

**Studiengang:** Informatik

**Prüfer:** Prof. Dr.-Ing. Bernhard Mitschang

**Betreuer:** Dipl.-Inf. Ralf Rantza

**begonnen am:** 15. November 2002

**beendet am:** 20. Juni 2003

**CR-Klassifikation:** H.2.4, H.2.8

Diplomarbeit Nr. 2061

## **Frequent Itemset Discovery Using SQL and Relational Division**

Daniel Böck

Institut für Parallele und  
Verteilte Systeme  
Universität Stuttgart  
Universitätsstraße 38  
D-70569 Stuttgart



Für meine Eltern



## Inhaltsverzeichnis

<b>1</b>	<b>Einleitung</b>	<b>3</b>
<b>2</b>	<b>Aufgabenstellung</b>	<b>5</b>
<b>3</b>	<b>Charakteristika der Transaktionsdaten</b>	<b>7</b>
<b>4</b>	<b>Frequent Itemset Discovery mit SQL</b>	<b>11</b>
4.1	Horizontales Layout . . . . .	14
4.1.1	Kandidatentabelle . . . . .	14
4.1.2	K-way Join . . . . .	15
4.1.3	Subquery . . . . .	16
4.1.4	Subquery mit materialisierten Zwischenergebnissen . . . . .	16
4.1.5	Set-oriented Apriori . . . . .	17
4.2	Anpassung der Algorithmen . . . . .	18
4.3	Vertikales Layout . . . . .	19
4.3.1	Universal Quantification . . . . .	19
4.3.2	K-Way Join vertikal . . . . .	22
<b>5</b>	<b>Frequent Itemset Discovery mit XXL</b>	<b>25</b>
5.1	XXL . . . . .	25
5.2	Übersicht der Operatoren . . . . .	26
5.3	K-Way Join in Java . . . . .	27
5.4	Quiver in Java . . . . .	29
5.5	Division in Java . . . . .	31
5.6	Wichtige Klassen . . . . .	31
5.6.1	BufferedFileMetaDataCursor . . . . .	31
5.6.2	MyBTree . . . . .	32
5.6.3	MyBTreeQueryCursor . . . . .	32
5.6.4	NestedLoopsAntiSemiJoin . . . . .	33
5.6.5	HashJoin . . . . .	33
5.6.6	MySingleObjectCursor . . . . .	34
5.6.7	XXL - NestedLoopsJoin . . . . .	34
5.6.8	XXL - MergeSorter . . . . .	35
<b>6</b>	<b>Messungen</b>	<b>37</b>
6.1	SQL - Ergebnisse . . . . .	37
6.2	XXL - Ergebnisse . . . . .	45
6.2.1	Synthetische Daten . . . . .	46
6.2.2	Reale Anwendungsdaten . . . . .	49
6.3	Vergleich SQL - XXL . . . . .	51
<b>7</b>	<b>Zusammenfassung und Ausblick</b>	<b>53</b>

<b>A</b>	<b>SQL-Quellcode</b>	<b>55</b>
A.1	Candidate-Generation . . . . .	55
A.2	K-Way Join . . . . .	55
A.3	Subquery . . . . .	56
A.4	Subquery-modified . . . . .	57
A.5	Set Oriented Apriori . . . . .	58
A.6	Quiver . . . . .	58
<b>B</b>	<b>XXL-Quellcode</b>	<b>63</b>
B.1	SQL Ausführungsplan für K-Way Join . . . . .	63
B.2	XXL Quelltext von K-Way Join Algorithmus . . . . .	63
B.3	SQL Ausführungsplan für Quiver . . . . .	64
B.4	XXL Quelltext von Quiver Algorithmus . . . . .	65
B.5	XXL Quelltext von der Klasse HashJoin . . . . .	66
B.6	XXL Quelltext von der Klasse NestetLoopAntiSemiJoin . . . . .	69
B.7	SQL - Messergebnisse im Überblick . . . . .	72
B.8	XXL - Messergebnisse im Überblick . . . . .	72
	B.8.1 synthetische Datentabellen . . . . .	72
	B.8.2 Real life Datentabellen . . . . .	73
<b>C</b>	<b>Abbildungsverzeichnis</b>	<b>75</b>
<b>D</b>	<b>Tabellenverzeichnis</b>	<b>77</b>
	<b>Literaturverzeichnis</b>	<b>78</b>

# 1 Einleitung

Frequent Itemset Discovery (FID) ist frei übersetzt die Suche nach Begriffen, die in einer Datenbank oft zusammen vorkommen. Als Beispiel seien die Einkaufszettel in Abbildung 1 in einem Supermarkt angeführt. Auf Kassenbon 1 steht eine bestimmte Anzahl verschiedener Waren, ebenso auf Kassenbon 2 und 3. Nun werden die Waren gesucht, die häufig zusammen auf einem Bon stehen.

Einkaufszettel 1	Einkaufszettel 2	Einkaufszettel 3
Bleistift	Chips	Tomaten
Radiergummi	Salzletten	Gurken
Füller	Schokolade	Chips
Kugelschreiber	Erdnüsse	Erdnüsse
Tinte	Kugelschreiber	Kugelschreiber
Papier	Papier	Papier
		Bleistift

Abbildung 1: Beispiel - Drei Kassenbons

Es werden zuerst die einzelnen Waren ermittelt, die häufig vorkommen. Unter häufig versteht man einen Schwellwert, der überschritten werden muss. Hier im Beispiel sind dies einmal 50 Prozent. Sobald ein Artikel auf mehr als 50 Prozent aller Einkaufszettel vorkommt, wird er als häufig angesehen. Also ergeben sich folgende Waren: Kugelschreiber, Papier, Bleistift, Erdnüsse und Chips. Als nächstes werden die Zweierkombinationen untersucht. Es ergibt sich folgendes Ergebnis: Kugelschreiber - Papier, Kugelschreiber - Bleistift, Kugelschreiber - Chips, Kugelschreiber - Erdnüsse, Chips - Erdnüsse, Chips - Papier, Erdnüsse - Papier. Als einzige Dreierkombination gibt es dann noch Erdnüsse - Kugelschreiber - Papier. Man erkennt, dass diese Ergebnisse bei großen Datenmengen schnell unübersichtlich werden und schwer zu bestimmen sind. Daher wird auf Algorithmen zurückgegriffen, welche die oben genannte Berechnung (FID) übernehmen.

Werden die Daten von einem relationalen Datenbankverwaltungssystem (DBVS) verwaltet, so lassen sich einige SQL-Algorithmen aus der Literatur für dieses Problem einsetzen. In den meisten Fällen wird die Lösung in zwei Teile aufgeteilt:

- Kandidatentabellengenerierung: Hierbei werden alle möglichen Begriffe, die überhaupt in Betracht kommen können, in eine Tabelle geschrieben.
- Zählphase: Dies ist der aufwändige und zeitraubende Teil. Es werden der Reihe nach alle Kandidaten auf Ihre Häufigkeit in der Transaktionstabelle hin überprüft.

Dies wird sequentiell für jeden einzelnen Durchgang wiederholt. Will man zum Beispiel die 3 häufig gemeinsam auftretenden Begriffe in einer Transaktionstabelle finden, so wird

erst die Kandidatentabelle für alle einzelnen Begriffe gebildet. Danach kommt die Zählphase und es werden nur die häufigen Begriffe ausgewählt. Anschließend werden hieraus die Begriffspaare, die gemeinsam vorkommen, gebildet, dann wieder gezählt. Schließlich die Kandidaten für die drei Begriffe, die gemeinsam vorkommen, gebildet und danach die Häufigen selektiert.

In dieser Arbeit werden die oben genannten Algorithmen einem neuen Ansatz gegenübergestellt. Dieser neue Ansatz heißt Quiver, basiert auf der Division und stammt von Ralf Rantau, Institut für Parallele und Verteilte Systeme, Universität Stuttgart. Da kein DBVS einen Divisionsoperator bereitstellt, werden sehr schlechte Ergebnisse erwartet. Daher wurden sowohl der Algorithmus K-Way Join, als auch der neue Ansatz Quiver in Java implementiert. Dazu werden die Ausführungspläne der Algorithmen, die durch das DBVS erstellt werden, untersucht und diese Vorgehensweise in Java implementiert. Nun wird der neue Ansatz Quiver mit der Division in Java realisiert, und den anderen Algorithmen gegenübergestellt werden.

## 2 Aufgabenstellung

Die Ausarbeitung ist aufgeteilt in 3 Hauptteile. Der erste Teil befasst sich mit der Implementierung von bekannten Algorithmen für FID in SQL. Danach wird der neue Algorithmus Quiver erläutert und ebenfalls in SQL implementiert. Im zweiten Teil werden wichtige SQL-Statements in XXL, eine Klassenbibliothek für Java, implementiert und auf signifikante Unterschiede zu der SQL-Implementierung hin untersucht. Im dritten Teil werden die Algorithmen mit verschiedenen Daten getestet, die Ergebnisse präsentiert und mögliche Schlussfolgerungen gezogen.

Im folgenden Kapitel wird zuerst auf die SQL-Implementierung der Algorithmen und deren Ergebnisse eingegangen. Anschließend wird analog die XXL-Implementierung diskutiert. Im nächsten Kapitel werden die Messungen präsentiert und die Schlussfolgerungen gezogen. Schließlich folgt die Zusammenfassung der ganzen Arbeit und ein Ausblick.



### 3 Charakteristika der Transaktionsdaten

Um die Untersuchungen durchzuführen, werden Daten benötigt, in denen nach den frequent Itemsets gesucht wird. Diese Transaktionstabellen sind vertikal aufgebaut und enthalten die zwei Spalten Transaktionsidentifikationsnummer (TID) und die dazugehörigen Begriffe (Items). Es wurden synthetische Tabellen generiert, die zum einen unterschiedlich viele Transaktionen und eine variierende Anzahl an der durchschnittlichen Itemanzahl pro Transaktion enthalten, zum anderen einen unterschiedlichen Verlauf der Anzahl der Items über alle Transaktionen hinweg vorweisen. Folgende Abbildungen 2 und 3 sollen dies veranschaulichen:

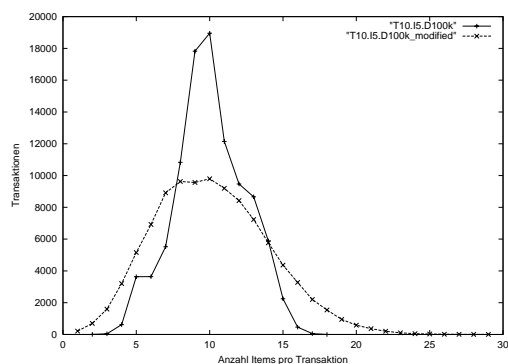


Abbildung 2: Transaktionstabelle T10.I5.D100k und T10.I5.D100k\_modified

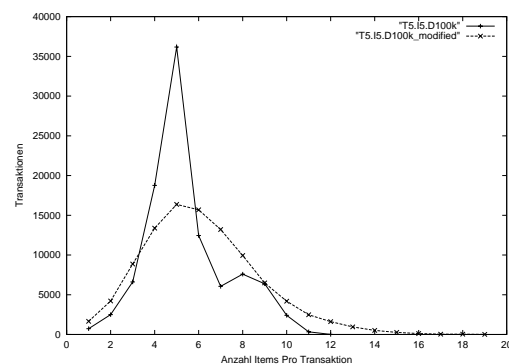


Abbildung 3: Transaktionstabelle T5.I5.D100k und T5.I5.D100k\_modified

Die vier abgebildeten Tabellen (T10.I5.D100k, T10.I5.D100k\_modified, T5.I5.D100k und T5.I5.D100k\_modified) enthalten jeweils ungefähr 100 000 Transaktionen. Um einen groben Anhaltspunkt für die Tabellengröße zu bekommen, kann man den Wert mit der “average itemsize” multiplizieren. Das ergäbe dann zweimal ungefähr 500 000 Einträge und zweimal 1 000 000 Einträge, wobei dieser Wert gut bis zu 30 Prozent abweichen kann. Der Grund für diese starken Abweichung wird weiter unten erklärt.

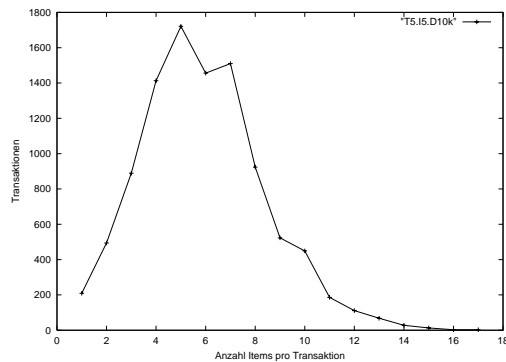


Abbildung 4: kleine Transaktionstabelle T5.I5.D10k

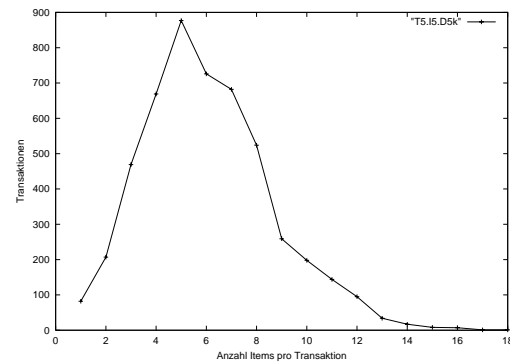


Abbildung 5: kleine Transaktionstabelle T5.I5.D5k

Eine andere Art von Transaktionstabellen enthält viel weniger Transaktionen. In Abbildung 4 und 5 werden zwei kleine Tabellen abgebildet, die jeweils die durchschnittliche Anzahl von Items pro Transaktion von 5 haben und einmal 10000 und einmal 5000 Transaktionen beinhalten. Diese kleinen Tabellen waren nötig, um Ergebnisse für den neuen Ansatz Quiver zu bekommen. Bei den großen Transaktionstabellen ist dieser Ansatz entweder abgebrochen oder hatte Laufzeiten von über sechzig Stunden. Daher wurden kleinere Tabellen erzeugt, um mit Quiver Messergebnisse erzielen zu können. Die Gründe für diese Abbrüche und langen Laufzeiten werden in Kapitel “Messungen und Schlussfolgerungen” genauer erläutert.

Eine Übersicht der Transaktionstabellen kann folgender Tabelle 1 entnommen werden.

Tabellenname	Zeilen	Transaktionen	durchschnittl. Transaktionsgröße	Anzahl der FI	Patternlänge	Anzahl der versch. Items	Art
T5.I5.D100k_mod	611652	100000	6.12	2000	5	1000	flach
T5.I5.D100k (s. Abbildung 3)	544590	100000	5.45	2000	5	1000	hoch
T10.I5.D100k_mod	1016906	100000	10.17	2000	5	1000	flach
T10.I5.D100k (s. Abbildung 2)	1002226	100000	10.02	2000	5	1000	hoch
T5.I5.D10k (s. Abbildung 4)	58964	10000	5.90	200	5	100	hoch
T5.I5.D5k (s. Abbildung 5)	30268	5000	6.05	200	5	100	hoch

Tabelle 1: Transaktionstabellen Informationen

Die Tabellennamen sind nach folgendem Muster aufgebaut:

Zuerst wird nach dem “T” die Anzahl der durchschnittlichen Items pro Transaktion angegeben, darauf die Patternlänge und anschließend die Anzahl der Transaktionen in tausend. Als Beispiel sei T10.I5.D100k die Transaktionstabelle, dann besitzt diese im Schnitt

10 Items pro Transaktion, eine Patternlänge von 5 und 100000 Transaktionen. Unter Patternlänge versteht man eine Menge, die eine bestimmte Anzahl von Frequent Itemsets enthält. Diese Itemsets beinhalten im Schnitt jeweils fünf Items. Aus dieser Menge werden dann Itemsets ausgewählt und zu der Transaktionstabelle hinzugefügt. Teilweise werden die ausgewählten Itemsets noch modifiziert, indem Items herausgelöscht werden, durch andere ersetzt werden, oder neue Items hinzugefügt werden. Dadurch wird gewährleistet, dass diese Tabellen auf jeden Fall Frequent Itemsetkombinationen enthalten.

Die Unterschiede - ein flacher oder ein hoher Verlauf - dieser oben genannten Kurven rühren von dem Algorithmus aus [1] her, der benutzt worden ist, um diese synthetischen Transaktionstabellen zu erstellen. Die steilen und hohen Kurven sind durch die Tabellen entstanden, die mit dem originalen Algorithmus erzeugt wurden. Dazu wurde folgender Algorithmus verwendet:

```
{
// even though a Poisson distribution with mean 10 can be well
// approximated by a normal distribution, you will use the standard
// algorithm for means up to 100

//if (mean < 0)
if (mean < 100)
{
// See Knuth, TAOCP, vol. 2, second print
// section 3.4.1, algorithm Q on page 117
// Q1. [Calculate exponential]
//double p = Math.exp(-(double)mean);
double p = Math.exp(-(double)mean);
long N = 0;
double q = 1.0;

while (true)
{
// Q2. [Get uniform variable]
double U = rand.nextDouble();
// Q3. [Multiply]
q = q * U;
// Q4. [Test]
if (q >= p)
N = N + 1;
else
return N;
}
}
// for larger mean values you approximate the Poisson distribution
// using a normal distribution
else
{
double z = rand.nextGaussian();

//long value = (long)(mean + 0.3 * z * Math.sqrt(mean) + 0.5);
long value = (long)(mean + z * Math.sqrt(mean) + 0.5);
if (value >= 0)
return value;
else return 0;
}
}
```

Die in dieser Implementierung fett abgedruckten Stellen sind relevant, um einen flacheren Verlauf der Kurven zu realisieren. Der so abgebildete Algorithmus erzielt die ori-

ginalen Transaktionstabellen. Werden an den fett gedruckten Stellen die jeweils zweite Zeile auskommentiert und die jeweils erste Zeile verwendet, so werden die Transaktionstabellen mit dem flacheren Verlauf erzeugt.

Durch diese Änderung des Algorithmus kann es nun zu größeren Abweichungen bei der Anzahl der durchschnittlichen Items pro Transaktion kommen. Dies kommt daher, dass der Algorithmus versucht, bestimmte Rahmenbedingungen einzuhalten.

## 4 Frequent Itemset Discovery mit SQL

Wie bereits erwähnt, teilt sich Frequent Itemset Discovery in zwei Phasen auf: die Kandidatengenerierung und die Erstellung der Frequent-Itemset-Tabellen  $F_k^h$  und  $F_k^v$ . Die Kandidatentabellen werden im Folgenden mit  $C_k^h$  und  $C_k^v$  bezeichnet. Dabei steht das  $v$  für einen vertikalen Tabellenaufbau, und das  $h$  für einen horizontalen. Das  $k$  gibt an, in welchem Durchlauf man sich befindet. Es bestimmt somit die Anzahl der Items pro ItemsetID.

Quiver besitzt einen ganz neuen Ansatz. Die bekannten Algorithmen hatten sowohl für die Kandidatentabellen, als auch für die Frequent-Itemset-Tabellen eine horizontale Anordnung. Diese Tabellen sind nach folgendem Muster aufgebaut:

$C_1^h$	item <sub>1</sub>			
$C_2^h$	item <sub>1</sub>	item <sub>2</sub>		
...				
$C_k^h$	item <sub>1</sub>	item <sub>2</sub>	...	item <sub>k</sub>
$F_1^h$	item <sub>1</sub>	count(*)		
$F_2^h$	item <sub>1</sub>	item <sub>2</sub>	count(*)	
...				
$F_k^h$	item <sub>1</sub>	...	item <sub>k</sub>	count(*)

Beispiel  $C_4^h$

item <sub>1</sub>	item <sub>2</sub>	item <sub>3</sub>	item <sub>4</sub>
Salz	Mehl	Zucker	Eier
Salz	Mehl	Eier	Hefe
...	...	...	...

Tabelle 2: horizontaler Tabellenaufbau

Bei Quiver will man nun erreichen, dass die Breite der Tabellen und somit die Anzahl der Attribute für alle Durchgänge  $k=1$  bis 10 gleich bleibt. Daher hat man einen vertikalen Aufbau der Tabellen gewählt:

$C_1^v$	itemsetID	pos	item
...			
$C_k^v$	itemsetID	support	
...			
$S_k^v$			
$F_1^v$	itemsetID	pos	item
...			
$F_k^v$			

Beispiel  $C_4^v$

itemsetID	pos	item
101	1	Salz
101	2	Mehl
101	3	Zucker
101	4	Eier
102	1	Salz
102	2	Mehl
102	3	Eier
103	4	Hefe
...	...	...

Tabelle 3: vertikaler Tabellenaufbau

Um bei beiden Ansätzen einen ähnlichen Aufbau der  $F_k^h$ - und  $F_k^v$ -Tabellen zu erreichen, hat man  $S_k^h$ -Tabellen eingefügt. Man modifiziert die ursprünglichen  $F_k^h$ -Tabellen der bekannten Algorithmen aus der Literatur und entfernt aus diesen Tabellen den Eintrag "count(\*)". Diese  $S_k^h$ -Tabellen werden vor den  $F_k^h$ -Tabellen angelegt und enthalten die berechneten ItemsetIDs und deren dazugehörigen Supportwert. Man erhält daraus die  $F_k^h$ -Tabellen, indem man nur diejenigen Kandidaten in  $C_k^h$  auswählt, deren Support-Wert über dem Schwellwert *minsup* liegen. Nun steht sowohl in den  $F_k^h$ - als auch in den  $F_k^v$ -Tabellen die Information der ItemsetIDs und die dazugehörigen Items.

Der Aufbau aller einzelnen Tabellen, sowohl für den vertikalen, als auch für den horizontalen Ansatz, wie sie bei den folgenden Implementierungen verwendet worden sind, ist folgenden Tabellen 4 und 5 zu entnehmen. Dabei sind zusätzlich die Tabellen  $Q_k^h$ , die für den folgenden Subquery Algorithmus notwendig sind, und die Tabellen  $T_k^h$ , die bei dem Algorithmus Set-Oriented Apriori (SOAP) eingesetzt werden, abgebildet.

$C_k^h$	ItemsetID	Item <sub>1</sub>	Item <sub>2</sub>	Item <sub>3</sub>	...	Item <sub>k</sub>
$Q_k^h$	TID	Item <sub>1</sub>	Item <sub>2</sub>	Item <sub>3</sub>	...	Item <sub>k</sub>
$T_k^h$	TID	Item <sub>1</sub>	Item <sub>2</sub>	Item <sub>3</sub>	...	Item <sub>k</sub>
$S_k^h$	ItemsetID	support				
$F_k^h$	ItemsetID	Item <sub>1</sub>	Item <sub>2</sub>	Item <sub>3</sub>	...	Item <sub>k</sub>

Tabelle 4: verwendete horizontale Tabellen

$P^v$	Itemset_new	Itemset_old1	Itemset_old2
$C_k^v$	ItemsetID	pos	item
$S_k^v$	ItemsetID	support	
$F_k^v$	ItemsetID	pos	item

Tabelle 5: verwendete vertikale Tabellen

Um einen Performancegewinn zu erreichen, wurden auf allen Tabellen Indizes angelegt. Die Tabelle 6 enthält einen Überblick dieser Indizes:

$C_k^h$ :	<b>itemset</b> item <sub>1</sub> item <sub>2</sub> ... item <sub>k</sub>	$P^v$ :	<b>itemset_new</b> itemset_old1 itemset_old2
$Q_k^h$ :	<b>tid, item<sub>1</sub>, item<sub>2</sub>, ..., item<sub>k</sub></b> item <sub>1</sub> item <sub>2</sub> ... item <sub>k</sub>	$C_k^v$ :	<b>itemset, item</b> item itemset pos
$T_k^h$ :	<b>tid, item<sub>1</sub>, item<sub>2</sub>, ..., item<sub>k</sub></b> item <sub>1</sub> item <sub>2</sub> ... item <sub>k</sub>	$S_k^v$ :	<b>itemset</b> support
$S_k^h$ :	<b>itemset</b> support itemset, support support, itemset	$F_k^v$ :	<b>itemset, item</b> item itemset pos
$F_k^h$ :	<b>itemset</b> item <sub>1</sub> item <sub>2</sub> ... item <sub>k</sub>	Transaktions- tabellen:	Item TID Item, TID TID, Item

Tabelle 6: Angelegte Indizes für jede Tabelle

Die hierbei fett abgedruckten Indizes, sind die Primärschlüssel. Bei SQL sind diese gleichzeitig *clustered* Indizes.

