, 50$  und  $100$  gemessen:

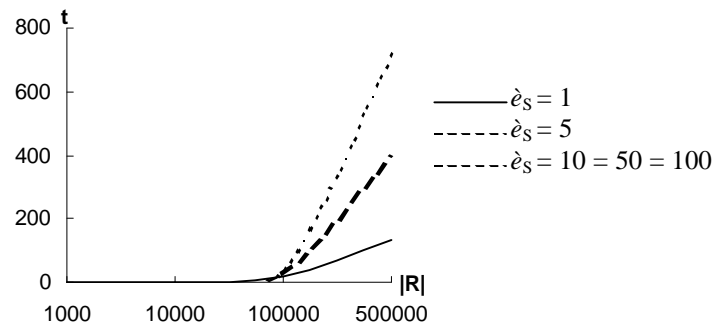
$ R $	1000	10000	100000	500000
$\epsilon_S = 1$	0s	1,85s	15,94s	134,82s
$\epsilon_S = 5$	0s	1,48s	25,90s	393,10s
$\epsilon_S = 10 = 50 = 100$	0s	1,72s	38,50s	714,96s

**Abbildung 37:** Messwerte für  $\epsilon_R = 100$  und Einsatz von *HashDivision*

Aus diesen Werten ergaben sich dann folgende Kurvenverläufe für die Laufzeit von *SetContainmentDivision*:



**Abbildung 38a):** Laufzeit bei  $\hat{e}_R = 10$



**Abbildung 38b):** Laufzeit bei  $\hat{e}_R = 100$  und Einsatz von HD

In Abb. 38 a) bleibt die Laufzeit für  $\hat{e}_S = 1, 5$  und  $10$  in allen drei Fällen gleich, da gleichzeitig die Anzahl der generierten Tupel um den gleichen Faktor sinkt ( $10000, 2000, 1000$  bei  $\hat{e}_S = 1, 5$  bzw.  $10$ ), sodass sich die Faktoren gegenseitig aufheben. Für  $|R|$  lässt sich wieder eine proportionale Abhängigkeit zu  $t$  ableiten (besonders gut erkennbar bei den Messergebnissen mit *MySortBasedDivision* als Divisionoperator). Wie in Abb. 34 ist auch hier die Laufzeit von *SetContainmentDivision* beim Einsatz von *MySortBasedDivision* höher, als bei der Verwendung von *HashDivision*, allerdings wird der (prozentuale) Unterschied bei steigendem  $|R|$  geringer.

Bei  $\hat{e}_R = 100$  in Abb. 38b) erhöht sich der Einfluss von  $\hat{e}_S$ , wie die Ergebnisse zeigen. Trotz gleichbleibendem Wert von  $|R|$  und gleicher Anzahl von Ergebnistupeln, erhöht sich bei  $\hat{e}_S = 1, 5$  und  $10$  jeweils die Laufzeit  $t$ . Die Kurven für  $\hat{e}_S = 10, 50$  und  $100$  sind nahezu identisch, da sich die Mengenkardinalität von  $S$  zuerst verfünffacht und anschließend verdoppelt, während die Anzahl der Ergebnistupel sich um den Faktor fünf und den Faktor zwei verringert ( $10000, 2000$  und  $1000$  Tupel bei  $\hat{e}_S = 10, 50$  bzw.  $100$ ), d.h.  $\hat{e}_S$  und die Anzahl der generierten Ergebnistupel haben ungefähr die gleichen Auswirkungen auf die Laufzeit von *SetContainmentDivision*. Auf die Darstellung von  $t$  bei der Verwendung von *MySortBasedDivision* wurde aus Gründen der Übersichtlichkeit verzichtet.

Außerdem wurden bei *SetContainmentDivision* noch der zeitliche Anteil der Selektion, Projektion, Hash-Division und des kartesischen Produkts an der (Gesamt-) Laufzeit ermittelt. Dabei stellte sich heraus, dass für Selektion, Projektion und kartesisches Produkt jeweils 0 oder wenige Millisekunden ( $< 50$  ms) gemessen wurde, während die Durchführung von *HashDivision* nahezu 100% der Laufzeit von *SetContainmentDivision* ausmachte.

### 7.3 Experimentelle Analyse von SetContainmentJoin

Für die Klasse *SetContainmentJoin* wurden bei der Erzeugung des Dividenden und Divisors durch die Klasse *MyDataGenerator* die gleichen Parameter wie bei *SetContainmentDivision* verwendet. Einzige Ausnahme ist der Parameter *layout*, dessen Wert ungleich *unnest* sein muss. Außerdem kann man bei *SetContainmentJoin* die Anzahl der Partitionen variieren.

Ist beim Dividenden *elementValues same*, wurden z.B. folgende Daten verwendet (siehe Abb. 39):

Tid	Item	Item	Tid
1	[1, 2, 3, 4, ..., 9, 10]	[1, 2, ..., 5]	1
2	[11, 12, 13, 14, ..., 19, 20]	[6, 7, ..., 10]	2
3	[21, 22, 23, 24, ..., 29, 30]	[11, 12, ..., 15]	3
...	...	...	...
10000	[9991, 9992, 9993, ..., 9999, 10000]	[46, 47, ..., 50]	10

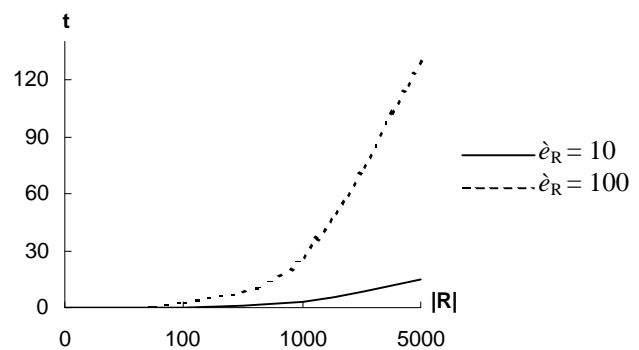
**Abbildung 39 a):** Dividend bei  $\hat{e}_R = 10$ , im Fall  $|R| = 10000$ , Divisor für  $\hat{e}_S = 5$

Tid	Item	Item	Tid
1	[1, 2, 3, 4, ..., 99, 100]	[1, 2, ..., 100]	1
2	[101, 102, ..., 199, 200]	[101, 102, ..., 200]	2
3	[201, 202, ..., 299, 300]	[201, 202, ..., 300]	3
...	...	...	...
100000	[99901, 99902, ..., 99999, 100000]	[901, 902, ..., 1000]	10

**Abbildung 39 b):** Dividend bei  $\hat{e}_R = 100$ , im Fall  $|R| = 100000$ , Divisor für  $\hat{e}_S = 100$

Da  $numGroups = 10$  gilt, hat der Divisor in allen Fällen die Länge 10 ( $|S| = 10$ ). Für  $\hat{e}_R = 10$  mit  $\hat{e}_S = 1, 5$  und  $10$ , sowie für  $\hat{e}_R = 100$  mit  $\hat{e}_S = 1, 5, 10, 50$  und  $100$  ergeben sich folgende Messergebnisse:

$ R $	$t_{\hat{e}=10}$	$t_{\hat{e}=100}$
0	0s	0s
100	0,33s	3,16s
1000	3,60s	26,03s
5000	14,95s	130s