Im Folgenden werden die Algorithmen K-Way Join, Subquery und SOAP aus der Literatur genommen und für die Durchgänge 1 bis 10 implementiert. Ebenso wird der Algorithmus Quiver für dieselben Durchgänge realisiert. Zusätzlich wird noch der Algorithmus K-Way Join umgeschrieben, der dann den vertikalen Tabellenaufbau nutzen kann.

Folgende Tabelle 7 soll für jeden Algorithmus veranschaulichen, welche Tabellen der Reihe nach berechnet werden. Die noch nicht erwähnte Tabelle  $T_f^v$ , die für den Algorithmus Soap benötigt wird, ist die gleiche Tabelle wie die ursprüngliche Transaktionstabelle, nur dass in dieser Tabelle  $T_f^v$  lediglich die Items vorkommen, die in  $F_1^h$  - die zuvor durch den Algorithmus berechnet worden ist - auch vorkommen.

K-Way Join	$C_1^h, S_1^h, F_1^h$	$C_2^h, S_2^h, F_2^h$	$C_3^h, S_3^h, F_3^h$	...	$C_{10}^h, S_{10}^h, F_{10}^h$
Subquery	$C_1^h, Q_1^h, S_1^h, F_1^h$	$C_2^h, Q_2^h, S_2^h, F_2^h$	$C_3^h, Q_3^h, S_3^h, F_3^h$	...	$C_{10}^h, Q_{10}^h, S_{10}^h, F_{10}^h$
SOAP	$C_1^h, Q_1^h, S_1^h, F_1^h$	$T_f^v, C_2^h, S_2^h, F_2^h$	$C_3^h, T_3^h, S_3^h, F_3^h$	...	$C_{10}^h, T_{10}^h, S_{10}^h, F_{10}^h$
Quiver	$C_1^v, S_1^v, F_1^v$	$P^v, C_2^v, S_2^v, F_2^v$	$P^v, C_3^v, S_3^v, F_3^v$	...	$P^v, C_{10}^v, S_{10}^v, F_{10}^v$
K-Way vert.	$C_1^v, S_1^v, F_1^v$	$P^v, C_2^v, S_2^v, F_2^v$	$P^v, C_3^v, S_3^v, F_3^v$	...	$P^v, C_{10}^v, S_{10}^v, F_{10}^v$

Tabelle 7: Berechnungsreihenfolge der einzelnen Algorithmen

## 4.1 Horizontales Layout

### 4.1.1 Kandidatentabelle

In jedem Durchlauf muss zuerst eine Kandidatentabelle erstellt werden. Diese hat den Namen  $C_k^h$ , wobei das  $k$  für den Durchgang und gleichzeitig für die Anzahl der Items pro Itemset steht. In Durchgang 1 werden alle Begriffe, die in der Transaktionstabelle vorkommen, jeweils einmal in diese Tabelle eingefügt. In Durchgang  $k$  werden die  $k$  Items pro Itemset ausgewählt, deren Teilmengen  $(k-1)$  in  $F_{k-1}^h$  vorkommen. Diese Tabelle enthält die möglichen Begriffe, die als häufig in Betracht kommen können und im nächsten Schritt dann genauer untersucht werden. Zu Beginn sind alle Begriffe, die in der Transaktionstabelle vorkommen, auch in der Kandidatentabelle vertreten. Im Schritt 1 ist die Kandidatentabellen Erstellung daher trivial:

```
CREATE TABLE Ch1 (itemset INTEGER IDENTITY
                  CONSTRAINT pk_c1 PRIMARY KEY NOT NULL,
                  item1 INTEGER)

INSERT INTO Ch1 (item1)
SELECT DISTINCT item
FROM data-table
```

Für die nachfolgenden Kandidatentabellen Generierung [9] wird immer auf die zuletzt berechneten Frequent-Itemset-Tabelle  $F_{k-1}^h$  zurückgegriffen. Für den Fall  $k$  ergibt sich folgender SQL Ausdruck:

```
CREATE TABLE Chk (itemset INTEGER IDENTITY
                  CONSTRAINT pk_ck PRIMARY KEY NOT NULL,
                  item1 INTEGER,
                  item2 INTEGER,
                  ...,
                  itemk INTEGER)

INSERT INTO Chk (item1, item2, ..., itemk)
FROM Fhk-1 AS a, Fhk-1 AS b
WHERE a.item1 = b.item1 AND
      ...
      a.itemk-2 = b.itemk-2 AND
      a.itemk-1 < b.itemk-1
```

Der obige Algorithmus wird an folgendem aus [9] entnommenen Beispiel kurz erklärt.  $F_3^h$  habe die Menge  $\{\{1\ 2\ 3\}, \{1\ 2\ 4\}, \{1\ 3\ 4\}, \{1\ 3\ 5\}, \{2\ 3\ 4\}\}$ . Der Algorithmus liefert für  $C_4^h$  das Ergebnis  $\{\{1\ 2\ 3\ 4\}, \{1\ 3\ 4\ 5\}\}$ . Man erkennt aber, dass nicht alle Teilmengen von

{1 3 4 5} in  $F_3^h$  vorkommen. Es fehlt die Teilmenge {1 4 5}. Daher kann man den Algorithmus noch erweitern, indem man nach Teilmengen (k-1) in  $C_k^h$  sucht, die nicht in  $F_{k-1}^h$  vorkommen. Dadurch werden die Kandidatentabellen verkleinert. Diese Suche bringt bei der Erstellung der Tabellen kleine Performanceeinbußen mit sich, dafür wird die Zeit für die Zählphase dann verkürzt. Folgende Erweiterungen werden bei der Kandidatentabellen Erstellung hinzugefügt:

```

siehe oben
FROM      Fhk-1 AS a, Fhk-1 AS b
          Fhk-1 AS c, Fhk-1 AS d...
WHERE     a.item1 = b.item1 AND
          ...
          a.itemk-2 = b.itemk-2 AND
          a.itemk-1 < b.itemk-1
-----
-- ERWEITERUNG:
-- Skip Item1
c.item1 = a.item2 AND
c.item2 = a.item3 AND
...
c.itemk-2 = a.itemk-1 AND
c.itemk-1 = b.itemk-1
-- Skip Item2...
d.item1 = a.item1 AND
d.item2 = a.item3 AND
...
d.itemk-2 = a.itemk-1 AND
d.itemk-1 = b.itemk-1
...
-- Skip Itemk-2
x.item1 = a.item1 AND
x.item2 = a.item2 AND
...
x.itemk-3 = a.itemk-3 AND
x.itemk-2 = a.itemk-1 AND
x.itemk-1 = b.itemk-1

```

Für das Beispiel  $k=4$  ist die obige Implementierung im Anhang abgedruckt.

### 4.1.2 K-way Join

Bei K-way Join [9] wird in jedem Durchgang  $k$  ein Join der Kandidatentabelle  $C_k^h$  mit  $k$  Transaktionstabellen durchgeführt und anschließend nach  $item_1$  bis  $item_k$  gruppiert.

```

INSERT INTO Fhk
SELECT   item1, ...itemk, count(*)
FROM     Chk, data-table AS t1, ...data-table AS tk
WHERE    t1.item = Chk.item1 AND
          ...
          tk.item = Chk.itemk AND
          t1.tid = t2.tid AND
          ...
          tk-1.tid = tk.tid
GROUP BY item1, item2, ...itemk
HAVING   count(tid) >=:minsup

```

K-Way Join überprüft im Durchgang  $k$ , ob ein Itemset aus der Kandidatentabelle  $C_k^h$  auch in der Transaktionstabelle vorkommt. Es wird eine Transaktion in der Transaktionstabelle gesucht, die alle Items aus dem Itemset enthält. Dieser Vorgang wird für alle Itemsets aus der  $C_k^h$ -Tabelle wiederholt.

### 4.1.3 Subquery

Um die Zählphase zu verkürzen, wird bei dem Subquery Algorithmus [9] von gemeinsamen Präfixen der Itemsets in den Kandidatentabellen profitiert. In der  $k$ -ten Subquery stehen alle TIDs, die diejenigen Itemsets enthalten, die in den ersten  $k$  Spalten der Kandidatentabelle  $C_k^h$  vorkommen. Die Subquery Tabellen werden folgendermaßen erstellt:

```

INSERT INTO Qhi (for any i between 1 and k)
SELECT   item1, item2, ...itemi, tid
FROM     data-table AS t1,
         Qhi-1 AS ri-1,
         (SELECT DISTINCT item1...itemi from Chk) AS di
WHERE    ri-1.item1 = di.item1 AND
         ... AND
         ri-1.itemi-1 = di.itemi-1 AND
         ri-1.tid = t1.tid AND
         t1.item = di.itemi

```

Es müssen erst alle  $Q_i^h$ -Tabellen (für  $i=1$  bis  $k$ ) berechnet werden, um schließlich die  $Q_k^h$ -Tabelle zu erhalten. Auf diese wird dann beim Erstellen der  $F_k^h$ -Tabellen in folgender Weise zurückgegriffen:

```

INSERT INTO Fhk
SELECT   item1, ..., itemk, tid
FROM     Qhk
GROUP BY item1, item2, ..., itemk
HAVING  count(tid) >=:minsup

```

### 4.1.4 Subquery mit materialisierten Zwischenergebnissen

Folgende Überlegung stellt sich bei oben genanntem Subquery Algorithmus:

Weshalb werden in jedem Schritt  $k$  alle  $Q_i^h$  ( $i=1$  bis  $k$ ) Tabellen von neuem berechnet? Dies ist sinnvoll, wenn nur eine bestimmte  $F_k^h$ -Tabelle (zum Beispiel  $F_6^h$ ) berechnet werden soll. Hier werden aber immer die Ergebnisse "bottom up" von  $k=1$  bis 10 berechnet. Daher genügt es, die letzte  $Q_k^h=Q_k^h$ -Tabelle zu berechnen. Es muss auf die Tabelle  $Q_{i-1}^h$  zurückgegriffen werden. Diese ist die im Schritt ( $k-1$ ) berechnete  $Q_{k-1}^h$ -Tabelle. Analog wird dies bis zur  $Q_1^h$ -Tabelle fortgeführt. Eine  $Q_0^h$ -Tabelle existiert nicht.

Durch diese Modifikation wird auf bereits berechnete und nun auf Festplatte vorhandene Tabellen zurückgegriffen. Dies hat zur Folge, dass die CPU-Zeit für die Berechnung der Tabellen verkürzt wird. Natürlich wird die I/O Zeit etwas erhöht, sofern die benötigten Tabellen nicht mehr im Arbeitsspeicher vorliegen, denn diese Daten müssen wieder von der Festplatte geladen werden. Diese Tabellen  $Q_k^h$  sind mit denen des originalen Algorithmus Subquery identisch. Die Berechnung der  $F_k^h$ -Tabellen erfolgt analog zu oben genanntem Subquery Algorithmus und wird hier nicht noch einmal aufgeführt. Die Generierung der  $Q_k^h$ -Tabellen sieht dann folgendermaßen aus:

```

INSERT INTO Qhk
SELECT      item1, item2, ...itemk, tid
FROM        data-table AS tk,
            Qhk-1 AS rk-1,
            (SELECT DISTINCT item1...itemk from Chk) AS dk
WHERE       rk-1.item1 = dk.item1 AND
            ... AND
            rk-1.itemk-1 = dk.itemk-1 AND
            rk-1.tid = tk.tid AND
            tk.item = dk.itemk

```

#### 4.1.5 Set-oriented Apriori

Um nicht immer auf die ganze Transaktionstabelle zurückgreifen zu müssen, wird eine neue Tabelle  $T_f$  erstellt. Diese enthält nur noch die Frequent Items. Davor wird zuerst  $F_1^h$  mit dem Subquery Algorithmus berechnet. Nachdem  $F_1^h$  vorliegt, wird  $T_f$  folgendermaßen generiert. Es wird die Transaktionstabelle mit  $F_1^h$  gejoin:

```

INSERT INTO Tf
SELECT      a.tid, a.item
FROM        Data-table AS a,
            $F^h_1$ AS b
WHERE       a.item = b.item1

```

Nachdem der Set-oriented Apriori (SOAP) [10] Algorithmus erst für die Berechnung höherer ( $k \geq 3$ ) Frequent Itemsets effizient ist, wird  $F_2^h$  direkt berechnet:

```

INSERT INTO Fh2
SELECT      p.item, q.item, count(*)
FROM        Tf AS p,
            Tf AS q
WHERE       p.tid = q.tid AND
            p.item < q.item
GROUP BY   p.item, q.item
HAVING     count(*) >:minsup

```

Je nach Transaktionstabelle und *minsup* Wert kann für  $F_3^h$  noch der Subquery Algorithmus schneller sein. Bei den folgenden Tests wurde jedoch  $T_3^h$  durch einen Join von  $T_f$  und  $C_3^h$  direkt berechnet. Die Tabelle  $T_3^h$  ist mit der Tabelle  $Q_3^h$ , die mit dem Subquery Algorithmus entsteht, identisch.

```

INSERT INTO T3
SELECT      p.tid, p.item, q.item, r.item
FROM        Tf AS p,
            Tf AS q,
            Tf AS r,
            Ch3
WHERE       p.item = Ch3.item1 AND
            q.item = Ch3.item2 AND
            r.item = Ch3.item3 AND
            p.tid = q.tid AND
            q.tid = r.tid

```

Nachdem bei allen Durchgängen ähnliche Arbeit verrichtet wird, macht man sich dies bei SOAP zunutze. Die eigentliche Arbeit ist das Generieren von Item-Kombinationen.

Um nicht immer wieder die gleichen Kombinationen erstellen zu müssen, speichert man sich diese in den  $T_k^h$ -Tabellen ab. Speziell bei höheren Durchgängen ( $k \geq 3$ ) wird ein Performancegewinn bemerkbar sein, wenn der Algorithmus nicht immer sich wiederholende Arbeit leisten muss.

Da der SOAP Algorithmus, im Gegensatz zu den beiden bisher genannten Algorithmen, bei höheren Itemsets an Effizienz gewinnt, wird erst ab  $T_4^h$  der SOAP Ansatz verwirklicht und die  $T_k^h$ -Tabellen folgendermaßen erstellt:

```

INSERT INTO Tk
SELECT      p.tid, p.item1, ... p.itemk-1, q.item
FROM        Chk,
           Tk-1 AS p,
           T1 AS q
WHERE       p.item1 = Chk.item1 AND
           ... AND
           p.itemk-1 = Chk.itemk-1 AND
           q.item = Chk.itemk AND
           p.tid = q.tid

```

Die  $F_k^h$ -Tabellen werden folgendermaßen berechnet:

```

INSERT INTO Fhk
SELECT      Shk.itemset,
           Chk.item1,
           ...
           Chk.itemk
FROM        Chk, Shk
WHERE       Shk.itemset = Ck.itemset

```

## 4.2 Anpassung der Algorithmen

Wie im Anhang zu erkennen ist, wurde die Implementierung der drei Algorithmen K-Way Join, Subquery und Soap nicht direkt aus der Literatur übernommen. Es wurden Anpassungen gemacht, damit man die einzelnen Tabellen untereinander besser vergleichen kann. Die Idee der Algorithmen ist beibehalten worden, lediglich das Layout der einzelnen Tabellen wurde geändert. Bei den Algorithmen in der Literatur kommen keine  $S_k^h$ -Tabellen vor und deren  $F_k^h$ -Tabellen enthielten noch das Attribut `count(*)`. Hier wurde eine neue Tabelle  $S_k^h$  und  $S_k^v$  eingefügt. Diese werden immer vor den eigentlichen  $F_k^h$ - und  $F_k^v$ -Tabellen berechnet und enthalten alle Itemsets und deren Supportwert. In diesen Tabellen stehen jeweils die ItemsetIDs und danach der Supportwert, der angibt, wie oft dieses Itemset in der Transaktionstabelle vorkommt. Hier wird also die eigentliche Arbeit getan, die in der Literatur immer bei der  $F_k^h$  Berechnung erfolgt ist. Um jetzt die  $F_k^h$ - und  $F_k^v$ -Tabellen zu berechnen, muss man noch bei den  $S_k^h$ - und  $S_k^v$ -Tabellen suchen, welche Itemsets dem geforderten *minsup* genügen. Das heißt, dass die  $S_k^h$ -Tabellen genau so gebildet werden, wie bisher die  $F_k^h$ -Tabellen gebildet worden sind, nur dass die letzte Bedingung "*HAVING count(\*) >=:minsup*" wegfällt.

## 4.3 Vertikales Layout

### 4.3.1 Universal Quantification

Das Prinzip dieses Algorithmus, der *Quiver* genannt wird (von QUAntification und VERtical) ist die Division. Man dividiert die Transaktionstabelle durch das gerade gesuchte Itemset und erhält somit all die Transaktionen, in denen das Itemset vorkommt. Nun zählt man noch die Anzahl dieser Transaktionen. Somit erhält man als Ergebnis, wie oft dieses Itemset in der Transaktionstabelle vorkommt. Meist hat man jedoch nicht nur ein Itemset, sondern eine ganze Tabelle  $C_k^v$  mit vielen Itemsets und man will von jedem einzelnen berechnen, wie oft dieses in der Transaktionstabelle vorkommt. Dazu wurde ein Algorithmus *SetContainmentDivision* in der Studienarbeit [4] implementiert. Dieser liefert zu jedem Itemset einer Kandidatentabelle den Wert, wie oft dieses in der Transaktionstabelle vorkommt. Auf diese Weise erhält man die  $S_k^v$ -Tabelle, die die vorkommenden Itemsets und deren Anzahl in der Transaktionstabelle enthält.

Der Nachteil bei SQL ist, dass es keinen Divisionsoperator gibt. Daher muss man sich mit anderen Mitteln helfen. Dies erfolgt in [6] mit dem mathematischen Ausdruck  $\forall$  (für alle). Nachdem jedoch auch dieser Operator in SQL nicht zur Verfügung steht, muss man dies mit  $\nexists$  (nicht existiert) erledigen, das das gleiche Ergebnis - nach der entsprechenden Umwandlung der Formel - liefert. Man erkennt, dass man in SQL komplizierte Statements erhält, um diese Division zu erreichen. Dies resultiert in einem immensen Performanceverlust. Ein Überblick kann der Tabelle 8 entnommen werden.

**Kandidatentabelle:** In den  $C_k^v$ -Tabellen stehen die gleichen Informationen, wie in den  $C_k^h$ -Tabellen. Der Unterschied besteht darin, dass die Informationen erstens anders berechnet werden und zweitens in einem vertikalen Layout vorliegen. Im Folgenden wird die Generierung der  $C_k^h$ -Tabellen für Quiver geschildert. Zuvor werden jeweils Prefix-Tabellen erstellt. Diese Tabellen beinhalten die neue ItemsetID. Hier werden zu den zwei ItemsetID, aus denen das neue Itemset gebildet worden ist, die zugehörige neue ID zugeordnet:

```

INSERT INTO Pv
SELECT      itemset1, itemset2
FROM
  SELECT DISTINCT  a1.itemset AS itemset1,
                   a2.itemset AS itemset2
FROM           Fvk-1 AS a1,
              Fvk-1 AS a2
WHERE NOT EXISTS (
  SELECT      *
FROM         Fvk-1 AS b1,
              Fvk-1 AS b2
WHERE        ((b1.itemset = a1.itemset) AND
              (b2.itemset = a2.itemset) AND
              (b1.pos < @k-1) AND
              (b1.pos = b2.pos)) AND
              NOT (b2.item = b1.item)
) AND
EXISTS (
  SELECT      b1.item,

```

```

        b2.item
FROM    Fvk-1 AS b1,
        Fvk-1 AS b2
WHERE   ((b1.itemset = a1.itemset) AND
        (b2.itemset = a2.itemset) AND
        (b1.pos < @k-1) AND
        (b1.pos = b2.pos)) AND
        (b1.item < b2.item)
)

```

Nun kann der obige Algorithmus noch so erweitert werden, dass alle Items, die gar nicht als Frequent in Betracht kommen können, nicht in Betracht gezogen werden. Dazu werden folgende SQL Statements an obige Anweisung angehängt:

```

AND
-- Man entfernt das Item an Position p von Itemset a1

-- Entferne Item an Position p = 1
EXISTS (
  SELECT a3.itemset
  FROM   Fvk-1 AS a3
  WHERE NOT EXISTS (
    SELECT b1.item,
           b2.item
    FROM   Fvk-1 AS b1,
           Fvk-1 AS b2
    WHERE NOT (
      -- Bedingung 1: 1 <= i < p
      (
        NOT (
          (b1.itemset = a1.itemset) AND
          (b2.itemset = a3.itemset) AND
          (1 <= b2.pos) AND
          (b2.pos < 1) AND
          (b1.pos = b2.pos)
        ) OR
        (b1.item = b2.item)
      )
      AND
      -- Bedingung 2: p <= i < k-1
      (
        NOT (
          (b1.itemset = a1.itemset) AND
          (b2.itemset = a3.itemset) AND
          (1 <= b2.pos) AND
          (b2.pos < @k-1) AND
          (b1.pos = b2.pos + 1)
        ) OR
        (b1.item = b2.item)
      )
      AND
      -- Bedingung 3: i = k-1
      (
        NOT (
          (b1.itemset = a1.itemset) AND
          (b2.itemset = a3.itemset) AND
          (b2.pos = @k-1) AND
          (b1.pos = b2.pos)
        ) OR
        (b1.item = b2.item)
      )
    )
  )
)

```

```

)

-- Analog diese EXIST Anweisung bei p = 2..(k-1)
-- Entferne Item an Position p = 2
-- Entferne Item an Position p = 3
-- ...
-- Entferne Item an Position p = (k-1)

```

Mit Hilfe dieser erstellten Tabelle wird dann die eigentliche Kandidatentabelle  $C_k^v$  erstellt:

```

INSERT INTO Ckv
SELECT      p.itemset_new,
            f.pos,
            f.item
FROM        Fk-1v AS f,
            Pv AS p
WHERE       f.itemset = p.itemset_old1
UNION
SELECT      p.itemset_new,
            @k,
            f.item
FROM        Fk-1v AS f,
            Pv AS p
WHERE       f.itemset = p.itemset_old2 AND
            f.pos      = @k-1

```

**Die  $S_k^v$ - und  $F_k^v$ -Tabellen** Anschließend wird die  $S_k^v$ -Tabellen erstellt. Diese Tabelle beinhaltet alle  $k$ -Itemsets, die in der Transaktionstabelle mindestens einmal vorkommen, und den dazugehörigen *support* Wert. Dieser beinhaltet, wie oft das Itemset in der Transaktionstabelle vorkommt. Um diese Tabellen zu erstellen, wird auf die Division zurückgegriffen. Man teilt die Transaktionstabelle durch ein Itemset, das in der Kandidatentabelle steht. Dadurch erhält man die Anzahl, wie oft dieses Itemset in der Transaktionstabelle vorkommt. Dies wird für alle Itemsets aus der Kandidatentabelle wiederholt und man erhält somit die  $S_k^v$  Tabelle. Leider gibt es für SQL keinen Divisionsoperanden und man behilft sich (wie oben bereits erklärt) mit dem SQL-Statement *NOT EXISTS*. Folgende Tabelle 8 veranschaulicht dies:

Division	$\frac{T}{c_1 \in C}$
Relational (aus [6])	<p><b>Query:</b>  <math>\{(c_1 \in C, t_1 \in T) \mid \text{contains}\}</math></p> <p><b>Expression:</b>  contains =  <math>\forall c_2 \in C \exists t_2 \in T(</math>  <math>(c_1.\text{itemset} = c_2.\text{itemset}) \rightarrow (t_1.\text{tid} = t_2.\text{tid} \wedge t_2.\text{item} = c_2.\text{item}))</math></p>
Umformung in Statements, die in SQL vorhanden sind	<p><b>Query:</b>  <math>\{(c_1 \in C, t_1 \in T) \mid \text{contains}\}</math></p> <p><b>Expression:</b>  contains =  <math>\nexists c_2 \in C \nexists t_2 \in T(</math>  <math>\neg(c_1.\text{itemset} = c_2.\text{itemset}) \vee (t_1.\text{tid} = t_2.\text{tid} \wedge t_2.\text{item} = c_2.\text{item}))</math></p>

Tabelle 8: Division durch SQL Statements ersetzt

In SQL sieht dann die Generierung der  $S_k^v$ -Tabellen folgendermaßen aus:

```

INSERT
INTO   Svk
SELECT itemset,
       COUNT (DISTINCT Tid)
FROM   (
  SELECT c1.itemset,
         t1.tid
  FROM   Cvk AS c1,
         data-table AS t1
  WHERE  NOT EXISTS (
    SELECT *
    FROM   Cvk AS c2
    WHERE  NOT EXISTS (
      SELECT *
      FROM   data-table AS t2
      WHERE  NOT (c1.itemset = c2.itemset) OR
              (t2.tid = t1.tid AND
               t2.item = c2.item))
    ) AS relevant
)
GROUP BY itemset

```

### 4.3.2 K-Way Join vertikal

Die  $S_k^v$ -Tabellen sind nach dem neuen Ansatz mit Quantifizierung erstellt worden. Um diesen Schritt besser mit den bisherigen Methoden vergleichen zu können, hat man diese

$S_k^v$ -Tabellen noch mit K-Way Join Ansatz berechnet. Dazu wurde der K-Way Join Algorithmus umgeschrieben, so dass dieser die Kandidatentabelle  $C_k^v$  benutzen kann. Der originale Algorithmus basiert auf einem horizontalen Tabellenlayout. Die Modifikationen waren nötig, um die Kandidatentabellen des vertikalen Tabellenlayouts verwenden zu können und um die berechneten Informationen in die Tabellen  $S_k^v$  schreiben zu können. Diese Änderungen führen letztendlich zu demselben Ergebnis, da in den  $C_k^h$ - und  $C_k^v$ -Tabellen die gleichen Informationen vorliegen, nur dass die Tabellen einen unterschiedlichen Tabellenaufbau besitzen. Um die  $S_k^v$ -Tabelle mit dem K-Way Join Algorithmus zu berechnen, werden folgende SQL-Statements implementiert:

```

INSERT INTO  Svk
SELECT      a1.itemset,
           count(tid)
FROM        Cvk AS a1,
           Cvk AS a2,
           . . . ,
           Cvk AS a k,
           data-table AS t1,
           data-table AS t2,
           . . . ,
           data-table AS tk
WHERE       a1.itemset = a2.itemset AND
           a1.itemset = a3.itemset AND
           . . .
           a1.itemset = ak.itemset AND

           t1.tid      = t2.tid      AND
           t1.tid      = t3.tid      AND
           . . .
           t1.tid      = tk.tid      AND

           a1.item      = t1.item      AND
           a2.item      = t2.item      AND
           . . .
           ak.item     = tk.item      AND

           a1.pos       = 1             AND
           a2.pos       = 2             AND
           . . .
           ak.pos      = k

GROUP BY    a1.itemset

```

Um die  $F_k^v$ -Tabellen zu erstellen, müssen aus den  $S_k^v$ -Tabellen lediglich diejenigen ItemsetIDs ausgewählt werden, deren *support* Wert über *minsup* liegt:

```

INSERT INTO  Fvk (itemset, pos, item)
SELECT      c.itemset, c.pos, c.item
FROM        Cvk AS c,
           Svk AS s
WHERE       c.itemset = s.itemset AND
           s.support >=@minsup

```



## 5 Frequent Itemset Discovery mit XXL

In diesem Kapitel wird beschrieben, wie die oben genannten Algorithmen unter Java implementiert wurden. Dabei wird auf die Bibliothek XXL [2] zurückgegriffen, die schon einige Datenbankfunktionalitäten (wie zum Beispiel Gruppierung oder Nested-Loop Join) für Java bereitstellt. Dadurch sollen die Algorithmen aus der Literatur dem neuen Ansatz Quiver gegenübergestellt werden. Hierzu werden die Ausführungspläne, die der SQL Server für die Algorithmen erstellt hat, untersucht und mit den Methoden von XXL und zusätzlichen eigenen notwendigen Methoden in Java realisiert. Somit wird versucht, die internen Abläufe des SQL Servers zu simulieren. Und schließlich kann dann der eigentliche Ansatz von Quiver - die Division - mit Java realisiert werden. Bisher wurde in dem SQL-Server auf die bereits im vorigen Kapitel erwähnte Quantifizierung zurückgegriffen, da ein Divisionsoperator in SQL nicht vorhanden ist. Am Beispiel des FID soll nun die Nützlichkeit eines solchen Operators untersucht werden. Nachdem es allerdings viel zu komplex geworden wäre, alle oben genannten Algorithmen in Java zu realisieren, hat man sich auf einen kleinen Ausschnitt konzentriert: Hierzu wurde der Algorithmus K-Way Join und Quiver für den Schritt  $k=4$  implementiert. Es wurde nur der eigentliche Algorithmus in Java implementiert, also die Generierung der  $S_4$  und  $F_4$  Tabellen. Die Kandidatentabelle  $C_4$  wurden nicht mit Java generiert, sondern ist direkt aus dem SQL Server kopiert oder manuell erstellt worden. Allerdings wurde auf die Verwendung von Indizes nicht verzichtet. Anhand des Ausführungsplans konnte abgelesen werden, welche Indizes von SQL verwendet worden sind. Diese wurden dann auch in Java implementiert und verwendet. Somit hat man einen direkten Vergleich dieser Algorithmen unter den verschiedenen Umgebungen "SQL" und "Java".

Nun kann auch die eigentliche Idee der Division, die Quiver zu Grunde liegt, direkt implementiert werden. Unter Java hat man die Möglichkeit, solch einen Divisionsoperator zu realisieren. Man muss daher nicht mehr den Umweg über die Quantifizierung gehen. Es wurde eine Divisionsklasse implementiert, um das FID Problem für den Fall  $k=4$  zu realisieren. Somit können durch Gegenüberstellung der Division und Quiver in Java Rückschlüsse für SQL gewonnen werden, inwieweit sich ein solcher Divisionsoperator für SQL eignet.

### 5.1 XXL

XXL ist, wie oben bereits erwähnt, eine Klassenbibliothek für Java, entwickelt von der Universität Marburg, Institut für Mathematik und Informatik. Durch diese Bibliothek werden Indexstrukturen und Anfrageoperatoren zur Verfügung gestellt, mit deren Hilfe sich leicht Anfragen realisieren lassen. XXL stellt also schon einige nützliche Werkzeuge bereit, um bestimmte Operationen durchzuführen. Als Beispiel seien hier "NestedLoopsJoin" und "MergeSorter" erwähnt. Ebenso können bestehende Textdateien in einen Cursor geladen werden. Ein Cursor stellt einen Zeiger dar, der auf einer Tabelle positioniert wird und Zeile für Zeile weiterbewegt werden kann und schließlich Operationen auf der gerade positionierten Zeile ausführen kann. Ein einfaches Beispiel ist die Ausgabe einer

Transaktionstabelle, die in XXL als Cursor vorliegt: Solange weitere Tupel vorhanden sind, wird das Tupel, auf dem momentan der Cursor steht, ausgegeben und zum nächsten Tupel weiter gesprungen. Bei allen folgenden Implementierungen wurden immer *HashCodeArrayTuple* verwendet. Diese Tupel werden intern durch ein Array realisiert und besitzen einen Hash-Wert, der anhand der Informationen, die in dem Array gespeichert sind, berechnet wird. Analog wird zum Beispiel auch die Methode “NestedLoopsJoin” implementiert. Vereinfacht ausgedrückt, werden dieser Methode zwei Cursor übergeben und diese berechnet einen neuen Cursor, der zurückgeliefert wird. Dabei wird jeder der zwei Cursor Schritt für Schritt durchgegangen, und die jeweiligen mathematischen Operationen auf die positionierten Tupel angewendet und das Ergebnis in den dritten Ergebniscursor geschrieben. Folgende Abbildung 6 soll dies veranschaulichen:

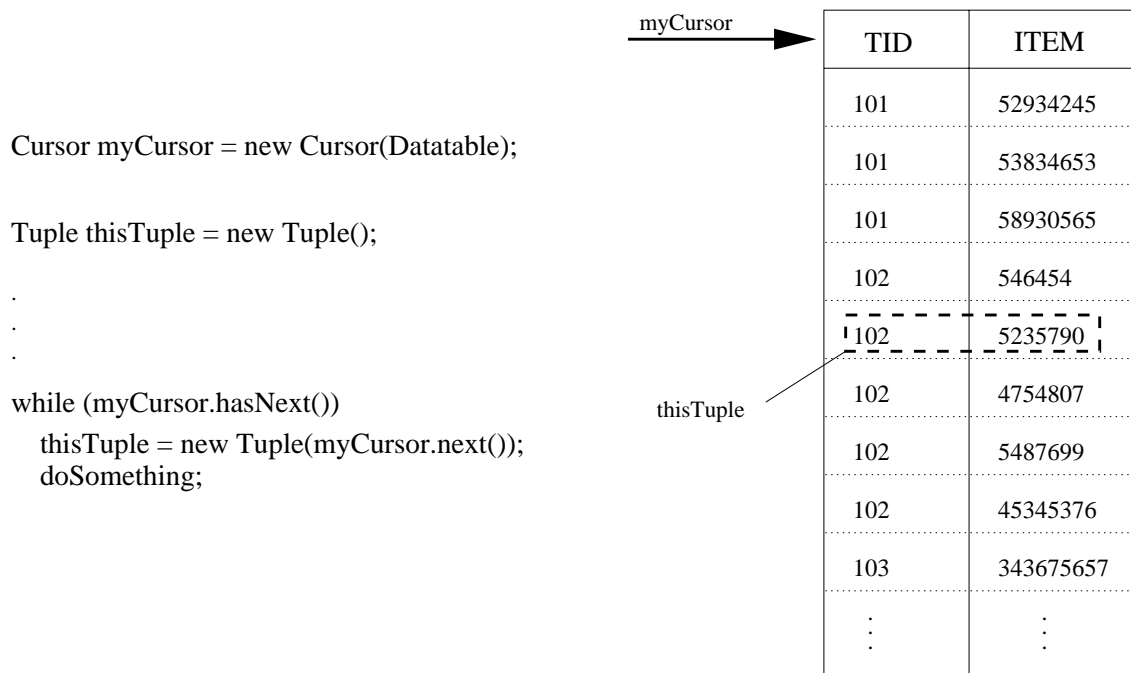


Abbildung 6: Cursor in XXL

Solch ein Cursor ist Hauptbestandteil der folgenden Implementierungen von K-Way Join, Divison und Quiver. Diese Cursor enthalten die Daten der Transaktions- und Kandidatentabellen und werden für weitere Berechnungen herangezogen. Bei den folgenden Algorithmen wird auch jeweils auf deren Implementierung eingegangen. Alle Klassen die dabei verwendet werden, sind im Folgenden kursiv dargestellt und werden in dem Kapitel 5.6 genauer erläutert.

## 5.2 Übersicht der Operatoren

In den Ausführungsplänen, die durch den SQL-Server berechnet worden sind, werden einige Operatoren verwendet. In diesem Kapitel wird eine Übersicht dieser Operatoren

in Tabelle 9 abgebildet. Bei den folgenden Algorithmen K-Way Join und Quiver werden diese Operatoren in Java realisiert.

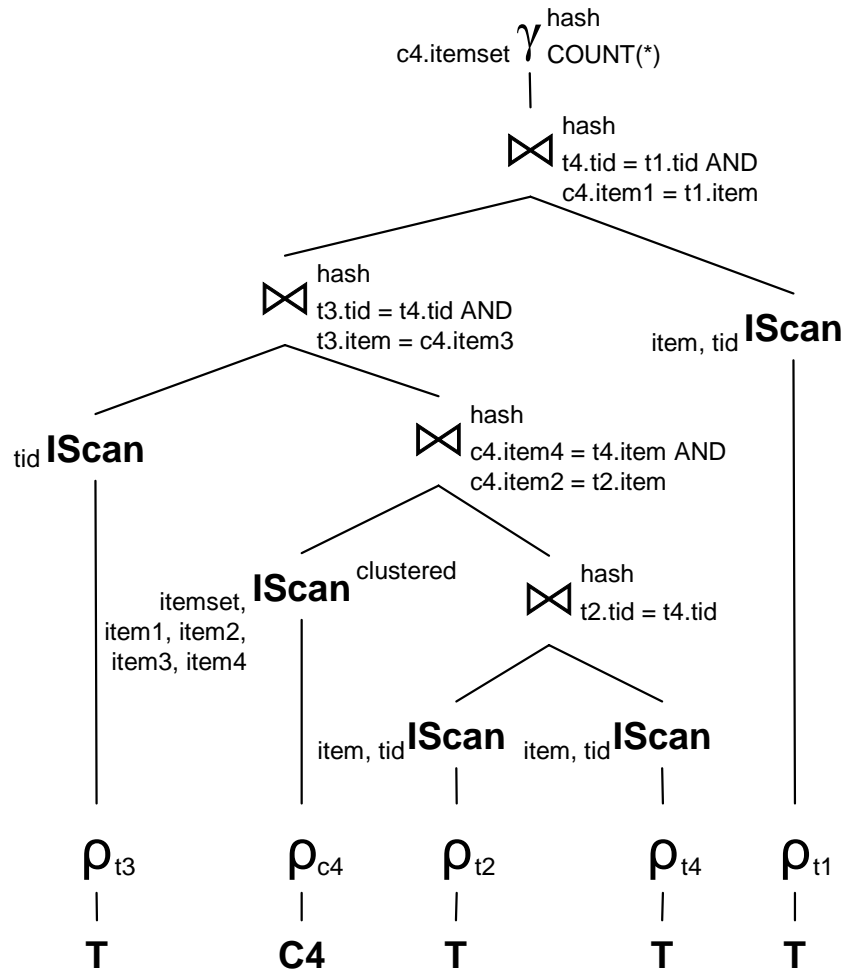
Operator	Beschreibung
$indexIScan$	Indexscan: Es wird entlang des angegebenen Index die zugrundeliegende Tabelle gelesen. Dabei werden alle Zeilen der Tabelle gelesen.
$indexISseek_{Bedingung}$	Es wird anhand des angegebenen Index die zugrundeliegende Tabelle gelesen. Dabei werden nur die Zeilen gelesen, bei denen die Bedingung zutrifft.
$RowCount$	Durch diesen Operator wird die Anzahl der Zeilen der Tabelle berechnet
$Top_1$	Es wird die erste Zeile der Tabelle ausgegeben.
$\gamma_{Attribut}$	Es wird nach dem Attribut gruppiert.
$\bowtie^{hash}$	Hash-Join
$\overline{\bowtie}^{NL}$	Nested-Loop-Anti-Semi-Join

Tabelle 9: Übersicht der Operatoren der Ausführungspläne

### 5.3 K-Way Join in Java

Um K-Way Join mit Java realisieren zu können, wurde die Vorgehensweise des SQL Servers anhand des Ausführungsplans untersucht. Dazu wurde für das SQL-Statement des K-Way Join Algorithmus dieser Plan mit der Transaktionstabelle T5.I5.D5k und einem Supportwert von 100 für den Durchgang  $k=4$  erzeugt. Dieses SQL-Statement ist im Anhang aufgeführt. Dies geschah mit dem SQL-Statement *SET SHOWPLAN\_ALL ON* unter dem SQL-Server. Danach folgten die eigentlichen Statements für K-Way Join und schließlich das Statement *SET SHOWPLAN\_ALL OFF*. Man erhält somit eine Textdatei, die im Anhang abgedruckt ist. In [7] wurde diese Datei in einen grafischen Ausführungsplan umgewandelt und ist in Abbildung 7 dargestellt. Durch die grafische Abbildung wird die Vorgehensweise des SQL-Servers verdeutlicht. Diese Vorgehensweise wird in Java implementiert. Da die komplette Implementierung doch sehr umfangreich ist, wird im Folgenden nur auf die wesentlichen Bestandteile eingegangen.

Zu Beginn werden die Transaktionstabellen  $t1-t4$  und die Kandidatentabellen  $c4$  (siehe Abbildung 7) eingelesen. Dies geschieht jeweils durch die Erstellung eines *BufferedFile-MetaDataCursor*. Alle notwendigen Tabellen sind nun als Cursor vorhanden. Der K-Way Join Algorithmus benutzt aber immer einen IndexScan, um auf die Daten zuzugreifen. Daher werden auf diesen erstellten Cursor Indizes angelegt. Dies wird unter XXL mit der Klasse *MyBTree* implementiert. Man erstellt einen B-Baum, mit dessen Hilfe schnell auf bestimmte Tupel zugegriffen werden kann. In diesem Fall werden aber immer alle Tupel aus einem Index benötigt, da es sich um einen IndexScan und nicht um einen IndexSeek mit dazugehöriger Bedingung handelt. Daher wird mit der Klasse *MyBTreeQueryCursor* mit Hilfe des Index ein neuer Cursor erstellt, der alle Tupel - die Blätter des Index - enthält.

Abbildung 7: K-Way Join - Ausführungsplan für Berechnung von  $S_4^h$ 

Wird nun später der Cursor durchlaufen, so werden die Tupel anhand des erstellten Index ausgegeben. Nachdem dies für alle Tabellen t1-t4 und c4 realisiert worden ist, werden die Klassen *HashJoin* mit den jeweiligen zwei Cursor als Eingabe aufgerufen, und ein dritter Cursor wird als Ergebnis ausgegeben. Dies geschieht für alle vier im Ausführungsplan abgebildeten Hash-Joins. Schließlich wird der Cursor, der als Ergebnis berechnet worden ist, mit der Klasse *HashGrouper* gruppiert und das Ergebnis angezeigt. Bei dieser Implementierung wird ebenso die Zeit gemessen, um spätere Vergleiche zu ermöglichen. Dabei wird die Zeit erst gemessen, sobald alle Indizes erstellt worden sind. Das heißt, dass die Zeit für die Erstellung der Indizes nicht berücksichtigt wird, da man nur die eigentlichen Algorithmen miteinander vergleichen will.

## 5.4 Quiver in Java

Analog zu der Vorgehensweise bei K-Way Join, wurde auch bei dem Algorithmus Quiver in [7] der in Abbildung 8 gezeigte Ausführungsplan erstellt.

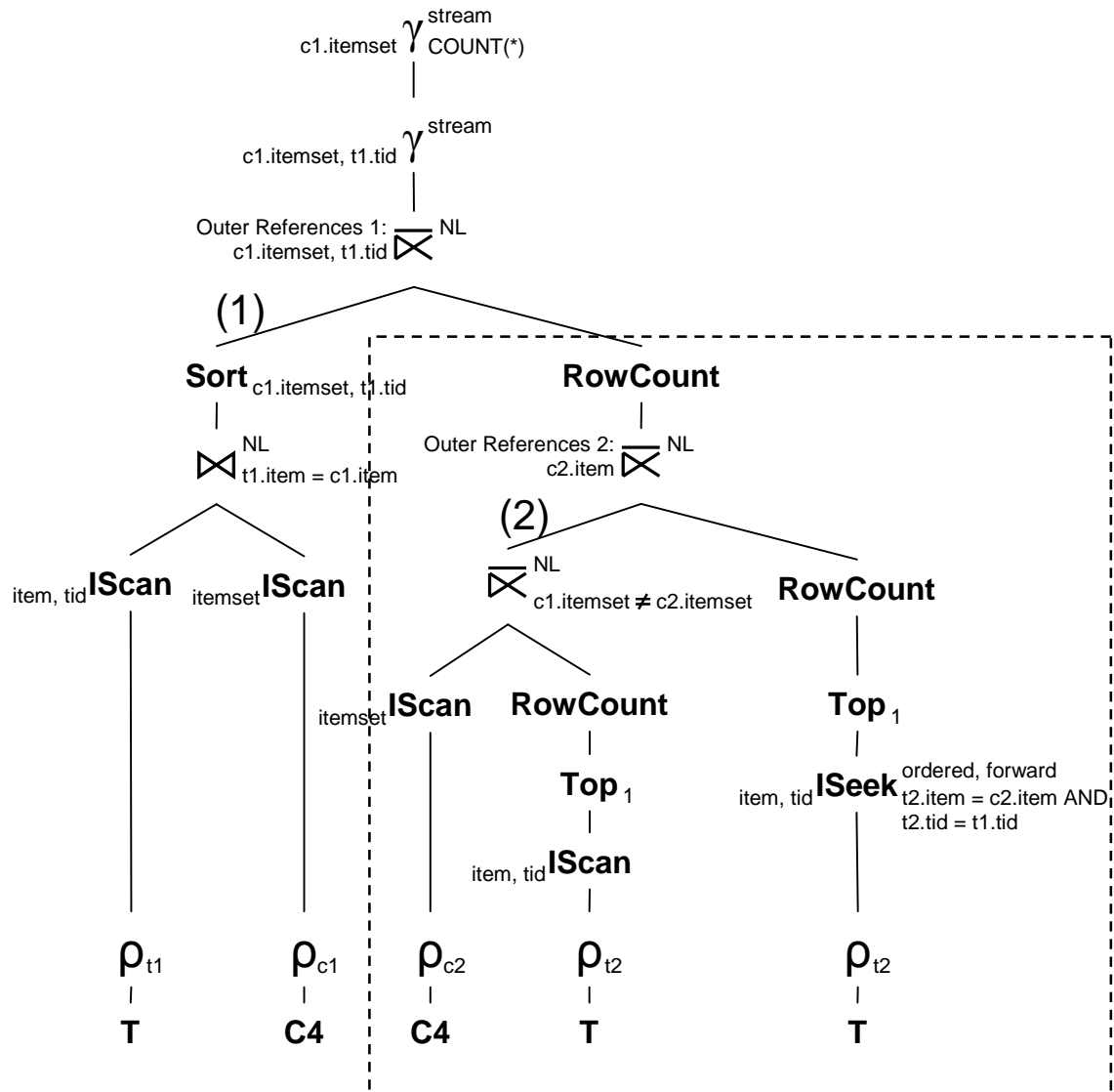


Abbildung 8: Quiver - Ausführungsplan für Berechnung von  $S_4^h$

Leider erfolgt die Vorgehensweise hier nicht bottom-up wie bei K-Way Join, da vom SQL-Server *Outer References* benutzt werden. Diese kommen vor, wenn eine Query in der FROM-Klausel eine Unteranfrage enthält. In dieser Unteranfrage wird dann auf Attribute einer Zeile der äußeren Anfrage zurückgegriffen. Der Wert der Spalte einer Outer Reference kommt von der momentan betrachteten Zeile der Hauptquery. Die Implementierung gestaltet sich somit etwas komplizierter. Begonnen wird aber - wie bereits bei

K-Way Join - mit dem Einlesen der Transaktionen und dem Erstellen der Indizes. Dazu werden analog die Daten  $t1$ ,  $t2$ ,  $c1$  und  $c2$  (siehe Abbildung 8) mit einem Objekt der Klasse *BufferedFileMetaDataCursor* eingelesen. Für alle vier entstandenen Cursor werden mit *MyBTree* Indizes erstellt. Bei  $t1$ ,  $c1$  und  $c2$  wird mit *MyBTreeQueryCursor* und den jeweiligen Indizes ein neuer Cursor implementiert. Für  $t2$  ist solch ein Cursor nicht notwendig, da ein Index Seek angewendet werden muss. Zwar kommt in dem Ausführungsplan auch ein Index Scan von  $t2$  vor, dieser muss aber nicht implementiert werden. Denn nach diesem Scan wird ein Top1 Operator und darauf ein RowCount angewendet. In diesem Ergebnis steht also entweder eine 1, sofern die Transaktionstabelle  $t2$  mindestens ein Tupel enthält, oder eine 0, wenn  $t2$  eine leere Tabelle ist. Da in den Transaktionstabellen üblicherweise immer Einträge stehen und auch das DBVS für diese Berechnung fast keine Zeit veranschlagt, wird immer diese (vom DBVS berechnete) Konstante 1 verwendet.