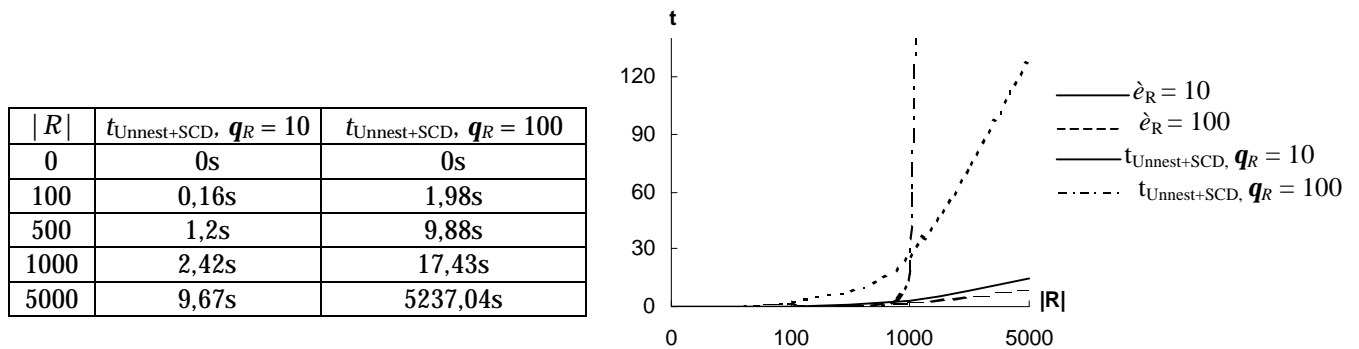
**Abbildung 40:** Laufzeit von *SetContainmentJoin* für  $\hat{e}_R = 10$  bzw.  $\hat{e}_R = 100$  und unterschiedlichen Mengen

Ist *elementValues same*, so hat die Mengenkardinalität von  $S$  wie bei *SetContainmentDivision* keine Auswirkungen auf die Laufzeit von *SetContainmentJoin*, während bei  $\hat{e}_R = 100$  die Laufzeit ca. 10-mal höher ist als bei  $\hat{e}_R = 10$ .

Wird die Anzahl der Zeilen in der Relation  $R$  verfünffacht bzw. verzehnfacht, wird auch  $t$  fünf- bzw. zehnmals größer, d.h.  $t$  ist, wie bei den anderen implementierten Klassen auch, proportional zu  $|R|$ .

Wenn man Abb. 40 mit den Messergebnissen für *SetContainmentDivision* bei ungleichen Mengen (Abb. 34) und Verwendung von *HashDivision* in Zusammenhang bringen will, so muss man die Werte von  $|R|$  in Abb. 40 mit  $\hat{e}_R$  multiplizieren, da dies  $|R|$  für *SetContainmentDivision* bei Verwendung der gleichen Daten ergibt. Dabei fällt auf, dass die Werte für *SetContainmentJoin* bei  $|R| = 100, 1000, 5000$  und  $\hat{e}_R = 10$  ca. doppelt so hoch (100%), und bei  $|R| = 1000$  bzw.  $5000$  und  $\hat{e}_R = 100$  um ca. 65% höher sind. Die folgenden Experimente wurden durchgeführt, um festzustellen, ob bei Eingabedaten in Nested-Darstellung, durch die Anwendung des *Unnest*-

Operators mit anschließender Set-Containment-Division die Gesamtlaufzeit immer noch kürzer ist, als bei sofortiger Durchführung von *SetContainmentJoin*:



**Abbildung 41:** Vergleich der Laufzeiten von SCJ mit SCD+Unnest

Verwendet man beim Dividenden immer identische Mengen (*elementValues = same*), kommen folgende Daten (Divisoren bleiben unverändert wie in Abb. 39) zum Einsatz:

Tid	Item
1	[1, 2, 3, ..., 9, 10]
2	[1, 2, 3, ..., 9, 10]
3	[1, 2, 3, ..., 9, 10]
...	...
10000	[1, 2, 3, ..., 9, 10]

Item	Tid
[1, 2, ..., 5]	1
[6, 7, ..., 10]	2
[11, 12, ..., 15]	3
...	...
[46, 47, ..., 50]	10

**Abbildung 42:** Dividend im Fall  $e_R = 10$  und  $|R| = 10000$

Bei  $e_R = 10$  ergaben sich für  $e_S = 1, 5$  und  $10$  folgende Werte:

$ R $	0	100	1000	5000
$e_S = 1$	0s	0,33s	3,50s	14,46s
$e_S = 5$	0s	0,28s	2,63s	12,08s
$e_S = 10$	0s	0,27s	1,92s	9,39s

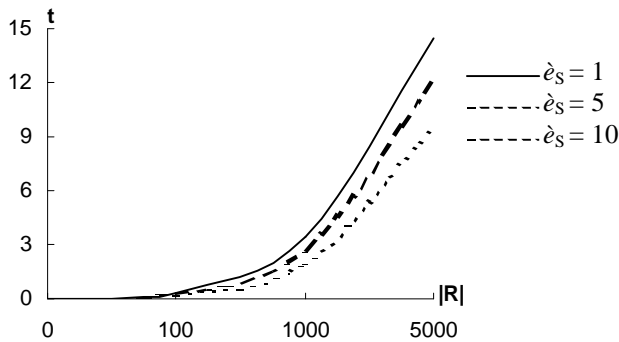
**Abbildung 43:** Messwerte für  $e_R = 10$

Bei  $e_R = 100$  wurden folgende Werte für  $e_S = 1, 5, 10, 50$  und  $100$  gemessen:

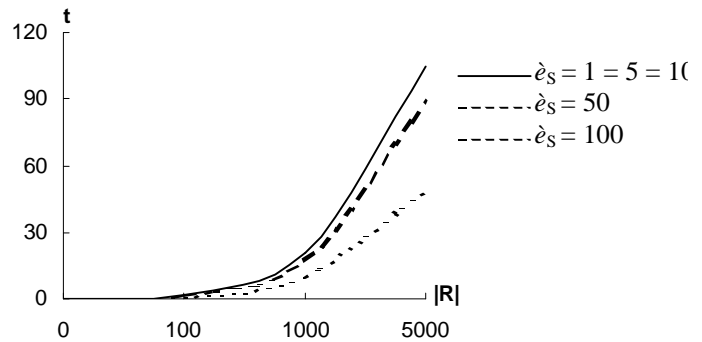
$ R $	0	100	1000	5000
$e_S = 1 = 5 = 10$	0s	2,20s	21,13s	104,70s
$e_S = 50$	0s	1,88s	17,47s	88,32s
$e_S = 100$	0s	1,27s	9,83s	48,67s

**Abbildung 44:** Messwerte für  $e_R = 100$

Aus diesen Werten ergaben sich dann folgende Kurvenverläufe für die Laufzeit von *SetContainmentJoin*:



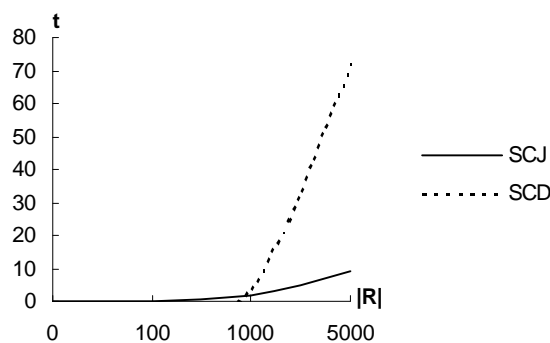
**Abbildung 45a):** Laufzeit bei  $e_R = 10$