Nachdem diese Vorarbeiten geleistet worden sind, beginnt der eigentliche Algorithmus. Bisher wurden Daten eingelesen und Indizes angelegt. Es wird nun zuerst der Nested-Loop-Join von  $t1$  und  $c1$  mit der Bedingung  $t1.item=c1.item$  ausgeführt. Dies kann unter XXL mit der Klasse *NestedLoopsJoin* realisiert werden. Dieses Ergebnis wird mit *MergeSorter* sortiert. An diesem Punkt, in der Abbildung 8 mit (1) gekennzeichnet, erfolgt der erste Nested-Loop-Anti-Semi-Join mit Outer References (in der Abbildung mit "Outer References 1" dargestellt). Danach wird aus der Ergebnisliste des Sortieralgorithmus immer das nächste Tupel einzeln betrachtet. Implementiert wird dies mit einer While-Schleife, in der alle Tupel nacheinander ausgewählt werden. Mit diesem ausgewählten Tupel wird nun der in der Abbildung 8 gestrichelt gekennzeichnete Teilbaum berechnet.

Dazu wird das ausgewählte Tupel in einen Cursor geschrieben. Dies geschieht mit *MySingleObjectCursor*. Wie der Name schon sagt, kann der Cursor nur ein einziges Tupel enthalten. Daraufhin wird ein Nested-Loop-Anti-Semi-Join von  $c2$  und diesem Single-Object-Cursor mit der Klasse *NestedLoopsAntiSemiJoin* ausgeführt und auf die Bedingung  $c1.itemset \neq c2.itemset$  überprüft, wobei  $c1$  das gerade ausgewählte Tupel ist. Aus diesem Ergebnis nun wird wiederum ein Nested-Loop-Anti-Semi-Join mit Outer References (in der Abbildung mit Outer References 2 dargestellt) angewendet. Das heißt, dass wiederum jedes Tupel aus dem letzten Ergebnis in einen SingleObjectCursor geschrieben wird und ein Nested-Loop-Anti-Semi-Join mit einem IndexSeek von  $t2$  ausgeführt wird. Hierbei kann allerdings nicht auf die Klasse *NestedLoopsAntiSemiJoin* zurückgegriffen werden, da keine Bedingung mit dem Join verknüpft ist. Es wird daher mit einer If-Bedingung realisiert und mit dem Single-Object-Cursor der Index-Seek auf  $t2$  angewendet. Dabei gilt die Bedingung  $t2.item=c2.item$  und  $t2.tid=t1.tid$ , wobei  $t2$  der Index,  $c2.item$  der jetzige Single-Object-Cursor aus dem letzten Ergebnis (2) (Outer References 2) und  $t1.tid$  der Single-Object-Cursor aus dem Ergebnis (1) (Outer References 1) ist. Wird nun in dem Index mit dieser Bedingung ein Tupel für nur ein Single-Object-Cursor aus (2) gefunden, so wird das momentan ausgewählte Tupel aus (1) nicht zum Ergebnis hinzugefügt. Wird dagegen kein Tupel gefunden, wird es zum Ergebnis hinzugefügt. Dies spiegelt den Nested-Loop-Anti-Semi-Join Operator des SQL Server wider.

## 5.5 Division in Java

Die eigentliche Grundidee des Quiver Algorithmus ist die Division. Dieser Operator ist in SQL aber nicht implementiert, weshalb eine Umformung zur Quantifizierung stattgefunden hat - siehe Tabelle 8. In Java kann man natürlich nun solch einen Divisionsoperator für zwei Cursor implementieren. Dieser Implementierung werden zwei Cursor übergeben und man erhält direkt die ItemsetIDs und deren Supportwert. Diese Klasse heißt *SetContainmentDivision* und ist in einer Studienarbeit [4] erarbeitet worden. Daher musste die vorhandene Klasse nur noch nach folgender Abbildung 9 aus [7] angewendet werden.

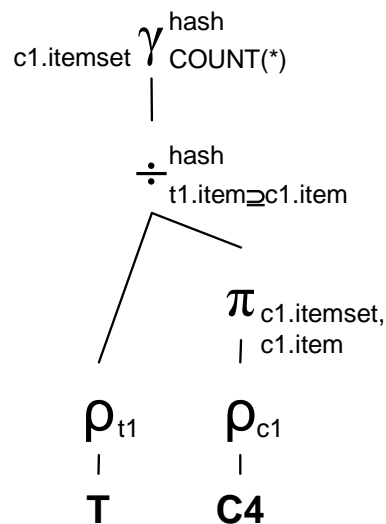


Abbildung 9: Division - graf. Ausführungsplan

## 5.6 Wichtige Klassen

Um die oben genannten Algorithmen zu implementieren, wurde auf einige Klassen von XXL zurückgegriffen, teilweise wurden auch eigene implementiert. Im Folgenden werden die benutzten XXL Klassen kurz beschrieben. In dem JavaDoc von XXL [2] sind diese ausführlich erläutert. Die neu implementierten Klassen werden etwas ausführlicher erklärt. Es wird jeweils zuerst der Konstruktor der Klasse abgebildet. Sollte die Klasse mehrere Konstruktoren besitzen, so wird der für die Implementierung verwendete Konstruktor abgebildet. Anschließend folgt die Erläuterung.

### 5.6.1 BufferedFileMetaDataCursor

<i>BufferedFileMetaDataCursor</i>	(MetaDataCursor	MetaDataCursor
	java.lang.String	bufferFileName
	Buffer	buffer
	int	blockSize)

Wie bereits erwähnt, wird bei der Implementierung der Algorithmen immer auf einen Cursor zurückgegriffen, der die Daten der Tabellen enthält. Um diese Informationen aus einer Textdatei in einen Cursor zu laden, wird die Klasse *BufferedFileMetaDataCursor* benutzt. Zuerst wird mit der Klasse *MyFileMetaDataCursor* die Textdatei in einen Cursor geladen. Dabei wird der Name der Datei, das Trennzeichen, das Kommentarzeichen und die Art der Tupel (HashCodeArrayTuple) übergeben. Danach wird dann die Klasse *BufferedFileMetaDataCursor* verwendet. Dadurch wird zu diesem Cursor noch ein Puffer angelegt, um einen ständigen Plattenzugriff auf das Textfile zu vermeiden. Beide Klassen wurden von Ralf Rantza, Institut für Parallele und Verteilte Systeme, Universität Stuttgart implementiert.

### 5.6.2 MyBTree

<i>MyBTree</i>	(java.lang.String	treeName
	int	blockSize
	Buffer	buffer
	int	objectSize
	int[]	keyColumns
	int	borderSize
	java.sql.ResultSetMetaData	tableMetaData
	java.sql.ResultSetMetaData	indexMetaData)

Diese Klasse, implementiert von Ralf Rantza, Institut für Parallele und Verteilte Systeme, Universität Stuttgart, dient dem einfacheren Programmieren. Die eigentliche Klasse BTree, die in XXL vorhanden ist, bietet zahlreiche und komplexe Möglichkeiten, um Indexstrukturen anzulegen. Daher wurde die Klasse *MyBTree* implementiert, um das Arbeiten zu erleichtern. Durch diese Klasse werden Methoden zur Verfügung gestellt, die ein einfaches Anlegen der Indizes, sowie das Öffnen, Schließen und Lesen der B-Bäume ermöglichen.

### 5.6.3 MyBTreeQueryCursor

<i>MyBTreeQueryCursor</i>	(BTree	btree
	ResultSetMetaData	tableMetaData)

Dieser Klasse wird ein BTree übergeben und man erhält einen MetaDataCursor mit allen Tupel, die in dem Index enthalten sind. Wird der Cursor später Schritt für Schritt durchlaufen, so werden die Tupel jeweils anhand des hinterlegten Index ausgewählt.

#### 5.6.4 NestedLoopsAntiSemiJoin

```
NestedLoopsAntiSemiJoin (MetaDataCursor leftTable
                          MetaDataCursor rightTable
                          int[] columnleft
                          int[] columnright
                          boolean equal)
```

Ein Algorithmus für den Operator Nested-Loop-Anti-Semi-Join war noch nicht vorhanden und musste realisiert werden. Hierzu werden der Klasse zwei *Cursor*, zwei *Integer*, die die jeweiligen Spalten angeben, und ein *Boolean* übergeben. Somit kann noch ausgewählt werden, ob auf Gleichheit oder Ungleichheit der jeweiligen Spalten überprüft werden soll. Die Implementierung erfolgt in diesem Fall recht einfach. Es werden zwei verschachtelte *while* Schleifen implementiert, so dass jedes Tupel der ersten Tabelle mit jedem Tupel der zweiten Tabelle verglichen werden kann. Diese Vorgehensweise nennt man "Nested-Loop". Sollten dann die jeweiligen Tupel in den jeweiligen Spalten gleich sein, so wird das Tupel des ersten *Cursor* NICHT zu dem Ergebnis hinzugefügt. Nachdem im Bedarfsfall nur die Tupel der ersten *Cursor* zu dem Ergebnis hinzugefügt werden, handelt es sich um einen "Semi-Join". Es werden aber bei Gleichheit diese Tupel des ersten *Cursor* nicht hinzugefügt, daher wird es "Anti-Semi-Join" genannt. Im Ganzen hat man somit einen Nested-Loop-Anti-Semi-Join implementiert.

#### 5.6.5 HashJoin

```
HashJoin (MetaDataCursor hashTable
           MetaDataCursor probeTable
           int[] columnhash
           int[] columnprobe)
```

Nachdem der SQL-Server intern einige Hash-Joins zur Berechnung der Frequent Itemsets bei dem Algorithmus K-Way Join benutzt, XXL aber diesen Algorithmus nicht bereitstellt, musste dieser implementiert werden. Als Parameter werden zwei *Cursor* erwartet, ebenso die jeweiligen Spalten, deren Gleichheit für den Join überprüft werden soll.

Im Folgenden wird auf die Vorgehensweise bei der Implementierung eingegangen. Es wird der Ablauf bei der Berechnung eines Hash-Joins geschildert, nicht jedoch die Feinheiten der Java-Implementierung. Hierzu ist der Quellcode im Anhang abgedruckt.

Der Klasse werden zwei *Cursor*, sowie die Spalten als Parameter mitgeliefert. Bei den zwei Tabellen nennt man die erste "hashTable" - das ist die Tabelle, auf der das *Hashing* angewendet wird - und dann folgt die "probeTable" - mit dieser erfolgt dann das *Probing*. Zunächst werden die neuen Metadaten berechnet. Metadaten sind zusätzliche Informationen neben den eigentlichen Daten, die eine Tabelle enthält. Zum Beispiel die Attributnamen der Spalten. Hierzu werden die Metadaten aus der *hashTable* und *probeTable* zusammengefügt. Bei einem Hash-Join entsteht ein neuer *Cursor*, der durch den Join der zwei *Cursor* *hashTable* und *probeTable* hervorgeht, mit der Bedingung, dass die jeweiligen Spalten, die als Parameter übergeben worden sind, gleich sind. Diese zwei

Spalten, die gleich sein müssen, werden dann im Ergebnis zusammengefügt. Anschließend wird eine Hash-Tabelle von der `hashTable` erstellt. Hierzu wird die Klasse `HashMap` benutzt, die von Java zur Verfügung gestellt wird. Als "Key" dienen hierbei die Spalten der `hashTable`, die der Klasse mit übergeben worden sind und auf deren Gleichheit auch später überprüft wird. Zu jedem Key existiert ein `Vector`, der dann für jedes hinzugefügte Tupel den Key und die Daten enthält. Nachdem bei Hashing auch mehrere verschiedene Einträge mit unterschiedlichen Keys auf den selben Hash-Wert abgebildet werden können, muss der Key immer explizit mit abgespeichert werden, um später ein einfacheres Vergleichen zu ermöglichen. Nachdem diese Hash-Tabelle existiert, wird die `probeTable` Schritt für Schritt abgearbeitet und die als Parameter übergebenen Spalten werden auf Gleichheit überprüft. Dieser Schritt wird `Probing` genannt. Hierzu wird für jedes Tupel, das überprüft wird, wieder der Hash-Wert von dem Key (das sind die jeweiligen übergebenen Spalten) berechnet. Danach wird in der Hash-Tabelle nach diesem Wert gesucht. Ist dieser nicht vorhanden, wird das momentan betrachtete Tupel der `RighTable` nicht weiter beachtet, und man geht zu dem nächsten Tupel über. Existiert ein Eintrag für den Hash-Wert in der Hash-Tabelle, so muss noch die Gleichheit der Keys überprüft werden. Hinter dem Hash-Eintrag liegt ein `Vector`, der zu jedem Tupel ein Key und Data hat. Stimmt bei einem Vektoreintrag der Key mit dem Key des momentan betrachteten Tuples überein, so wird dieses Tupel der neuen Tabelle, die letztendlich das Ergebnis beinhalten wird, hinzugefügt.

### 5.6.6 MySingleObjectCursor

*MySingleObjectCursor* (HashCodeArrayTuple object)

Diese Klasse ist von Ralf Rantau, Institut für Parallele und Verteilte Systeme, Universität Stuttgart implementiert worden und musste noch durch eine weitere Methode angepasst werden. Ein Objekt dieser Klasse ist ein Cursor, der nur ein einziges Objekt - in diesem Fall ein Tupel - enthält. Durch den Konstruktor wird eine Instanz mit einem Tupel erzeugt. Wichtig sind noch die Methode `reset()`, die den Pointer des Cursors wieder auf den Anfang setzt und die Methode `put(HashCodeArrayTuple)`, die das vorhandene Tupel durch das neue ersetzt.

### 5.6.7 XXL - NestedLoopsJoin

*NestedLoopsJoin* (MetaDataCursor cursor1  
 MetaDataCursor cursor2  
 Function newCursor  
 Function createTuple  
 int TYPE)

Diese Klasse implementiert einen Nested Loop Join. Es kann sowohl ein `Thetajoin`, als auch ein `Leftouterjoin`, `Rightouterjoin`, `Outerjoin`, `Naturaljoin`, `Semijoin` und `Cartesian-product` als `TYPE` übergeben werden. Als weiterer Parameter kommt noch hinzu, welche

Art von Tupel der neu berechnete Cursor beinhalten soll. Im Folgenden sind das immer *HashCodeArrayTuple*. Es soll hier nicht näher auf diese Klasse eingegangen werden. Man begnügt sich damit, dass die korrekte Funktion realisiert worden ist, und wendet diese an.

### 5.6.8 XXL - MergeSorter

<i>MergeSorter</i>	(MetaDataCursor	cursor
	java.util.Comparator	comparator
	int	objectSize
	int	memSize
	int	finalMemSize
	Function	newQueue
	boolean	verbose)

In dieser Klasse, ebenso schon von XXL vorgegeben, wird ein Sortieralgorithmus implementiert. Dieser Klasse werden folgende Parameter übergeben: Einen Cursor, der sortiert werden soll, ein Comparator, der angibt auf welchen Spalten und in welcher Reihenfolge sortiert wird, eine Funktion, um zu bestimmen wie die Daten intern materialisiert werden. Mit dieser Funktion lässt sich somit bestimmen, ob ein interner Sortieralgorithmus angewendet werden soll, indem die Daten im Hauptspeicher gehalten werden, oder ein externer Algorithmus, wobei die Daten auf Festplatte geschrieben werden. Ebenso sind noch weitere Eingabemöglichkeiten, wie zum Beispiel Parameter zur Speicherverwaltung, möglich.



## 6 Messungen

In diesem Kapitel wird auf die Ergebnisse eingegangen, die sich bei dem DBVS und in Java ergeben haben. Diese Ergebnisse werden erläutert und interpretiert.

### 6.1 SQL - Ergebnisse

Die Messungen erfolgten auf einem 4-CPU Intel Pentium III 900 MHz PC mit 4 GB RAM unter dem Betriebssystem Microsoft Windows 2000 Server und dem DBVS Microsoft SQL Server 2000 Standard Edition. Es wurden die in Kapitel 4 beschriebenen und implementierten SQL-Statements auf die in Kapitel 3 genannten Transaktionstabellen angewendet.

Im Folgenden werden die Ergebnisse auf den Transaktionstabellen T10.I5.D100k und T10.I5.D100k\_modified präsentiert. Die Charakteristika der Transaktionstabellen sind in Kapitel 3 beschrieben.

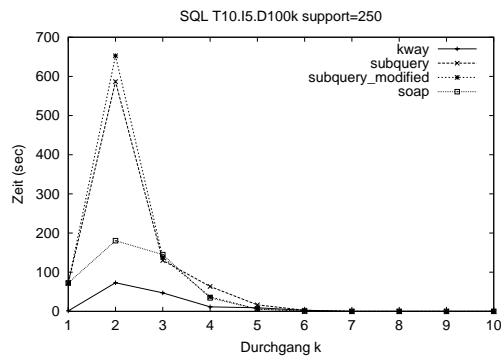


Abbildung 10: Messergebnisse SQL T10.I5.D100k support=250

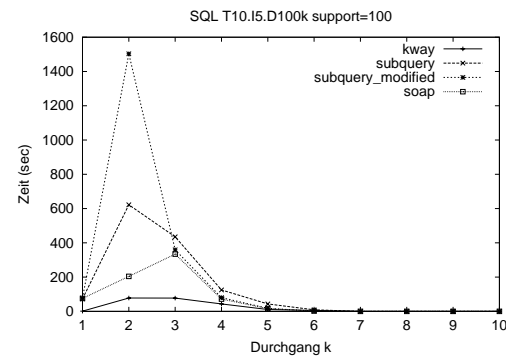


Abbildung 12: Messergebnisse SQL T10.I5.D100k support=100

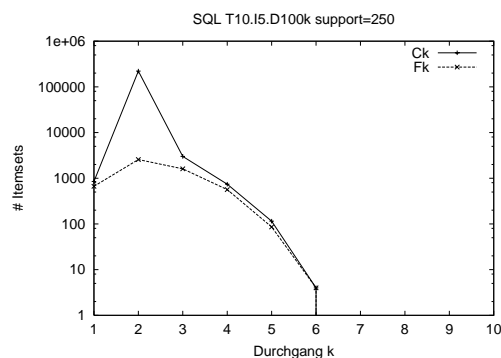


Abbildung 11: Charakteristika der  $F_k^h$ - und  $C_k^h$ -Tabellen bei T10.I5.D100k und support=250

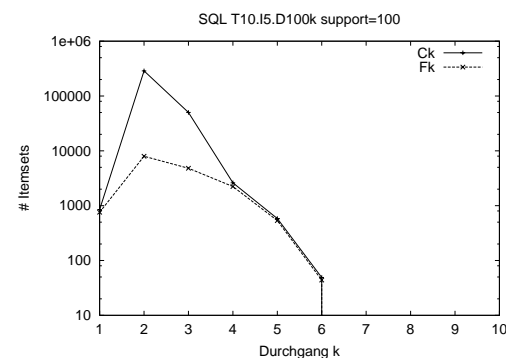


Abbildung 13: Charakteristika der  $F_k^h$ - und  $C_k^h$ -Tabellen bei T10.I5.D100k und support=100

In Abbildung 10 ist das Ergebnis für die Transaktionstabelle T10.I5.D100k mit einem Supportwert von 250 abgebildet. Man erkennt, dass K-Way Join schneller ist als SOAP und Subquery. In Abbildung 11 erkennt man die Anzahl der berechneten Itemsets der  $C_k^h$ - und  $F_k^h$ -Tabellen. Da die Anzahl der ItemsetIDs der einzelnen Tabellen stark voneinander abweichen, wurde diese Abbildung mit einem logarithmischen Maßstab dargestellt. Da die  $F_1^h$  Tabellen viele Einträge enthalten, wird die Tabelle  $C_2^h$  sehr groß. Es werden alle möglichen Zweierkombinationen an Itemsets, die sich aus der Tabelle  $F_1^h$  bilden lassen, in der Tabelle  $C_2^h$  eingetragen. Analog erkennt man in Abbildung 12 und 13 die Messergebnisse für die gleiche Transaktionstabelle, aber mit einem Supportwert von 100.

Nachfolgend sind die Messergebnisse für die Tabelle T10.I5.D100k\_modified und den Supportwerten von 250 und 100 abgebildet:

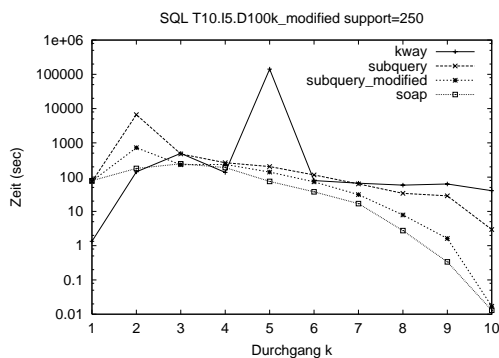


Abbildung 14: Messergebnisse SQL T10.I5.D100k\_modified support=250

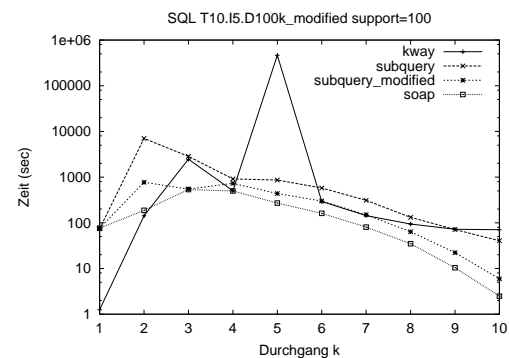


Abbildung 16: Messergebnisse SQL T10.I5.D100k\_modified support=100

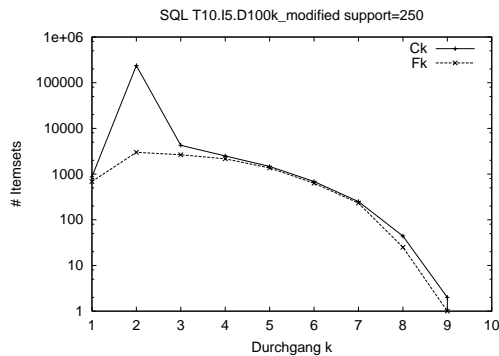


Abbildung 15: Charakteristika der  $F_k^h$ - und  $C_k^h$ -Tabellen bei T10.I5.D100k\_modified und support=250

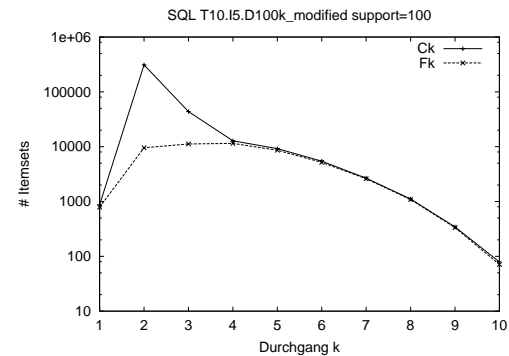


Abbildung 17: Charakteristika der  $F_k^h$ - und  $C_k^h$ -Tabellen bei T10.I5.D100k\_modified und support=100

Es wird deutlich, dass nun ein ganz anderes Messergebnis zustande gekommen ist. Vor allem ist nun K-Way Join um ein Vielfaches langsamer. Es ist zu beachten, dass auch diese Ausführungszeiten mit einem logarithmischen Maßstab dargestellt sind. Vor allem wird bei K-Way Join für den Durchgang 5 ein sehr großer Ausschlag erkennbar. Für die



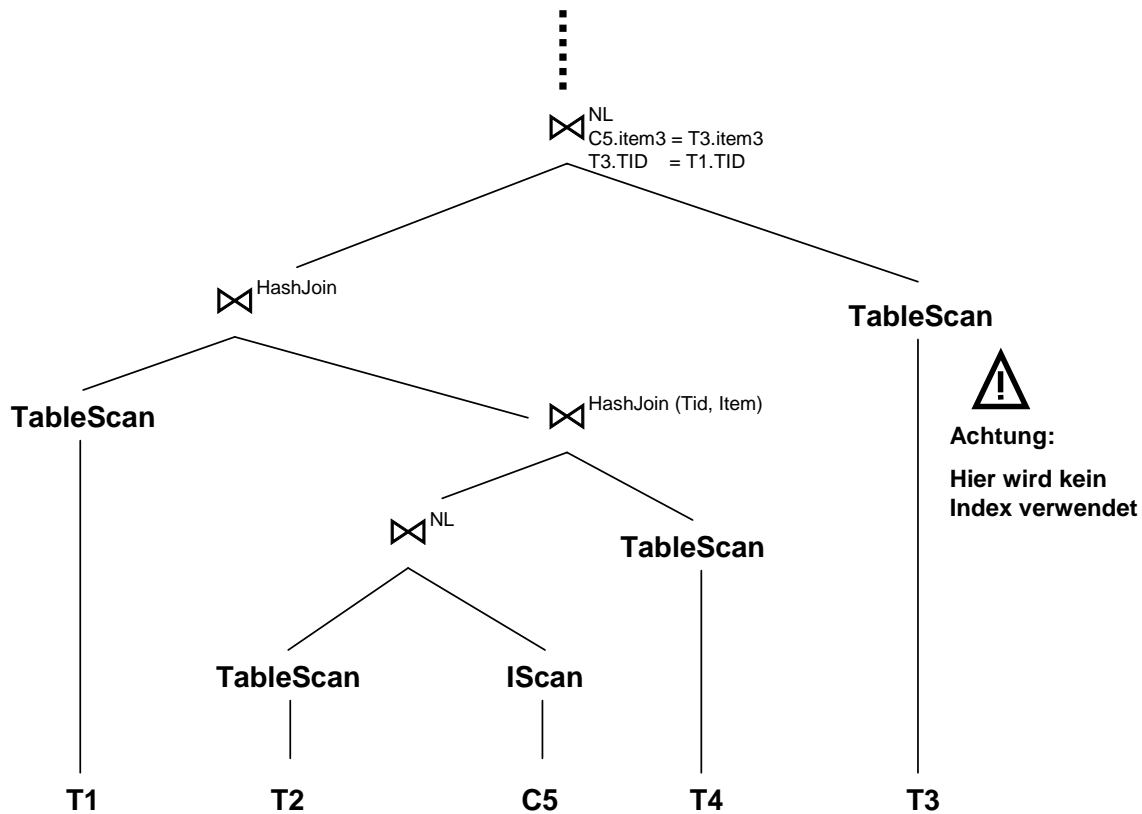


Abbildung 19: Ausführungsplan für K-Way Join - T10.I5.D100k\_modified support=250

Bei den originalen T10.I5.D100k Transaktionstabellen wurden Frequent Itemsets bis  $F_6^h$  berechnet. Hier zeigen sich die Vorzüge des SOAP-Algorithmus, der am schnellsten ist.

Bisher wurden die Transaktionstabellen T10.I5.D100k und T10.I5.D100k\_modified betrachtet. Nachfolgend werden diese mit den Ergebnissen der zwei Tabellen T5.I5.D100k und T5.I5.D100k\_modified verglichen. Bei diesen Tabellen ist lediglich die durchschnittliche Anzahl der Items pro Transaktion von 10 auf 5 verringert worden. Dennoch sehen die Messergebnisse sehr unterschiedlich aus, wie den folgenden Abbildungen entnommen werden kann. Es folgen die Ergebnisse mit einem Supportwert von 250 und 100.

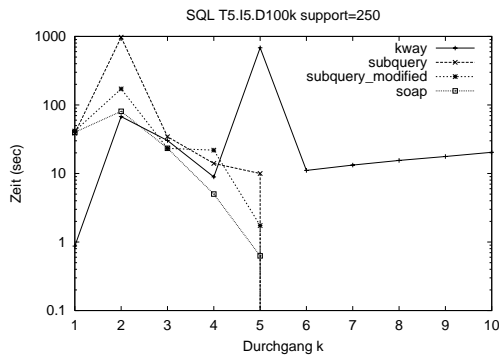


Abbildung 20: Messergebnisse SQL T5.I5.D100k support=250

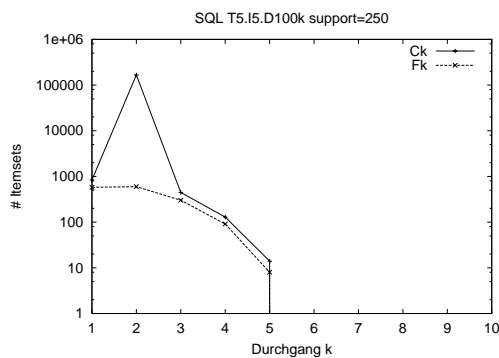
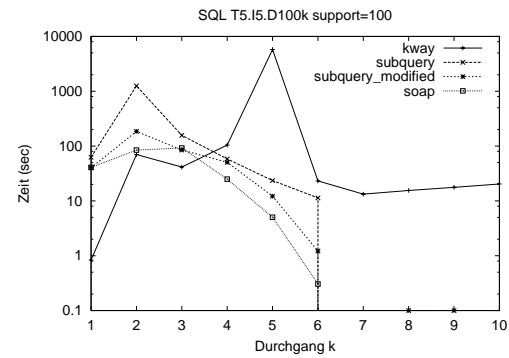
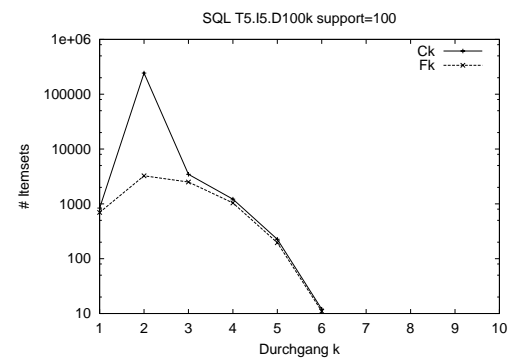
Abbildung 21: Charakteristika der  $F_k^h$ - und  $C_k^h$ -Tabellen bei T5.I5.D100k und support=250

Abbildung 22: Messergebnisse SQL T5.I5.D100k support=100

Abbildung 23: Charakteristika der  $F_k^h$ - und  $C_k^h$ -Tabellen bei T5.I5.D100k und support=100

Man erkennt, dass für jeden Durchlauf  $k$  sehr unterschiedliche Ergebnisse erzielt werden, die zu diesem “zickzack” in Abbildung 20 und 22 führen. Der K-Way Join Algorithmus benötigt für die Durchgänge  $k=6$  bis 10 noch jeweils ungefähr 15 Sekunden, obwohl die Kandidatentabellen  $C_6^h$  bis  $C_{10}^h$  keine Einträge mehr enthalten. Der Grund für dieses Verhalten ist leider unklar, vor allem da bei den vorigen Tabellen T10.I5.D10k und T10.I5.D10k\_modified sich dieses Verhalten nicht gezeigt hat.

Der Algorithmus SOAP erzielt das beste Ergebnis, danach folgt der modifizierte Subquery Algorithmus, dann der originale Subquery und schließlich K-Way Join. Es folgen die Messergebnisse der Transaktionstabelle T5.I5.D100k\_modified mit einem Supportwert von 250 und 100.

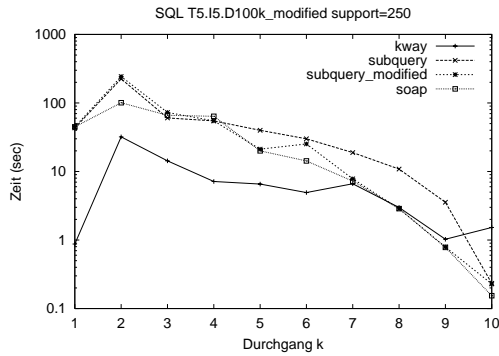


Abbildung 24: Messergebnisse SQL T5.I5.D100k\_modified support=250

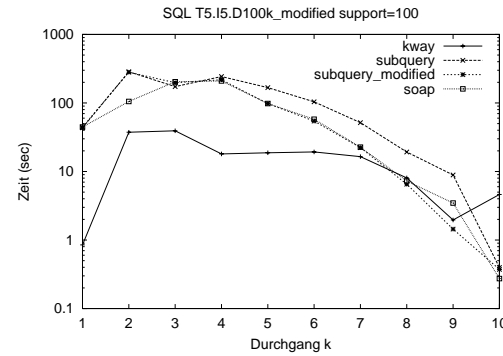


Abbildung 26: Messergebnisse SQL T5.I5.D100k\_modified support=100

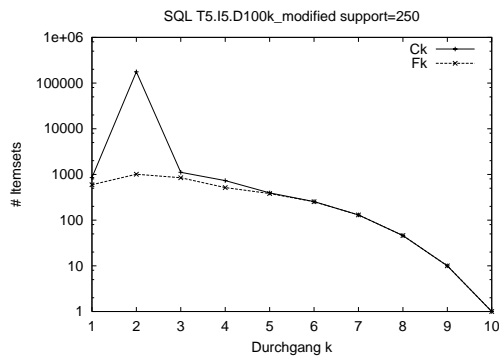


Abbildung 25: Charakteristika der  $F_k^h$ - und  $C_k^h$ -Tabellen bei T5.I5.D100k\_modified und support=250

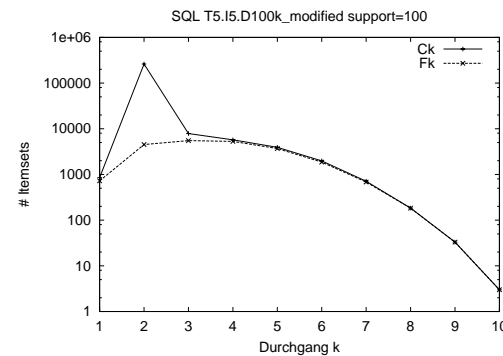


Abbildung 27: Charakteristika der  $F_k^h$ - und  $C_k^h$ -Tabellen bei T5.I5.D100k\_modified und support=100

Wiederum ist zu erkennen, dass die modifizierte Transaktionstabelle bis zu  $k=10$  Ergebnisse liefert. Dieses Mal jedoch ist K-Way Join der schnellste Algorithmus.

Bei den bisherigen Ergebnissen ist erkennbar, dass die Charakteristika der Transaktionstabellen auf die Geschwindigkeit der jeweiligen Algorithmen einen großen Einfluß haben.

Bisher ist der neue Ansatz Quiver bei den Ergebnissen nicht aufgetaucht. Das liegt daran, dass Quiver keine Ergebnisse geliefert hat. Der Microsoft SQL Server hatte 5 GB freien Festplattenspeicher. Wurden die SQL-Statements des Quiver-Algorithmus ausgeführt, so brach das DBVS mit der Fehlermeldung ab, dass für das Log-File kein weiterer Plattenplatz zur Verfügung steht, nachdem es bis auf 5 GB angewachsen ist. Daher werden im Folgenden die kleineren Transaktionstabellen T5.I5.D10k und T5.I5.D5k für die Messungen benutzt. Es werden die Algorithmen K-Way Join, SOAP und Quiver gegenübergestellt. Auf die Algorithmen Subquery und Subquery\_modified wurde verzichtet, da sich diese immer zwischen den Zeiten von K-Way Join und SOAP einfügten. Somit wurde wegen der Übersicht auf diese zwei Kurven in den folgenden Abbildungen verzichtet.

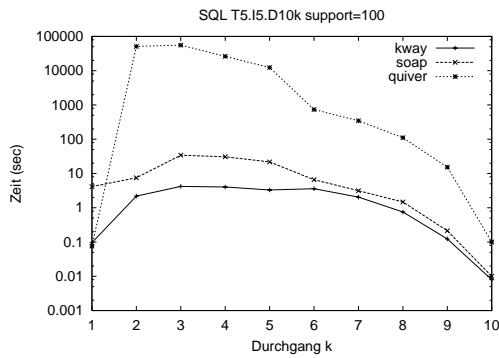


Abbildung 28: Messergebnisse SQL T5.I5.D10k support=100

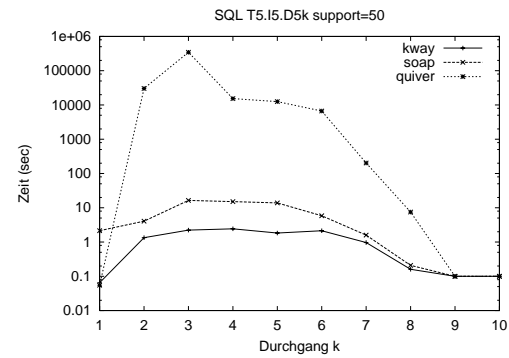
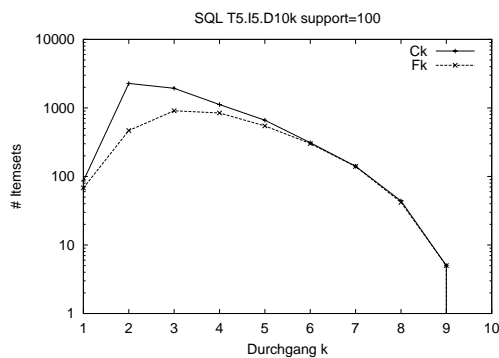
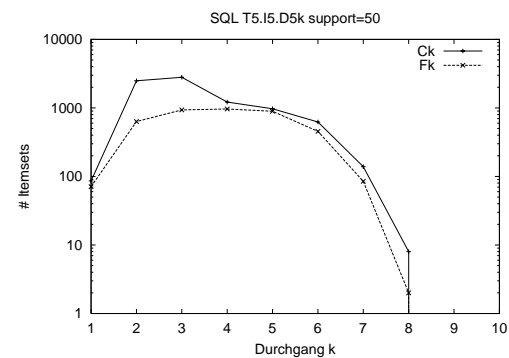


Abbildung 30: Messergebnisse SQL T5.I5.D5k support=50

Abbildung 29: Charakteristika der  $F_k^h$ - und  $C_k^h$ -Tabellen bei T5.I5.D10k und support=100Abbildung 31: Charakteristika der  $F_k^h$ - und  $C_k^h$ -Tabellen bei T5.I5.D5k und support=50

Der neue Algorithmus Quiver ist am langsamsten. Nachdem die Idee des Quiver, die Division, in SQL mit zwei “Not Exists” Anweisungen realisiert worden ist (siehe auch Kapitel 4), waren diese Ergebnisse auch zu erwarten. Hier ist auch deutlich sichtbar, dass K-Way Join schneller ist als SOAP. Dies liegt daran, dass die Tabellen sehr klein sind. SOAP benötigt einige Zeit, um die Tabellen  $T_f$  und  $T_3-T_{10}$  zu erstellen. Nachdem aber nur wenige Itemsets darin dann enthalten sind, sind auch die Performancegewinne, die durch diese Tabellen mit den Vorinformationen erzielt werden können, marginal und werden bei weitem wieder durch die Erstellung dieser Tabellen aufgebraucht.

Allein die Erstellung der Kandidatentabellen mit vertikalem Aufbau dauert viel länger, als die Zeit für die Erstellung der Tabellen mit horizontalem Aufbau. Dies wird in folgender Abbildung 32 deutlich veranschaulicht. Es werden die Zeiten für die Generierung der vertikalen und der horizontalen Kandidatentabellen bei der Transaktionstabelle T5.I5.D10k und einem Supportwert von 100 gegenübergestellt. Für die horizontalen Kandidatentabellen sind die Zeiten für die  $C_k^h$ -Tabellen herangezogen worden. Bei den vertikalen Kandidatentabellen müssen die Zeiten der  $P_k$ - (die Präfix-Tabellen) und die Zeiten der  $C_k^v$ -Tabellen addiert werden, da die Präfix-Tabellen zur Berechnung der  $C_k^v$ -Tabellen

benötigt werden.

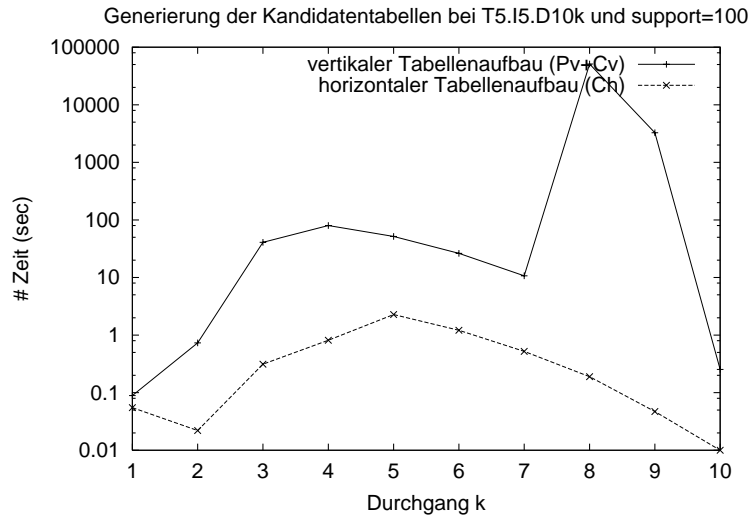


Abbildung 32: horiz. und vert. Kandidatentabellenerstellung

Abschließend folgt noch ein Vergleich bei dem vertikalen Tabellenaufbau. Dass die Zeit für die Generierung der Kandidatentabellen schon lang dauert, kann der obigen Abbildung 32 entnommen werden. Es wird nun noch der Algorithmus Quiver dem K-Way Join vertikal Algorithmus gegenübergestellt. Der folgenden Abbildung 33 können die Zeiten für die Algorithmen Quiver und K-Way Join vertikal entnommen werden. Als Referenz wird noch die Zeit für den originalen K-Way Join Algorithmus abgebildet, der auf den horizontalen Tabellen durchgeführt worden ist. Es wurde hierzu wieder die Transaktionstabelle T5.I5.D10k mit einem Supportwert von 100 benutzt.

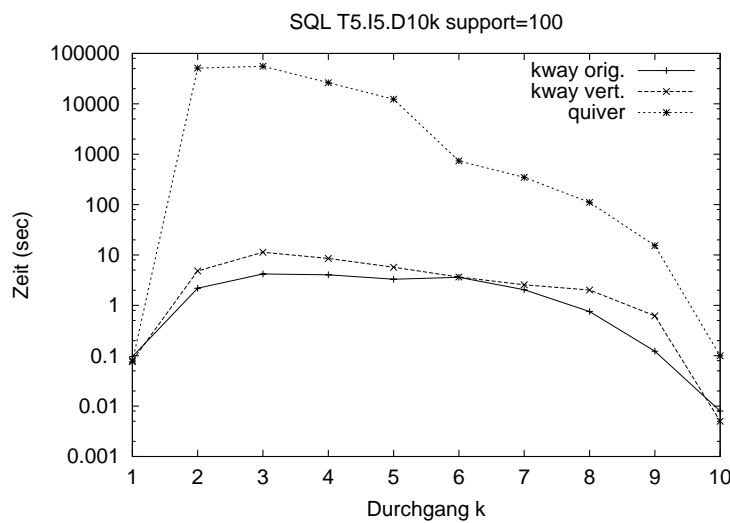


Abbildung 33: Vergleich Quiver - K-Way Join vert. - K-Way Join orig.

Es lässt sich deutlich erkennen, dass durch die Modifikationen des K-Way Join Algorithmus - um den K-Way Join vertikal Algorithmus zu realisieren - Performanceeinbußen hingenommen werden müssen. Allerdings ist der K-Way Join vertikal Algorithmus bei weitem noch schneller als der Ansatz mit Quiver. Ob dies allein daran liegt, dass die Division in SQL nicht ausgedrückt werden kann und daher Änderungen notwendig waren, oder ob auch andere Faktoren mitspielen, soll mit den XXL Ergebnissen im nächsten Kapitel geklärt werden.

Alle SQL Ergebnisse im Überblick sind im Anhang abgedruckt.

## 6.2 XXL - Ergebnisse

Die Messungen der Algorithmen K-Way Join, Quiver und Division unter Java wurden auf einem 4-CPU Sun Ultra 80 Server mit Sun UltraSPARC-II 450 MHz CPU, 4 GB RAM, Sun Solaris, JRE 1.4.1 durchgeführt. Nachdem mit Java die Berechnung der Algorithmen viel länger dauert als mit dem DBVS, mussten zuerst die Transaktionstabellen angepasst werden. Mit der Transaktionstabelle T5.I5.D10k konnten noch in akzeptabler Zeit Ergebnisse erzielt werden. Daher wurden die Transaktionstabellen T5.I5.D10k und T5.I5.D5k - die auch schon bei dem DBVS verwendet worden sind - mit Java benutzt. Zwei Transaktionstabellen sind zum Vergleichen etwas wenig, daher wurden aus der kleinen Transaktionstabelle T5.I5.D5k zufällige Transaktionen herausgeschnitten. Somit entstanden zusätzlich die Tabellen T5.I5.D1k, T5.I5.D500 und T5.I5.D50. Die Transaktionstabellen wurden danach in eine Textdatei geschrieben, die mit Java eingelesen werden kann.