**Abbildung 45b):** Laufzeit bei  $e_R = 100$

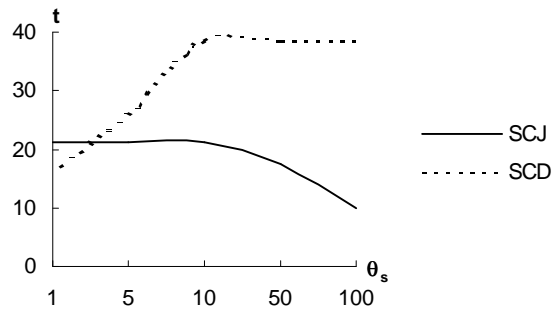
In Abb. 45 a) wird die Laufzeit von *SetContainmentJoin* bei steigendem  $e_S$  niedriger, da die Anzahl der Ergebnistupel mit steigendem  $e_S$  um den gleichen Faktor abnimmt. Folglich hat in diesem Fall die Anzahl der generierten Tupel im Gegensatz zum entsprechenden Fall bei *SetContainmentDivision*, einen größeren Einfluß auf  $t$  als  $e_S$ . Wie in allen anderen Messungen zuvor ist auch hier  $t$  proportional zu  $|R|$ . Bei  $e_R = 100$  in Abb. 45b) bleiben für  $e_S = 1, 5$  und  $10$  die Werte (fast) identisch, obwohl hier in allen drei Fällen die Anzahl der erzeugten Tupel gleich bleibt, d.h.  $e_S$  hat auch hier so gut wie keine Auswirkungen, während es bei *SetContainmentDivision* zu einer Erhöhung der Laufzeit kam. Für  $e_S = 10, 50$  und  $100$  ergeben sich unterschiedliche Kurven. Da die Anzahl der Ergebnistupel bei steigendem  $e_S$  im gleichen Maße sinkt und die Laufzeit im Gegensatz zu *SetContainmentDivision* ebenfalls geringer wird, folgt, dass  $e_S$  bei *SetContainmentJoin* einen geringeren Einfluss auf  $t$  hat wie bei *SetContainmentDivision*.

Vergleicht man die gemessenen Werte von *SetContainmentJoin* und *SetContainmentDivision* für diesen Fall direkt miteinander (wiederum muss beim Vergleich der Zahlenwerte bei *SetContainmentJoin* die Mengenkardinalität von  $R$  mit  $|R|$  multipliziert werden), so fällt auf, dass *SetContainmentJoin* bei steigendem  $|R|$  und steigendem  $e_S$  deutlich effizienter als *SetContainmentDivision* ist. Die folgende Abbildung zeigt dies für den Fall  $e_S = 10$  bei  $e_R = 10$ :



**Abbildung 46:** Vergleich SCJ mit SCD bei  $e_S = 10$  und  $e_R = 10$

Die nächste Abbildung vergleicht *SetContainmentJoin* und *SetContainmentDivision* in Abhängigkeit von  $e_S$  für den Fall  $|R| = 1000$ :



**Abbildung 47:** Vergleich SCJ mit SCD in Abhängigkeit von  $\hat{\epsilon}_S$  und bei  $|R| = 1000$

Wie man sieht, wird die Laufzeit ab einem gewissen  $\hat{\epsilon}_S$  (hier bei ca.  $\hat{\epsilon}_S = 2,5$ ) bei *SetContainmentJoin* geringer als bei *SetContainmentDivision*. Wird  $|R|$  größer, driften die Kurven immer weiter auseinander.

## 8. Zusammenfassung

Im ersten Kapitel wurden in dieser Studienarbeit die wesentlichen Charakteristika der drei Operatoren relationale Division, Set-Containment-Division (SCD) und Set Containment Join (SCJ) beschrieben. Die relationale Division und die Set-Containment-Division arbeiten auf Daten in 1NF, wobei bei der relationalen Division durch eine Gruppe bzw. Menge und bei SCD durch mehrere Gruppen bzw. Mengen geteilt wird. SCJ kann wie SCD mehrere Gruppen handhaben, verarbeitet aber Daten, die nicht in 1NF vorliegen sondern eine Nested-Darstellung haben.

Das zweite Kapitel beschreibt die Eigenschaften der relationalen Division genauer, und stellt Algorithmen vor, welche diese Operation realisieren. Von diesen Algorithmen werden die Merge-Sort-Division und die Hash-Division detailliert beschrieben. Bei der Merge-Sort-Division müssen der Dividend bzw. Divisor in einer sortierten Form vorliegen, während bei der Hash-Division die Tupel im Dividenden bzw. Divisor in einer beliebigen Reihenfolge auftreten dürfen.