Im Folgenden werden mit Java durch die Algorithmen jeweils die  $S_4^v$ -Tabellen berechnet. Dazu werden Transaktionstabellen und Kandidatentabellen benötigt. Um verschiedene Kandidatentabellen testen zu können, wird die originale Kandidatentabelle  $C_4^v$  verwendet, die durch SQL-Statements für die Transaktionstabelle T5.I5.D10k und den Supportwert=100 berechnet wurde. Aus dieser Tabelle  $C_4^v$  wurden 128, 64, 32, 16, 8 und 4 Itemsets ausgewählt. Somit erhält man sechs verschiedene Kandidatentabellen, die unter Java zur Berechnung der  $S_4^v$  benutzt werden können. Die Kandidatentabellen wurden so erstellt, dass auch bei derjenigen mit nur vier Itemsets mindestens eines immer als Frequent in den fünf verschiedenen Transaktionstabellen vorkommt.

Es wurden zusätzlich zu den bisher betrachteten Transaktionstabellen noch sogenannte "Real-Life-Datensätze" untersucht. Hierzu wird die Tabelle t\_bms\_wv2 von Blue Martini Software [11] benutzt. In dieser Tabelle ist das Surfverhalten der Benutzer einer E-Commerce Webseite festgehalten. Jede Transaktion ist eine Websession, die die Produktdetails enthält, die in der Session betrachtet worden sind.

Diese Tabelle wurde wiederum etwas gekürzt, um eine vergleichbare Anzahl an Transaktionen zu erreichen. Man erhält somit die Transaktionstabellen tbmswv2\_D10k, tbmswv2\_D5k, tbmswv2\_D1k, tbmswv2\_D500 und tbmswv2\_D50. Analog zu den oben genannten Kandidatentabellen wurden hier wieder mit SQL-Statements die fünf verschiedenen Kandidatentabellen erstellt. Die originale Kandidatentabelle  $C_4^v$  der Transaktionstabelle t\_bms\_wv2 wurde mit einem Supportwert = 100 berechnet.

Die Charakteristika der Transaktionstabellen, die im Folgenden benutzt werden, sind in Tabelle 10 abgebildet.

Transaktions-tabelle	#Trans-aktionen	#Zeilen	#unterschiedl. Items	durchschnittl. Itemsetgröße	max. Itemsetgröße
T5.I5.D10k	10000	58964	86	5.900	17
T5.I5.D5k	5000	30268	86	6.050	18
T5.I5.D1k	1000	5942	85	5.940	15
T5.I5.D500	500	2862	84	5.720	14
T5.I5.D0	50	282	65	5.640	11
tbmswv2_D10k	10000	39900	2429	3.990	135
tbmswv2_D5k	5000	19971	2088	3.990	135
tbmswv2_D1k	1000	4096	1287	4.100	65
tbmswv2_D500	500	2151	937	4.300	65
tbmswv2_D50	50	249	190	4.980	42

Tabelle 10: Charakteristika der Transaktionstabellen

### 6.2.1 Synthetische Daten

Zu Beginn wurden die Algorithmen K-Way Join und Quiver, die anhand des Ausführungsplans in Java implementiert worden sind, getestet. Anschließend wird der neue Ansatz der Division, der durch die Klasse *SetContainmentDivision* implementiert worden ist, untersucht. Es wurden erst die von den SQL-Ergebnissen bekannten Transaktionstabellen T5.I5.t10k und T5.I5.D5k verwendet. Die Ergebnisse sind in Abbildung 34 und 35 zu sehen:

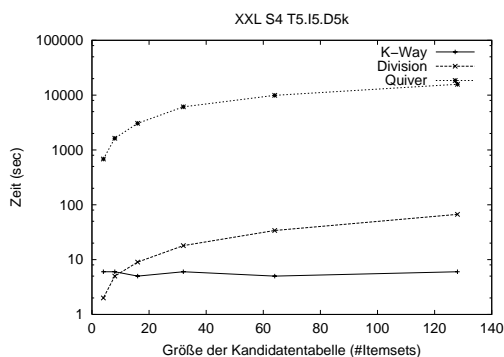


Abbildung 34: Messergebnisse XXL T5.I5.D5k

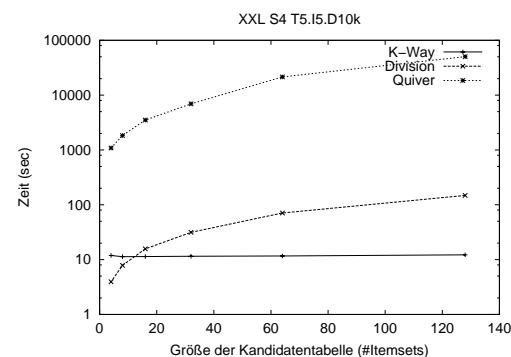


Abbildung 35: Messergebnisse XXL T5.I5.D10k

Man erkennt, dass Quiver - wie schon beim SQL Server - erheblich länger zur Berechnung der  $S_4^p$ -Tabellen benötigt. Die Abbildung 34 ist im logarithmischen Maßstab

abgebildet. Man erkennt daher deutlich, dass Quiver viel mehr Zeit als die beiden anderen Algorithmen benötigt hat. Interessant ist nun, wie lange der neue Ansatz der Division - die Klasse *SetContainmentDivision* - benötigt. Es ist ersichtlich, dass sie vor allem für größere Kandidatentabellen länger als K-Way Join dauert, bei weitem aber nicht mehr die lange Laufzeit des Quiver-Algorithmus hat. Für die kleinen Kandidatentabellen mit vier und acht Itemsets ist dieser Ansatz sogar schneller als der K-Way Join Algorithmus.

Ein weiteres Ergebnis lässt sich in den Abbildungen 36, 37 und 38 erkennen.

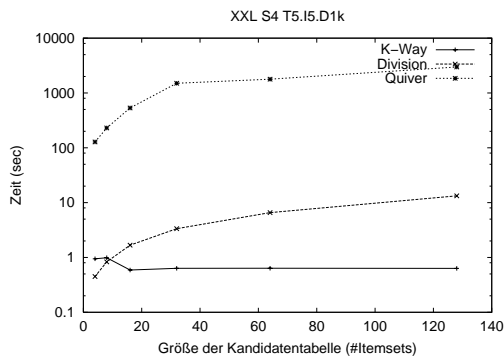


Abbildung 36: Messergebnisse XXL T5.I5.D1k

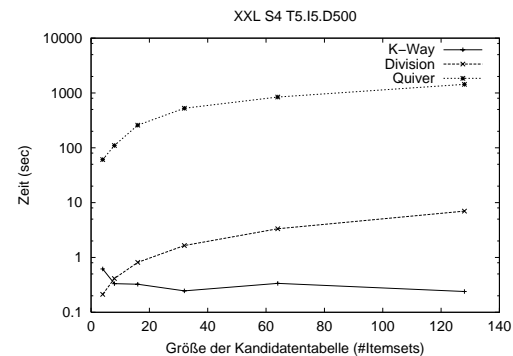


Abbildung 37: Messergebnisse XXL T5.I5.D500

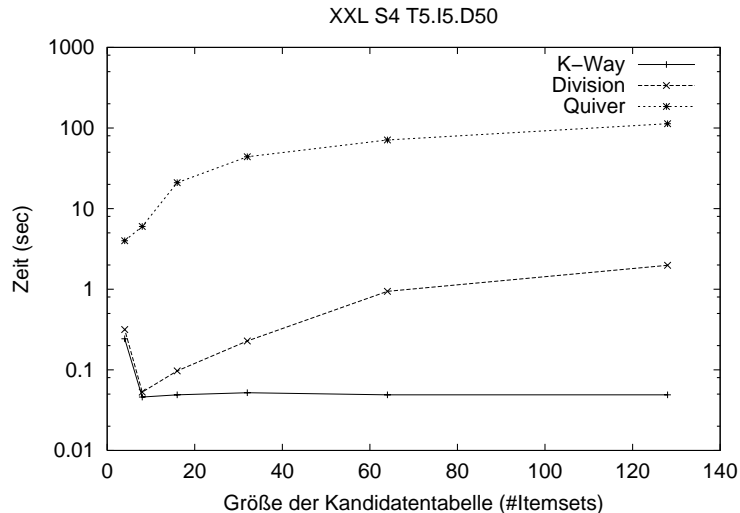


Abbildung 38: Messergebnisse XXL T5.I5.D50

All diese abgebildeten Ergebnisse sind mit Indizes sowohl auf den Transaktionstabellen als auch auf den Kandidatentabellen entstanden. K-Way Join wurde auch ohne die Verwendung der Indizes ausgeführt, was zu dem Ergebnis in Abbildung 39 führte. Man erkennt, dass auch unter Java mit der Klassenbibliothek XXL die Verwendung von Indizes zu einem Performancegewinn mit ungefähr dem Faktor 1.25 führt. Allerdings ist dies

nicht immer der Fall. Bei Quiver kann die Verwendung von Indizes zu Performanceeinbußen führen, wie in Abbildung 40 zu erkennen ist.

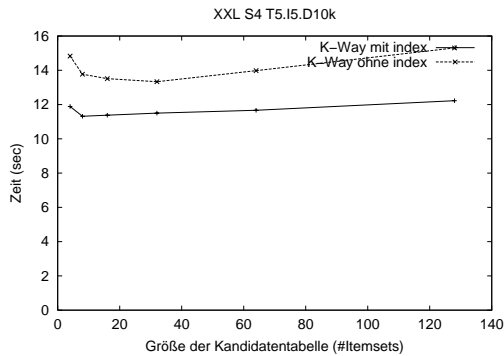


Abbildung 39: Messergebnisse XXL K-Way Join T5.I5.D10k

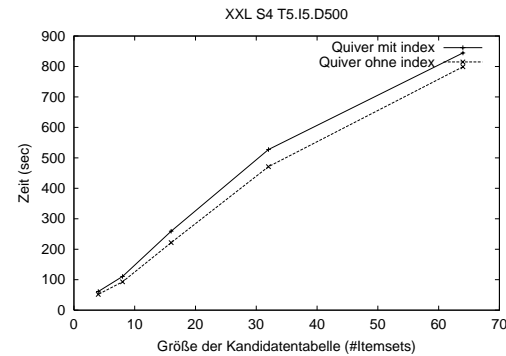


Abbildung 40: Messergebnisse XXL Quiver T5.I5.D500

Ebenso hat die Verwendung von Indizes bei Quiver mit den Transaktionstabellen T5.I5.D500 bis T5.I5.D10k zu Performanceeinbußen geführt. Lediglich bei der Transaktionstabelle T5.I5.D50 hat Quiver mit Hilfe der Indizes Performancegewinne gebracht - siehe Abbildung 41.

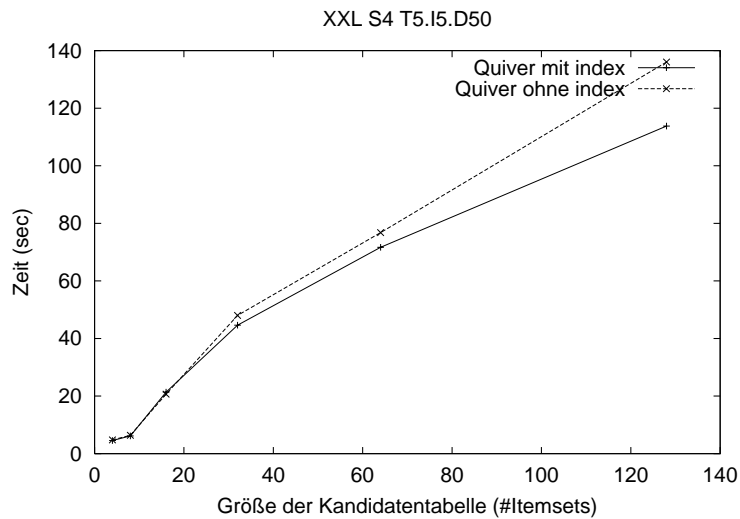


Abbildung 41: Messergebnisse XXL Quiver T5.I5.D50

Allerdings muss auch beachtet werden, dass die Erzeugung der Indizes Zeit in Anspruch nimmt. Bei den Kandidatentabellen ist diese Zeit zu vernachlässigen, bei den Transaktionstabellen dauert die Erstellung der B-Bäume für den Index jedoch einige Zeit, wie es in Abbildung 42 und 43 zu erkennen ist.

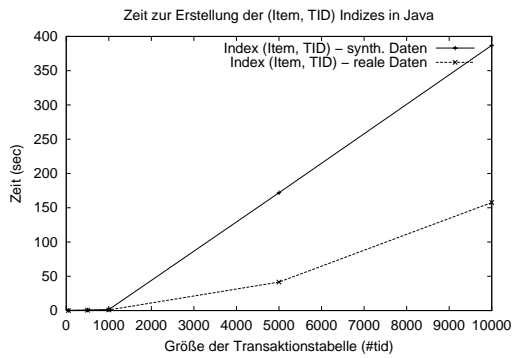


Abbildung 42: Erstellung des Index (Item, TID)

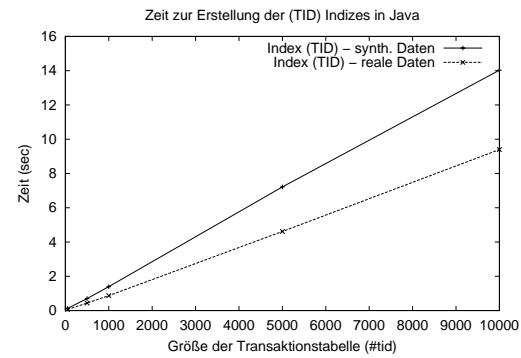


Abbildung 43: Erstellung des Index (TID)

### 6.2.2 Reale Anwendungsdaten

Erstaunlicherweise ergeben sich bei diesen Transaktionsdaten - obwohl sie in der Größe ungefähr den synthetischen Daten entsprechen - Messergebnisse, die sehr unterschiedlich zu den bisher gewonnenen Ergebnissen sind. Es liefen alle Algorithmen bei den entsprechenden Transaktionstabellen gleicher Größe schneller. Ein Vergleich der bisherigen synthetischen Transaktionstabellen und der neuen Real Life Datensätze für die Algorithmen K-Way Join, Division und Quiver kann den Abbildungen 44, 45 und 46 entnommen werden.

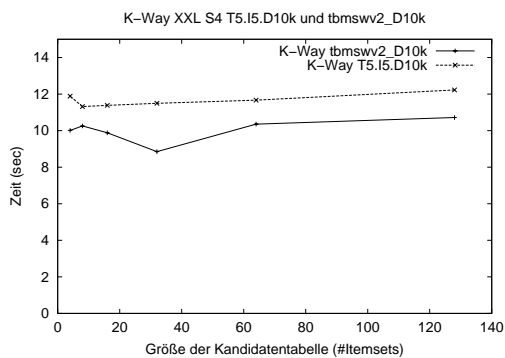


Abbildung 44: K-Way Join - XXL - T5.I5.D10k und tbmswv2\_D10k

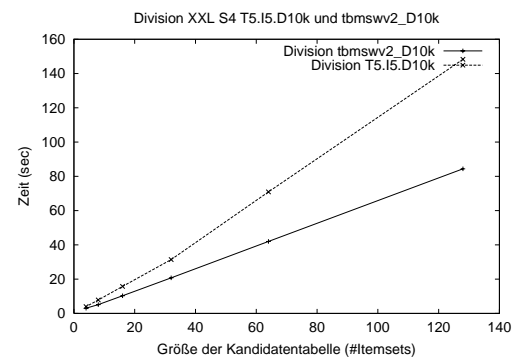


Abbildung 45: Division - XXL - T5.I5.D10k und tbmswv2\_D10k

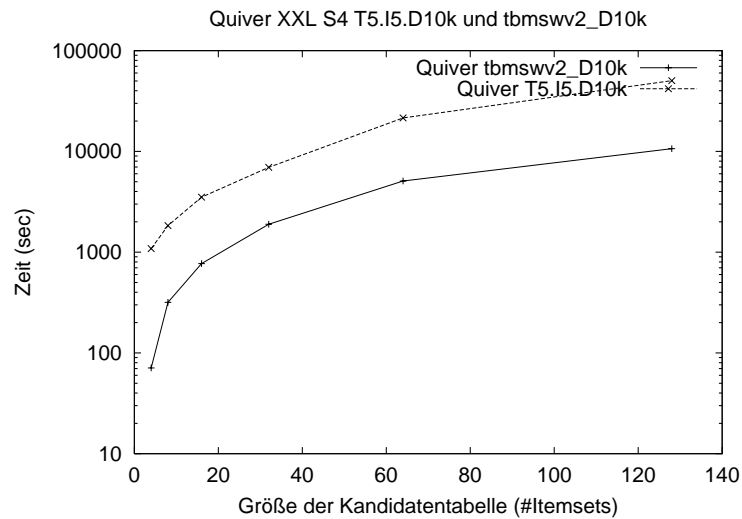


Abbildung 46: Quiver - XXL- T5.I5.D10k und tbmswv2\_D10k

Im Fokus der Leistungsmessungen steht das Ergebnis für Quiver, welches in Abbildungen 47 und 48 dargestellt ist.

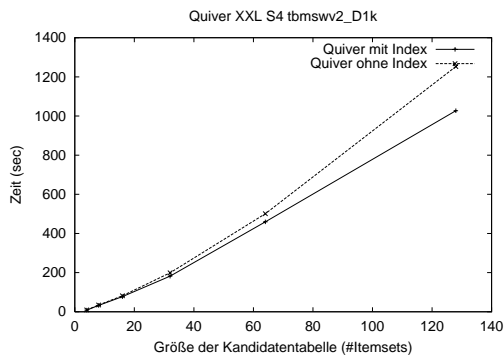


Abbildung 47: Messergebnisse XXL Quiver tbmswv2\_D1k

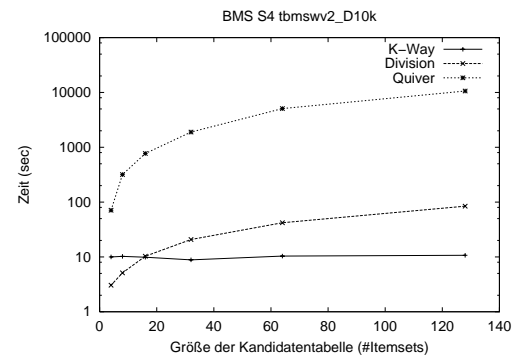


Abbildung 48: Messergebnisse XXL Vergleich K-Way Join - Division - Quiver mit tbmswv2\_D10k

Es ist erkennbar, dass - wie bereits erwähnt - der Algorithmus für die Real Life Transaktionstabelle bessere Messergebnisse erzielt und dass nun auch der Index einen Performancegewinn für Quiver bringt. Das sind doch zwei überraschende Ergebnisse, da dies bei den synthetischen Tabellen ganz anders aussah.

### 6.3 Vergleich SQL - XXL

Es hat sich gezeigt, dass die Algorithmen K-Way Join, Subquery und SOAP auf dem DBVS schneller die Frequent Itemsets lieferten, als der neue Ansatz der Division, der in SQL durch Quiver realisiert worden ist. Zumeist lieferte der neue Ansatz aufgrund der Größe der Transaktionstabellen gar keine Messergebnisse, und es mussten sehr kleine Tabellen gewählt werden. Diese waren dann so klein, das K-Way Join bereits schneller als SOAP und Subquery war, da diese Algorithmen ihre Vorzüge nicht ausspielen konnten. Nachdem in Java die Division gegenüber Quiver aber deutlich schneller war, kann man sich überlegen, ob man dies auch unter SQL erreichen würde, sofern ein "Divisions-Operator" vorhanden wäre. Und wahrscheinlich wäre tatsächlich eine deutliche Verbesserung der Ergebnisse zu erwarten. Durch den Ausführungsplan des Algorithmus Quiver konnte man sehen, dass dieser Ansatz sehr komplex realisiert worden ist. Auch die Messergebnisse unter Java zeigen deutliche Schwächen dieses Ansatzes. Die Implementierung der Division unter Java erfolgte in der Studienarbeit [4] ohne große Caching Algorithmen oder andere Strategien, wie das Ablegen der Zwischenergebnisse im Hauptspeicher. Hier könnten noch Verbesserungen einfließen, die die Division effizienter machen würden. Dennoch sind bereits die Ergebnisse im Vergleich zu Quiver um ein Vielfaches besser.



## 7 Zusammenfassung und Ausblick

Frequent Itemset Discovery (FID) fällt in den Bereich Data Mining. Es gibt schon einige SQL-Algorithmen, die diese Aufgabe lösen. Man geht von einer Transaktionstabelle aus, in der zu jeder Transaktion beliebig viele Items zugeordnet sind. Bei der FID will man herausfinden, welche Items und Itemkombinationen häufig in der Transaktionstabelle vorhanden sind. Dazu bestimmt man zuerst die einzelnen Items, die in der Transaktionstabelle häufig - ein zuvor festgelegter Support-Wert - vorkommen. Hat man nun die einzelnen Items berechnet, werden aus diesen alle mögliche Zweierkombinationen berechnet und man kontrolliert für jede Zweierkombination wieder, ob und wie oft sie in der Transaktionstabelle vorkommt. Liegt dieser Wert über dem Support-Wert, so wird diese Zweierkombination zu dem Ergebnis hinzugefügt. Dies wiederholt man für die Durchgänge 3,4... in denen dann die Dreier- und Viererkombinationen berechnet werden.

Die bekannten SQL-Algorithmen teilen das Problem in zwei Phasen auf. Zuerst werden die Kandidaten bestimmt, die überhaupt als "Frequent" in Betracht kommen können. Darauf folgt die Zählphase, in der kontrolliert wird, ob und wie oft die möglichen Kandidaten in den Transaktionstabellen vorkommen.

In dieser Arbeit wird ein neuere Ansatz - die Division - untersucht. Es wird herausgestellt, inwiefern sich diese Idee für das Problem FID nutzen lässt. In der ersten Phase berechnet man wieder die Kandidatentabelle. In der zweiten Phase wird die Transaktionstabelle durch jedes Item der Kandidatentabelle "dividiert". Man berechnet somit für alle Kandidaten, die Transaktionen, die in der Transaktionstabelle den Kandidaten enthalten. Nun müssen noch die Transaktionen gezählt werden, und man erhält den Support-Wert zu jedem Kandidaten. Dieser Schritt wird für die Zweier-, Dreier- und weitere Itemsetkombinationen wiederholt.

In einem Datenbankverwaltungssystem (DBVS) ist ein Divisionsoperator leider nicht vorhanden. Deshalb hat man auf die Quantifizierung zurückgegriffen. Man hat die Division so umgewandelt, dass sich diese durch *AND*-, *OR*-, *NOT*- und *NOT EXISTS*-Operatoren in SQL ausdrücken lässt.

Desweiteren hat man einen vertikalen Tabellenaufbau gewählt. Die bekannten Algorithmen haben die Ergebnisse immer in horizontalen Tabellen abgelegt. Je höher die berechneten Itemsets wurden, desto mehr Attribute hatten die dazugehörigen Tabellen. Bei dem vertikalen Tabellenaufbau ist die Anzahl der Attribute für alle Durchgänge konstant. Sowohl bei den ersten einzelnen Items, als auch bei Zehner-Itemsetkombinationen ist die Tabellenbreite gleich. Daher kommt auch der Name "Quiver" für diesen neuen Ansatz der Division. *QU* steht für Quantification, die nötig war, um die Division mit SQL-Statements auszudrücken und *VER* steht für den vertikalen Tabellenaufbau, den der neue Ansatz benutzt.

Dieser neue Algorithmus wird auf einem DBVS mit den bekannten Algorithmen verglichen. Es hat sich gezeigt, dass Quiver viel langsamer ist. Um zu Überprüfen, ob dies an der Idee der Division, oder an den Umformungen zur Quantifizierung gelegen hat, wurde die Division in Java implementiert. Da auch Referenzergebnisse in Java nötig sind, wurden die Ausführungspläne des DBVS für die Algorithmen K-Way Join - ein bekannter

Algorithmus aus der Literatur - und Quiver genauer untersucht. Mit Hilfe der Klassenbibliothek XXL [2], die einige Datenbankfunktionalitäten wie Joins oder Indexstrukturen bereitstellt, konnten diese Ausführungspläne in Java implementiert werden. Es hat sich gezeigt, dass K-Way Join meist am schnellsten die Ergebnisse berechnet hat, gefolgt von der Division. Der Ansatz Quiver war bei weitem am langsamsten bei der Berechnung.

In wie weit sich dieses Ergebnis auf ein DBVS übertragen lässt, kann schwer gesagt werden. Dennoch sind mit einem Divisionsoperator in SQL wahrscheinlich bessere Ergebnisse als mit Quiver zu erwarten. Ob die Idee der Division - zur Lösung des FID Problems - besser ist, als die bisher bekannten Algorithmen, kann erst gesagt werden, wenn tatsächlich solch ein Operator für SQL realisiert worden ist.

# Anhang

## A SQL-Quellcode

Im Folgenden wird der Quellcode der einzelnen Algorithmen für den Durchgang  $k=4$  abgebildet. Zuerst wird das Erstellen der Kandidaten Tabelle aufgeführt, da die folgenden Algorithmen K-Way Join, Subquery und SOAP auf diese Tabellen zurückgreifen. Nur der Quiver Algorithmus benutzt eigene Tabellen.

### A.1 Candidate-Generation

```
-- Candidate Generation c4
print 'C4'
insert into Ch4 (item1, item2, item3, item4)
select      a.item1, a.item2, a.item3, b.item3
from        Fh3 AS a, Fh3 AS b,
           Fh3 AS c, Fh3 AS d
where       a.item1=b.item1 and
           a.item2=b.item2 and
           a.item3<b.item3 and
           -- Apriori property.
           -- Skip item1.
           c.item1 = a.item2 AND
           c.item2 = a.item3 AND
           c.item3 = b.item3
           AND
           -- Skip item2.
           d.item1 = a.item1 AND
           d.item2 = a.item3 AND
           d.item3 = b.item3

-- updating statistics
print 'updating statistics'
update STATISTICS Ch4 WITH FULLSCAN, ALL, NORECOMPUTE;
```

### A.2 K-Way Join

```
-- Generation of S (itemset, support)
print 'S4'
insert into Sh4 (itemset, support)
select      c4.itemset, count(t1.tid)
from        Ch4 as c4, t5_d100k_flach as t1,
           t5_d100k_flach as t2, t5_d100k_flach as t3,
           t5_d100k_flach as t4
where       c4.item1=t1.item and
           c4.item2=t2.item and
           c4.item3=t3.item and
           c4.item4=t4.item and
           t1.tid=t2.tid and
           t2.tid=t3.tid and
           t3.tid=t4.tid
group by   c4.itemset

-- updating statistics
print 'updating statistics'
update STATISTICS Sh4 WITH FULLSCAN, ALL, NORECOMPUTE;
```

```

print '- - - - -'

-- FI Generation f4 K-Way Join
print 'F4'
insert into Fh4 (itemset, item1, item2, item3, item4)
select      s4.itemset,c4.item1, c4.item2, c4.item3, c4.item4
from        Ch4 as c4, Sh4 as s4
where       s4.itemset=c4.itemset AND
           s4.support>=@minsup

-- updating statistics
print 'updating statistics'
update STATISTICS Fh4 WITH FULLSCAN, ALL, NORECOMPUTE;

```

### A.3 Subquery

```

-- Subquery generation q4
print 'Q4'
insert into Qh4
select d4.item1, d4.item2, d4.item3, d4.item4, t4.tid
from   t10_d100k_flach as t4,
      (
        select d3.item1, d3.item2, d3.item3, t3.tid
        from   t10_d100k_flach as t3,
              (
                select d2.item1, d2.item2, t2.tid
                from   t10_d100k_flach as t2,
                      (
                        select item1, tid
                        from   t10_d100k_flach as t1,
                              (select distinct item1 from Ch4)as d1
                        where  t1.item=d1.item1
                      ) as r1,
                              (select distinct item1,item2 from Ch4)as d2
                        where  r1.item1=d2.item1 and
                               r1.tid=t2.tid and
                               t2.item=d2.item2
                      )as r2,
                              (select distinct item1,item2,item3 from Ch4)as d3
                        where  r2.item1=d3.item1 and
                               r2.item2=d3.item2 and
                               r2.tid=t3.tid and
                               t3.item=d3.item3
                      ) as r3,
                              (select distinct item1,item2,item3,item4 from Ch4)as d4
                where  r3.item1=d4.item1 and
                       r3.item2=d4.item2 and
                       r3.item3=d4.item3 and
                       r3.tid=t4.tid and
                       t4.item=d4.item4
              )
      )
where   r3.item1=d4.item1 and
       r3.item2=d4.item2 and
       r3.item3=d4.item3 and
       r3.tid=t4.tid and
       t4.item=d4.item4

-- updating statistics (index)
print 'updating index'
update STATISTICS Qh4 WITH FULLSCAN, ALL, NORECOMPUTE;
print '- - - - -'

-- Generating S Tables
print 'S4'
insert into Sh4 (itemset, support)
select      c4.itemset, count(tid)
from        Qh4 as q4, Ch4 as c4
where       q4.item1=c4.item1 and
           q4.item2=c4.item2 and
           q4.item3=c4.item3 and

```

```

                q4.item4=c4.item4
group by        c4.itemset
-- updating statistics (index)
print 'updating index'
update STATISTICS Sh4 WITH FULLSCAN, ALL, NORECOMPUTE;
print '-----'
-- FI Generation F4
print 'F4'
insert into     Fh4 (itemset, item1, item2, item3, item4)
select         s4.itemset,c4.item1, c4.item2, c4.item3, c4.item4
from          Ch4 as c4, Sh4 as s4
where         s4.itemset=c4.itemset AND
                s4.support>=@minsup
-- updating statistics (index)
print 'updating index'
update STATISTICS Fh4 WITH FULLSCAN, ALL, NORECOMPUTE;

```

## A.4 Subquery-modified

```

-- Subquery generation q4
print 'Q4'
insert into Qh4
select      d4.item1, d4.item2, d4.item3, d4.item4, t4.tid
from        t5_d100k_flach as t4,
            Qh3 as r3, -- this is the old Q3 Table, which
                    was generated in the last step k=3
            (select distinct item1,item2,item3,item4
             from             Ch4) as d4
where       r3.item1=d4.item1 and
            r3.item2=d4.item2 and
            r3.item3=d4.item3 and
            r3.tid=t4.tid and
            t4.item=d4.item4

-- updating statistics
print 'updating statistics'
update STATISTICS Qh4 WITH FULLSCAN, ALL, NORECOMPUTE;

print '-----'

-- Generating S Tables
print 'S4'
insert into Sh4 (itemset, support)
select      c4.itemset, count(tid)
from        Qh4 as q4, Ch4 as c4
where       q4.item1=c4.item1 and
            q4.item2=c4.item2 and
            q4.item3=c4.item3 and
            q4.item4=c4.item4
group by   c4.itemset

-- updating statistics
print 'updating statistics'
update STATISTICS Sh4 WITH FULLSCAN, ALL, NORECOMPUTE;

print '-----'

-- FI Generation f4
print 'F4'
insert into Fh4 (itemset, item1, item2, item3, item4)
select      s4.itemset,c4.item1, c4.item2, c4.item3, c4.item4
from        Ch4 as c4, Sh4 as s4

```

```

where      s4.itemset=c4.itemset AND
           s4.support>=@minsup

-- updating statistics
print 'updating statistics'
update STATISTICS Fh4 WITH FULLSCAN, ALL, NORECOMPUTE;

```

## A.5 Set Oriented Apriori

```

print 'creating T4'
insert into Th4
select      t3.item1, t3.item2, t3.item3, tf.item, t3.tid
from        Th3 as t3,
           Tf as tf,
           Ch4 as c4

where      t3.item1=c4.item1 and
           t3.item2=c4.item2 and
           t3.item3=c4.item3 and
           tf.item=c4.item4 and
           t3.tid=tf.tid

-- updating statistics
print 'updating statistics'
update STATISTICS Th4 WITH FULLSCAN, ALL, NORECOMPUTE;

print '-----'

print 'S4'
insert into Sh4
select      itemset, count(*)
from        Th4 as t4, Ch4 as c4
where      t4.item1=c4.item1 and
           t4.item2=c4.item2 and
           t4.item3=c4.item3 and
           t4.item4=c4.item4

group by   c4.itemset

-- updating statistics
print 'updating statistics'
update STATISTICS Sh4 WITH FULLSCAN, ALL, NORECOMPUTE;

print '-----'

-- FI Generation f4
print 'F4'
insert into Fh4 (itemset, item1, item2, item3, item4)
select      s4.itemset,c4.item1, c4.item2, c4.item3, c4.item4
from        Ch4 as c4, Sh4 as s4
where      s4.itemset=c4.itemset AND
           s4.support>=@minsup

-- updating statistics
print 'updating statistics'
update STATISTICS Fh4 WITH FULLSCAN, ALL, NORECOMPUTE;

```

## A.6 Quiver

```

SET @k = 4;

DELETE FROM Pv;
print @mymsg + 'p4 begin';

```

```

INSERT
INTO Pv (itemset_old1, itemset_old2)
SELECT itemset1, itemset2
FROM (
SELECT DISTINCT a1.itemset AS itemset1,
                a2.itemset AS itemset2
FROM Fv3 AS a1,
     Fv3 AS a2
WHERE NOT EXISTS (
  SELECT *
  FROM Fv3 AS b1,
       Fv3 AS b2
  WHERE ((b1.itemset = a1.itemset) AND
        (b2.itemset = a2.itemset) AND
        (b1.pos < @k-1) AND
        (b1.pos = b2.pos)) AND
        NOT (b2.item = b1.item)
) AND
EXISTS (
  SELECT b1.item, b2.item
  FROM Fv3 AS b1,
       Fv3 AS b2
  WHERE (b1.itemset = a1.itemset) AND
        (b2.itemset = a2.itemset) AND
        (b1.pos = @k-1) AND
        (b1.pos = b2.pos) AND
        (b1.item < b2.item)
) AND
-- From now on, we skip the item at position p
-- of itemset a1.
-- The next EXIST clause has to be added for
-- each value of p, where 1 <= p <= k-1.

-- Skip item at position p = 1.
EXISTS (
  SELECT a3.itemset
  FROM Fv3 AS a3
  WHERE NOT EXISTS (
    SELECT b1.item, b2.item
    FROM Fv3 AS b1,
         Fv3 AS b2
    WHERE NOT (
      -- Condition 1: 1 <= i < p
      (
        NOT (
          (b1.itemset = a1.itemset) AND
          (b2.itemset = a3.itemset) AND
          (1 <= b2.pos) AND (b2.pos < 1) AND
          (b1.pos = b2.pos)
        ) OR
        (b1.item = b2.item)
      )
    ) AND
    -- Condition 2: p <= i < k-1
    (
      NOT (
        (b1.itemset = a1.itemset) AND
        (b2.itemset = a3.itemset) AND
        (1 <= b2.pos) AND (b2.pos < @k-1) AND
        (b1.pos = b2.pos + 1)
      ) OR
      (b1.item = b2.item)
    )
    ) AND
    -- Condition 3: i = k-1

```

```

        (
            NOT (
                (b1.itemset = a2.itemset) AND
                (b2.itemset = a3.itemset) AND
                (b2.pos = @k-1) AND
                (b1.pos = b2.pos)
            ) OR
            (b1.item = b2.item)
        )
    )
)
AND

-- Skip item at position p = 2.
EXISTS (
    SELECT a3.itemset
    FROM Fv3 AS a3
    WHERE NOT EXISTS (
        SELECT b1.item, b2.item
        FROM Fv3 AS b1,
        Fv3 AS b2
        WHERE NOT (
            -- Condition 1: 1 <= i < p
            (
                NOT (
                    (b1.itemset = a1.itemset) AND
                    (b2.itemset = a3.itemset) AND
                    (1 <= b2.pos) AND (b2.pos < 2) AND
                    (b1.pos = b2.pos)
                ) OR
                (b1.item = b2.item)
            )
            AND
            -- Condition 2: p <= i < k-1
            (
                NOT (
                    (b1.itemset = a1.itemset) AND
                    (b2.itemset = a3.itemset) AND
                    (2 <= b2.pos) AND (b2.pos < @k-1) AND
                    (b1.pos = b2.pos + 1)
                ) OR
                (b1.item = b2.item)
            )
            AND
            -- Condition 3: i = k-1
            (
                NOT (
                    (b1.itemset = a2.itemset) AND
                    (b2.itemset = a3.itemset) AND
                    (b2.pos = @k-1) AND
                    (b1.pos = b2.pos)
                ) OR
                (b1.item = b2.item)
            )
        )
    )
)
) AS tmp;
print @mymsg + 'p4 end';

update STATISTICS Pv WITH FULLSCAN, ALL, NORECOMPUTE;

print '- - - - -'

```

```

DELETE FROM Cv4;
print @mymsg + 'c4 begin';
INSERT
INTO    Cv4
SELECT p.itemset_new, f.pos, f.item
FROM    Fv3 AS f,
        Pv AS p
WHERE   f.itemset = p.itemset_old1
UNION
SELECT p.itemset_new, @k, f.item
FROM    Fv3 AS f,
        Pv AS p
WHERE   f.itemset = p.itemset_old2 AND
        f.pos      = @k-1;
print @mymsg + 'c4 end';

update STATISTICS Cv4 WITH FULLSCAN, ALL, NORECOMPUTE;

print '-----'
-- S Tabellen Generierung nach K-Way vertikal:
DELETE FROM Sv4;
print @mymsg + 's4 begin';
INSERT
INTO    Sv4 (itemset, support)
SELECT  a1.itemset, COUNT(*)
FROM    Cv4 AS a1,
        Cv4 AS a2,
        Cv4 AS a3,
        Cv4 AS a4,
        t5_d100k_flach AS t1,
        t5_d100k_flach AS t2,
        t5_d100k_flach AS t3,
        t5_d100k_flach AS t4
WHERE   a1.itemset = a2.itemset AND
        a1.itemset = a3.itemset AND
        a1.itemset = a4.itemset AND
        t1.tid     = t2.tid     AND
        t1.tid     = t3.tid     AND
        t1.tid     = t4.tid     AND
        a1.item    = t1.item    AND
        a2.item    = t2.item    AND
        a3.item    = t3.item    AND
        a4.item    = t4.item    AND
        a1.pos     = 1          AND
        a2.pos     = 2          AND
        a3.pos     = 3          AND
        a4.pos     = 4

GROUP BY a1.itemset
HAVING  COUNT(*) >= @minimum_support;
print @mymsg + 's4 end';

update STATISTICS Sv4 WITH FULLSCAN, ALL, NORECOMPUTE;

print '-----'

-- S Tabellen Generierung nach Quiver
-- Dies ist der eigentliche "Divisions" Algorithmus !!!!!!!!!!!!!!!

DELETE FROM Sv4;
print @mymsg + 's4 begin';
INSERT
INTO    Sv4 (itemset, support)
SELECT  itemset, COUNT(DISTINCT TID) AS support
FROM    (
        SELECT c1.itemset, t1.tid

```

```

FROM    Cv4 AS c1, mfk AS t1
WHERE   NOT EXISTS (
  SELECT *
  FROM   Cv4 AS c2
  WHERE  NOT EXISTS (
    SELECT *
    FROM   mfk AS t2
    WHERE  NOT (c1.itemset = c2.itemset) OR
            (t2.tid = t1.tid AND
             t2.item = c2.item )
          )
        )
      ) AS relevant
GROUP BY itemset
print @mymsg + 's4 end';

update STATISTICS Sv4 WITH FULLSCAN, ALL, NORECOMPUTE;

print '-----'

DELETE FROM Fv4;
print @mymsg + 'f4 begin';
INSERT
INTO   Fv4 (itemset, pos, item)
SELECT c.itemset, c.pos, c.item
FROM   Cv4 AS c,
       Sv4 AS s
WHERE  c.itemset = s.itemset;
print @mymsg + 'f4 end';

update STATISTICS Fv4 WITH FULLSCAN, ALL, NORECOMPUTE;

```

## B XXL-Quellcode

### B.1 SQL Ausführungsplan für K-Way Join

Der vom SQL erzeugte Ausführungsplan für K-Way Join,  $k = 4$ , Datentabelle = T5.I5.D5k und support = 100:

```
print '-----' -- Generation of S (itemset, support)
print 'S4'
insert into Sh4 (itemset, support) select c4.itemset, count(t1.tid) from ... (see SQL-Statement)
|--Sequence
|--Index Insert(OBJECT:([dbmine2].[boeckdl].[Sh4].[i_s04_support]),
| | SET:([itemset1010]=[Sh4].[itemset], [support1009]=[Sh4].[support], [IdxBmk1008]=[Bmk1007]))
| |--Sort(ORDER BY:([Sh4].[support] ASC, [Sh4].[itemset] ASC), [Bmk1007] ASC)
| |--Table Spool
| |--Clustered Index Insert(OBJECT:([dbmine2].[boeckdl].[Sh4].[pk_s04]),
| | SET:([Sh4].[support]=[Expr1005], [Sh4].[itemset]=[c4].[itemset]))
| |--Sort(ORDER BY:([c4].[itemset] ASC)
| |--Top(ROWCOUNT est 0)
| |--Compute Scalar(DEFINE:([Expr1005]=Convert([Expr1019])))
| |--Hash Match(Aggregate, HASH:([c4].[itemset]) DEFINE:([Expr1019]=COUNT(*)))
| |--Hash Match(Inner Join, HASH:([t4].[TID], [c4].[item1])=([t1].[TID], [t1].[Item]),
| | RESIDUAL:([t1].[TID]=[t4].[TID] AND [c4].[item1]=[t1].[Item]))
| |--Hash Match(Inner Join, HASH:([t3].[TID], [t3].[Item])=([t4].[TID], [c4].[item3]),
| | RESIDUAL:([t3].[TID]=[t4].[TID] AND [c4].[item3]=[t3].[Item]))
| |--Index Scan(OBJECT:([dbmine2].[boeckdl].[t5_d5k].[i_t5_d5k_tid] AS [t3]))
| |--Hash Match(Inner Join, HASH:([c4].[item4], [c4].[item2])=([t4].[Item], [t2].[Item]),
| | RESIDUAL:([c4].[item4]=[t4].[Item] AND [c4].[item2]=[t2].[Item]))
| |--Clustered Index Scan(OBJECT:([dbmine2].[boeckdl].[Ch4].[pk_c04] AS [c4]))
| |--Hash Match(Inner Join, HASH:([t2].[TID])=([t4].[TID]),
| | RESIDUAL:([t2].[TID]=[t4].[TID]))
| |--Index Scan(OBJECT:([dbmine2].[boeckdl].[t5_d5k].[i_t5_d5k_itemtid] AS [t2]))
| |--Index Scan(OBJECT:([dbmine2].[boeckdl].[t5_d5k].[i_t5_d5k_itemtid] AS [t4]))
|--Index Scan(OBJECT:([dbmine2].[boeckdl].[t5_d5k].[i_t5_d5k_itemtid] AS [t1]))
|--Index Insert(OBJECT:([dbmine2].[boeckdl].[Sh4].[i_s04_supportitemset]),
| | SET:([itemset1013]=[Sh4].[itemset], [support1012]=[Sh4].[support], [IdxBmk1011]=[Bmk1007]))
| |--Sort(ORDER BY:([Sh4].[support] ASC, [Sh4].[itemset] ASC), [Bmk1007] ASC)
| |--Table Spool
|--Index Insert(OBJECT:([dbmine2].[boeckdl].[Sh4].[i_s04_itemsetsupport]),
| | SET:([support1016]=[Sh4].[support], [itemset1015]=[Sh4].[itemset], [IdxBmk1014]=[Bmk1007]))
| |--Sort(ORDER BY:([Sh4].[itemset] ASC, [Sh4].[support] ASC), [Bmk1007] ASC)
| |--Table Spool
print 'updating index'
update STATISTICS Sh4 WITH FULLSCAN, ALL, NORECOMPUTE;
print '-----' -- FI Generation f4 K-Way Join
```

### B.2 XXL Quelltext von K-Way Join Algorithmus

Anbei der Quellcode der XXL Implementierung des K-Way Join Algorithmus. Die Erstellung der Indizes ist nicht abgebildet. Die Cursor, die abgebildet sind, sind jeweils durch die Methode MyBtreeQueryCursor erstellt worden, und somit mit einem Index hinterlegt.

```
stopwatch[5].start();

// first join. t2 and t4, where t2.tid=t4.tid
// see also the execution plan of K-Way Join in documentation
// now we look in the btree

kwaycursor[5] = new HashJoin(kwaycursor[10], kwaycursor[11], new int[] { 1, 1 }, new int[] { 1 });
//kwaycursor3 = (data2.tid data2.item data4.tid data4.item)

// second join. result and cand4, where cand4.item4 = data4.item and cand4.item2 = data2.item

kwaycursor[6] = new HashJoin(kwaycursor[12], kwaycursor[5], new int[] { 5, 3 }, new int[] { 4, 2 });
//kwaycursor4 = (cand4.itemset, cand4.item1, cand4.item2, cand4.item3, cand4.item4, data2.tid,
data2.item, data4.tid, data4.item)

// third join
kwaycursor[7] = new HashJoin(kwaycursor[13], kwaycursor[6], new int[] { 1, 2 }, new int[] { 8, 4 });
//kwaycursor6 = (data3.tid, data3.item, cand4.itemset, cand4.item1, cand4.item2, cand4.item3,
cand4.item4, data2.tid, data2.item, data4.tid, data4.item)

//fourth join
kwaycursor[8] = new HashJoin(kwaycursor[7], kwaycursor[14], new int[] { 10, 4 }, new int[] { 1, 2 });
```

```

//kwaycursor6 = (data3.tid, data3.item, cand4.itemset, cand4.item1, cand4.item2, cand4.item3,
                cand4.item4, data2.tid, data2.item, data4.tid, data4.item, data1.tid, data1.item)

// now grouping and counting
// first it has to be grouped (c4.itemsetid)

//grouping for cand4.itemset (column3)
final HashGrouper myHashGroup = new HashGrouper(kwaycursor[8], new Function()
    public Object invoke(Object next)
        return ((HashCodeArrayTuple) next).getObject(3);
);

stopwatch[5].stop();

// in the Cursor myHashGroup, you have the Cursors to each grouo with the same itemset
// so it is a cursor of cursors
// debug -- print the S4 Table
int counter = 0;
while (myHashGroup.hasNext())
    counter = 0;
    Cursor tempcursor = (Cursor) myHashGroup.next();
    while (tempcursor.hasNext())
        counter++;
        if (counter == 1)
            if (returnresult.booleanValue())
                System.out.print("ItemsetID: "
                    + ((HashCodeArrayTuple) (tempcursor.next())).getObject(3));
            else
                tempcursor.next();
        else
            tempcursor.next();

    if (returnresult.booleanValue())
        System.out.print(" support: " + counter + "\n");