Im dritten Kapitel werden für den Set Containment Join die Algorithmen Partitioning Set Join (PSJ) und Adaptive Pick-and-Sweep Join beschrieben, das vierte Kapitel beschäftigt sich mit der Set-Containment-Division, wobei eine formale Definition dieses Operators angegeben wird.

Eine Analyse der Algorithmen der relationalen Division und von SCJ wird im fünften Kapitel durchgeführt. Bei den Algorithmen für die relationale Division hat der Merge-Sort-Division-Algorithmus Vorteile gegenüber der Hash-Division bei der Speicherkomplexität, während die Hash-Division bei der Zeitkomplexität trumpfen kann. Bei SCJ werden Vergleichs- und Replikationsfaktoren als Effizienzmaß verwendet. Dabei stellt sich heraus, dass PSJ bei Eingabedaten, die kleinere Mengen haben ( $< 10$  Elemente), zum Einsatz kommen soll und APSJ bei größeren Mengen. Im sechsten Kapitel wird die Java-Bibliothek XXL mit seinen wichtigsten Konzepten beschrieben. Diese Bibliothek kann eingesetzt werden, um die Funktionalität eines Datenbanksystems zu erweitern. Anschließend erfolgt eine Beschreibung der Klassen, die im Rahmen dieser Studienarbeit mit Hilfe von XXL implementiert wurden, um die Operatoren für die relationale Division, SCD und SCJ zu realisieren.

Das siebte Kapitel beschreibt die Tests, die auf den realisierten Operanden ausgeführt wurden. Dabei stellte sich heraus, dass der entwickelte SCD-Algorithmus im direkten Vergleich mit SCJ nur bei kleinen Mengenkardinalitäten ( $< 5$ ) für die Eingabedaten besser als SCJ abschneidet. Bei einem Wachstum der Kardinalitäten ergeben sich allerdings schnell erhebliche Laufzeitvorteile von SCJ gegenüber SCD.

Bei zukünftigen Arbeiten müssen neben weiteren Tests auf größeren Datenmengen, Konzepte zur Verbesserung des SCD-Algorithmus, der in dieser Studienarbeit nach einem einfachen Ansatz entsprechend der Definition implementiert wurde, realisiert werden, damit diese Operation auch für große Mengen effizient ist.

## Literaturverzeichnis

- [1] R. Rantzaou, L. Shapiro, B. Mitschang, Q. Wang: *Universal Quantification in Relational Databases: A Classification of Data and Algorithms*, Computer Science Department, University of Stuttgart
- [2] G. Graefe, R. Cole: *Fast Algorithms for Universal Quantification in Large Databases*. ACM Transactions on Database Systems, Volume 20, Number 2, 1995.
- [3] S. Melnik, H. Garcia-Molina: *Adaptive Algorithms for Set Containment Joins*. Technical Report, Department of Computer Science, Stanford University, CA, USA, November 2001.
- [4] S. Helmer, G. Moerkotte: *Evaluation of Main Memory Join Algorithms for Joins with Subset Join Predicates*. Proceedings VLDB, Athens, Greece, August 1997.
- [5] S. Chaudhuri, G. Weikum: *Rethinking Database System Architecture: Towards a Self-Tuning risc-style Database System*. In VLDB, 2000.
- [6] J. Bercken, B. Blohsfeld, J.-P. Dittrich, J. Krämer, T. Schäfer, M. Schneider, B. Seeger: *XXL – A Library Approach to Supporting Efficient Implementations of Advanced Database Queries*. Proceedings VLDB, Roma, Italy, 2001.

## **Erklärung**

Hiermit versichere ich, diese Arbeit  
selbständig verfasst und nur die  
angegebenen Quellen benutzt zu haben.

- Rudi Husser -