```

### B.3 SQL Ausführungsplan für Quiver

Der vom SQL Server erzeugte Ausführungsplan für Quiver,  $k = 4$ , Datentabelle = T5.I5.D10k und support = 100

```

print @mymsg + 's4 begin';
INSERT INTO S^4 (itemset, support) SELECT itemset,... (See SQL-Statement)
|--Sequence
|--Index Insert(OBJECT:([dbmine2].[boeckdl].[S^4].[i_S^4_is]),
| SET:([itemset1010]=[S^4].[itemset], [IdxBmk1009]=[Bmk1008]))
|--Sort(ORDER BY:([S^4].[itemset] ASC, [Bmk1008] ASC))
|--Table Spool
|--Clustered Index Insert(OBJECT:([dbmine2].[boeckdl].[S^4].[pk_S^4]),
| SET:([S^4].[support]=[Expr1006], [S^4].[itemset]=[c1].[itemset]))
|--Top(ROWCOUNT est 0)
|--Compute Scalar(DEFINE:([Expr1006]=Convert([Expr1016])))
|--Stream Aggregate(GROUP BY:([c1].[itemset]) DEFINE:([Expr1016]=COUNT_BIG([t1].[TID])))
|--Stream Aggregate(GROUP BY:([c1].[itemset], [t1].[TID]))
|--Nested Loops(Left Anti Semi Join, OUTER REFERENCES:([c1].[itemset], [t1].[TID]))
|--Sort(ORDER BY:([c1].[itemset] ASC, [t1].[TID] ASC))
|--Nested Loops(Inner Join)
|--Index Scan(OBJECT:([dbmine2].[boeckdl].[t5_d10k].[i_t5_d10k_itemtid] AS [t1]))
|--Table Spool
|--Index Scan(OBJECT:([dbmine2].[boeckdl].[C^4].[i_C^4_is] AS [c1]))
|--Row Count Spool
|--Nested Loops(Left Anti Semi Join, OUTER REFERENCES:([c2].[item]))
|--Nested Loops(Left Anti Semi Join, WHERE:([c1].[itemset]<>[c2].[itemset]))
|--Index Scan(OBJECT:([dbmine2].[boeckdl].[C^4].[i_C^4_is] AS [c2]))
|--Row Count Spool
|--Top(1)
|--Index Scan(OBJECT:([dbmine2].[boeckdl].[t5_d10k].[i_t5_d10k_itemtid] AS [t2]))
|--Row Count Spool
|--Top(1)
|--Index Seek(OBJECT:([dbmine2].[boeckdl].[t5_d10k].[i_t5_d10k_itemtid] AS [t2]),
| SEEK:([t2].[Item]=[c2].[item] AND [t2].[TID]=[t1].[TID]) ORDERED FORWARD)
--Index Insert(OBJECT:([dbmine2].[boeckdl].[S^4].[i_S^4_s]),
| SET:([itemset1013]=[S^4].[itemset], [support1012]=[S^4].[support], [IdxBmk1011]=[Bmk1008]))
|--Sort(ORDER BY:([S^4].[support] ASC, [S^4].[itemset] ASC, [Bmk1008] ASC))
|--Table Spool
print @mymsg + 's4 end';
update STATISTICS S^4 WITH FULLSCAN, ALL, NORECOMPUTE;

```

## B.4 XXL Quelltext von Quiver Algorithmus

Abgebildet ist nur der eigentliche Algorithmus, ohne das Laden der Tabellen und der Berechnung der Indizes:

```

stopwatch[4].start();
// quivercursor[3] is the first Nested Loop Join-----
// the following is join A -----
// you get a really really big table
//quivercursor[11].reset();
quivercursor[4] =
new NestedLoopsJoin(quivercursor[10],
                    quivercursor[11],
                    null,
                    hashCodeArrayTuple.FACTORY_METHOD,
                    NestedLoopsJoin.NATURAL_JOIN);
//Format of quivercursor[3]: (TID, Item, ItemsetID)
// sort the table -----
//Comparator: on columns 3 then 1 and ascending=true
// this means sort on itemsetID, TID
final Comparator comparator = Tuples.getTupleComparator(new int[] { 3, 1 }, new boolean[] { true, true });

//Default: 4096;
final int blockSize = 4096;

//Exact: 36; 3*INTEGER=3*12bytes
final int objectSize = 73; //36

//memorysize = objectsize * blocksize
final int memorySize = objectSize * blockSize;

//Default: 4 * blocksize
final int finalMemorySize = 8 * blockSize; //4

// intern sort -----
final Function newQueue = new Function()
    public Object invoke(Object function1, Object function2)
        return new ListQueue(); // Return a new ListQueue (resident in main memory)
    ;
;

//cursor = new MergeSorter(metaDataCursor, comparator, objectSize, memorySize, finalMemorySize);
quivercursor[5] =
    new MergeSorter(quivercursor[4], comparator, objectSize, memorySize, finalMemorySize, newQueue, false);

//now, u have to compute the "big part", for every each tuple from this big table quivercursor[4]-----
// The following is join B -----

//in the while, u add (or dont add) the outerCurrentCandidate

while (quivercursor[5].hasNext())
    // we look for the candidate for some conditions
    // will the next conditions be true, so the outerCurrentCandidate is added to the result,
    // and we make the same procedure for the next candidate
    hashCodeArrayTuple outerCurrentCandidate = (HashCodeArrayTuple) quivercursor[5].next();
    // so first we make a new cursor for each of this tuple
    // so the cursor just contains one tuple!!
    quivercursor[6] = new MySingleObjectCursor(outerCurrentCandidate);

    // the following is join C -----
    // NestedLoopAntiSemiJoin from the candtable (cursor2) and the onetuplecursor(5) with 2.itemset<5.itemset
    // itemset is in column 2 from cursor2
    // and itemset is in column 3 from cursor5
    // equal=false, because we look itemset <itemset and NOT itemset=itemset
    quivercursor[7] =
        new NestedLoopAntiSemiJoin(
            quivercursor[12],
            quivercursor[6],
            new int[] { 2 },
            new int[] { 3 },
            false);

    //the following is join D -----
    //now we take every tuple from cursor6
    //and compute the rest....
    //store the complete result from the "inner" in cursor9

    quivercursor[14] = new MySingleObjectCursor();
    //quivercursor[6].reset();

    //begin of the second while
    while (quivercursor[7].hasNext())
        hashCodeArrayTuple innerCurrentCandidate = (HashCodeArrayTuple) quivercursor[7].next();
        //making a new cursor with this one tuple (singleobjectcursor)
        quivercursor[8] = new MySingleObjectCursor(innerCurrentCandidate);

```

```

//the following is join E -----
Object[] tuple = new Object[2];
Object cursor7item = (innerCurrentCandidate.getObject(1));
Object cursor5tid = (outerCurrentCandidate.getObject(1));
tuple[1] = cursor5tid;
tuple[0] = cursor7item;

//now look in the btree index (dataitemtid) (IndexSeek)
//first we make a hashcodearraytuple with cursor7item, cursor5tid
//and it has the same MetaData as cursor1
HashCodeArrayTuple indexseektuple =
    new HashCodeArrayTuple(tuple, (ResultSetMetaData) quivercursor[3].getMetaData());
//now we look in the btree for this indexseektuple
myBTreeT3.open();

//
// Retrieve the rows of a range query on the B-tree.
// in this case just the single row with the right information from indexseektuple
Cursor treeCursor =
    myBTreeT3.getBTree().query(
        new AbstractIntervallD(
            new MyBtreeKey(indexseektuple),
            new MyBtreeKey(indexseektuple),
            MyBtreeKey.COMPARATOR));

// store the result from the BTree in a MetaDataCursor
quivercursor[9] =
    new MyMetaDataCursor(treeCursor, (ResultSetMetaData) quivercursor[3].getMetaData());

// now, if there is a row in the cursor8, we DONT take the current tuple
// otherwise we take the current tuple to the new cursor9
if (!quivercursor[9].hasNext())
    ((MySingleObjectCursor) quivercursor[14]).put(innerCurrentCandidate);

myBTreeT3.close();

// here we look if the cursor9 has an entry
// if there is one row in it, then we dont take the outerCurrentCandidate to the result
// otherwise the tuple is one of the resultset
// NOTICE: every ItemsetID from one tuple will appear k times in the result, such as in Candidatetable Ck
if (!quivercursor[14].hasNext())
    if (returnresult.booleanValue())
        System.out.print("RESULT: " + outerCurrentCandidate + "\n");

stopwatch[4].stop();

```

## B.5 XXL Quelltext von der Klasse HashJoin

```

import xxl.util.StopWatch;
import xxl.cursors.AbstractCursor;
import xxl.cursors.MetaDataCursor;
import java.sql.SQLException;
import java.sql.ResultSetMetaData;
import xxl.util.WrappingRuntimeException;
import java.util.Vector;
import java.util.HashMap;
import xxl.relationalNEU.AssembledResultSetMetaData;
/**
 * Just a HashJoin
 */
public class HashJoin extends AbstractCursor implements MetaDataCursor

    /** The meta data that will be returned. */
    private ResultSetMetaData metadata;

    /** Vector, containing the result. */
    private Vector result;

    /** index of the result vector. */
    private int index;

    // copy the parameters to global variables, to make a reset possible
    private MetaDataCursor leftTable;
    private MetaDataCursor rightTable;
    private int[] columnLeft;
    private int[] columnRight;

    /**
     * This function gets 2 Cursors, the LeftTable, which is the smaller one and from which
     * the hash table is built and the RightTable, to join with "Hash Inner Join" algorithm.
     * The data in both tables must be a object of HashCodeArrayTuple.
     * The Key for the hashtable will be built from the int[] columnleft objects!

```

```

*
* Example:
*     MetadataCursor test = new HashJoin(table1, table2, new int[] {1, 2}, new int[] {4, 2})
*     it builds a hash table for table1 and makes a join for the conditions:
*     table1.column1 = table2.column4 AND
*     table1.column2 = table2.column2
*
* Precondition:  the cursors of both tables have to be resetable
*
* @param LeftTable the smaller table on which is built the hash table
* @param RightTable the other table to join with
* @param columnleft the columnnumbers of the left table
* @param columnright the columnnumbers of the right table
*/
public HashJoin(MetadataCursor lefttable, MetadataCursor righttable, int[] columnleft, int[] columnright)

    //save the cursor, to make a reset() possible
    this.leftTable = lefttable;
    this.rightTable = righttable;
    this.columnLeft = columnleft;
    this.columnRight = columnright;

    initialize(lefttable, righttable, columnleft, columnright);

/**
 * Initialize
 * Here is the algorithm
 */
public void initialize(MetadataCursor lefttable, MetadataCursor righttable, int[] columnleft, int[] columnright)

    //reset the index of the result vector
    this.index = 0;

    // the joined result tuples
    HashCodeArrayTuple jointuple;

    try
        // initialize the result vector
        result = new Vector();

        int righttable_rows = ((ResultSetMetaData) righttable.getMetaData()).getColumnCount();
        int lefttable_rows = ((ResultSetMetaData) lefttable.getMetaData()).getColumnCount();
        int all_rows = righttable_rows + lefttable_rows;
        HashCodeArrayTuple temphashtuple;

        //-----joining metadata-----
        final String[] columnname = new String[all_rows];
        for (int i = 0; i < lefttable_rows; i++)
            columnname[i] = ((ResultSetMetaData) lefttable.getMetaData()).getColumnName(i + 1);
        for (int i = 0; i < righttable_rows; i++)
            columnname[lefttable_rows + i] = ((ResultSetMetaData) righttable.getMetaData()).getColumnName(i + 1);

        final ResultSetMetaData[] columnmetadata = new ResultSetMetaData[all_rows];
        for (int i = 0; i < lefttable_rows; i++)
            columnmetadata[i] =
                new MyMetaData(
                    ((ResultSetMetaData) lefttable.getMetaData()).getColumnType(i + 1),
                    columnname[i],
                    0,
                    0);
        for (int i = 0; i < righttable_rows; i++)
            columnmetadata[lefttable_rows + i] =
                new MyMetaData(
                    ((ResultSetMetaData) righttable.getMetaData()).getColumnType(i + 1),
                    columnname[lefttable_rows + i],
                    0,
                    0);

        metadata = new AssembledResultSetMetaData(columnmetadata, columnname);

        // ----- build hashtable -----

        // the new Hash Table, which will be built from the input LeftTable
        HashMap leftHashTable = new HashMap();
        // the TID/Items for a hash value
        Vector hashvalue = new Vector();
        // number of columnleft-values.. how many objects are in the key
        int buildkeysize = columnleft.length;

        ResultSetMetaData keymetadata;
        // computing the meta data for the key
        final String[] keymetadacolumnname = new String[columnleft.length];
        for (int i = 0; i < columnleft.length; i++)
            keymetadacolumnname[i] = ((ResultSetMetaData) lefttable.getMetaData()).getColumnName(columnleft[i]);

```

```

final ResultSetMetaData[] keymetadacolumnmetadata = new ResultSetMetaData[columnleft.length];
for (int i = 0; i < columnleft.length; i++)
    keymetadacolumnmetadata[i] =
        new MyMetaData(
            ((ResultSetMetaData) lefttable.getMetaData()).getColumnType(columnleft[i]),
            keymetadacolumnname[i],
            0,
            0);

keymetadata = new AssembledResultSetMetaData(keymetadacolumnmetadata, keymetadacolumnname);

// filling the thashtable with every tuple from the lefttable
Object buildobject[];
while (lefttable.hasNext())
    //computing the pure object for the key (without metadata)
    final HashCodeArrayTuple lefttuple = (HashCodeArrayTuple) lefttable.next();
    // generating the key object
    // the key is the current tuple, just containing the columnobjects columnleft[]
    buildobject = new Object[buildkeysize];
    for (int i = 0; i < buildkeysize; i++)
        buildobject[i] = lefttuple.getObject(columnleft[i]);

    // now compute the buildkey, containing buildobject and its keymetadata
    HashCodeArrayTuple buildkey = new HashCodeArrayTuple(buildobject, keymetadata);

    // now put the data to this key
    //looking if the hash table has a entry at the key columnleft
    if (leftHashTable.containsKey(buildkey))
        //hashvalue.clear();
        hashvalue = (Vector) leftHashTable.get(buildkey);
        hashvalue.add(lefttuple);
    else
        hashvalue = new Vector();
        hashvalue.clear();
        hashvalue.add(lefttuple);

    //put the data into the Hastable: put(key, data)
    //the key is the columnleft parameter, also the join attribute
    //the data is the Vector with format tuple tuple tuple...
    leftHashTable.put(buildkey, hashvalue);

// ----- probe the hash table -----

ResultSetMetaData keymetadata2;
// computing the meta data for the key from the righttable
final String[] keymetadacolumnname2 = new String[columnright.length];
for (int i = 0; i < columnright.length; i++)
    keymetadacolumnname2[i] =
        ((ResultSetMetaData) righttable.getMetaData()).getColumnName(columnright[i]);

final ResultSetMetaData[] keymetadacolumnmetadata2 = new ResultSetMetaData[columnright.length];
for (int i = 0; i < columnright.length; i++)
    keymetadacolumnmetadata2[i] =
        new MyMetaData(
            ((ResultSetMetaData) righttable.getMetaData()).getColumnType(columnright[i]),
            keymetadacolumnname2[i],
            0,
            0);

keymetadata2 = new AssembledResultSetMetaData(keymetadacolumnmetadata2, keymetadacolumnname2);

boolean matching = true;

// number of columnright-values.. how many objects are in the key
int probekeysize = columnright.length;
// the key is the tuple, just containing the columnobjects columnright[]
Object probeobject[] = new Object[probekeysize];

//Probe the LeftHashTable with all rows in RightTable
while (righttable.hasNext())
    final HashCodeArrayTuple righttuple = (HashCodeArrayTuple) righttable.next();
    //build the probekeyobject
    for (int i = 0; i < probekeysize; i++)
        probeobject[i] = righttuple.getObject(columnright[i]);

    // now compute the buildkey, containing buildobject and its keymetadata
    HashCodeArrayTuple probekey = new HashCodeArrayTuple(probeobject, keymetadata2);

    //for each object (hasNext) take key from righttable (probekey) and look in hashtable
    //if the objects matches
    if (leftHashTable.containsKey(probekey))
        //get the Vector from the LeftHashTable.
        //Look for every Tuple if the key matches
        //for every tuple in the vector which matches, make a join
        hashvalue = (Vector) leftHashTable.get(probekey);

```

```

//now check all tuples of the returned vector for the "right" tuples which matches
//these are the ones, which contain the same data as in the probekey

for (int h = 0; h < hashvalue.size(); h++)
    tempashtuple = (HashCodeArrayTuple) hashvalue.get(h);

//now look if this tuple from the vector has the same data as the probekey
//because of hashing, here can be placed also different data
//so we make a "key" from the tempashtuple and look if it is the same as the probekey
matching = true;
for (int i = 0; i < probekeysize; i++)
    if (!(tempashtuple.getObject(columnleft[i]).equals(probekey.getObject(i + 1))))
        matching = false;

if (matching)
    Object temp[] = new Object[all_rows];
    for (int i = 0; i < lefttable_rows; i++)
        temp[i] = tempashtuple.getObject(i + 1);
    for (int j = 0; j < righttable_rows; j++)
        temp[lefttable_rows + j] = righttuple.getObject(j + 1);
    jointuple = new HashCodeArrayTuple(temp, metadata);
    //if (!result.contains(jointuple))
    result.addElement(jointuple);
    index++;

catch (SQLException se)
    throw new WrappingRuntimeException(se);
catch (Exception e)
    System.out.println("Exception: " + e + " (message: " + e.getMessage() + ")");

/**
 * Computes the next element to be returned by the hash division.
 * CONTRACT: This method computes the
 * next element to be returned by a call to <code>next()</code>.
 */
public void computeNext()
    if (--index >= 0)
        setNext(result.get(result.size() - index - 1));

/**
 * Returns the meta data of the hash join.
 *
 * @return ResultSetMetaData object.
 */
public Object getMetaData()

    return metadata;

/**
 * Resets the cursor to its initial state.<br>
 * So the caller is able to traverse the underlying collection again.
 * The modifications, removes and updates concerning the underlying collection,
 * are still persistent. This operation is only successful, if the
 * underlying cursors are resettable.
 */
public void reset() throws UnsupportedOperationException
    leftTable.reset();
    rightTable.reset();
    initialize(leftTable, rightTable, columnLeft, columnRight);

```

## B.6 XXL Quelltext von der Klasse NestetLoopAntiSemiJoin

```

import xxl.cursors.AbstractCursor;
import xxl.cursors.MetaDataCursor;
import java.sql.ResultSetMetaData;
import java.util.Vector;

/**
 * This class implements a Nested Loop Left Anti Semi Join.
 */
public class NestedLoopAntiSemiJoin extends AbstractCursor implements MetaDataCursor

```

```

/** The meta data that will be returned. */
private ResultSetMetaData metadata;

/** Vector, containing the result. */
private Vector result;

/** index of the result vector. */
private int index;

// copy the parameters to global variables, to make a reset possible
private MetaDataCursor leftTable;
private MetaDataCursor rightTable;
private int[] columnLeft;
private int[] columnRight;
boolean equal;

/**
 * Nested Loop Left Anti Semi Join
 * This method gets the two tables, and 2 int arrays, to know the rows which have to be equal.
 * It also gets a boolean, so that it can compare the rows to unequalness.
 *
 * Precondition: the cursors of both tables have to be resetable
 *
 * boolean equal = true -> compare has to be true (standard)
 * boolean equal = false -> compare has to be false
 *
 * @param LeftTable the lefttable
 * @param RightTable the righttable
 * @param columnleft the columnnumbers of the left table with the join attribute
 * @param columnright the columnnumbers of the right table with the join attribute
 * @param equal the algorithm compares the columns to equalness or unequalness
 */
public NestedLoopAntiSemiJoin(
    MetaDataCursor lefttable,
    MetaDataCursor righttable,
    int[] columnleft,
    int[] columnright,
    boolean equal)

    //save the cursor, to make a reset() possible
    this.leftTable = lefttable;
    this.rightTable = righttable;
    this.columnLeft = columnleft;
    this.columnRight = columnright;
    this.equal = equal;
    initialize(lefttable, righttable, columnleft, columnright, equal);

/**
 * initialize method is called from the Constructor and from the reset() method
 * it contains the NestedLoopAntiSemiJoin algorithm
 */
public void initialize(
    MetaDataCursor lefttable,
    MetaDataCursor righttable,
    int[] columnleft,
    int[] columnright,
    boolean equal)

    //reset the index of the result vector
    this.index = 0;

    // the joined result tuples
    HashCodeArrayTuple jointuple;

    try

        // initialize the result vector
        result = new Vector();

        //-----compute metadata-----
        // meta data is the same as the one from the lefttable
        metadata = (ResultSetMetaData) lefttable.getMetaData();

        // -----join-----
        // matching: if there is one match then the tuple will NOT be added to the result,
        // because of ANTI semi join
        boolean matching;
        boolean tempmatching;

        //take every entry from lefttable
        lefttable.reset();
        while (lefttable.hasNext())
            //look for this entry from the lefttable in ALL rows from the righttable
            HashCodeArrayTuple leftTuple = (HashCodeArrayTuple) lefttable.next();
            matching = true;

```

```

        righttable.reset();
        while (righttable.hasNext())
            hashCodeArrayTuple rightTuple = (HashCodeArrayTuple) righttable.next();
            // look if the tuple matches in the specified columns
            tempmatching = true;
            for (int i = 0; i < columnleft.length; i++)
                // tempmatching, if all objects (which have to be checked) from the current righttuple,
                // are the same as them from the lefttuple
                if (!(leftTuple.getObject(columnleft[i])).equals(rightTuple.getObject(columnright[i])))
                    tempmatching = false;

            if (tempmatching)
                matching = false;

            // if equal, then u compare the specified columns to equalness
            if (equal)
                if (matching)
                    result.addElement(leftTuple);
                    index++;

            // else u comare the specified columns to unequalness
            else if (!(matching))
                result.addElement(leftTuple);
                index++;

        catch (Exception e)
            System.out.println("Exception: " + e + " (message: " + e.getMessage() + ")");

/**
 * Resets the cursor to its initial state.<br>
 * So the caller is able to traverse the underlying collection again.
 * The modifications, removes and updates concerning the underlying collection,
 * are still persistent. This operation is only successful, if the
 * underlying cursors are resettable.
 */
public void reset() throws UnsupportedOperationException
    leftTable.reset();
    rightTable.reset();
    initialize(leftTable, rightTable, columnLeft, columnRight, equal);

/**
 * Computes the next element to be returned by the hash division.
 * CONTRACT: This method computes the
 * next element to be returned by a call to <code>next()</code>.
 */
public void computeNext()
    if (--index >= 0)
        setNext(result.get(result.size() - index - 1));

/**
 * Returns the meta data of the join.
 *
 * @return ResultSetMetaData object
 */
public Object getMetaData()

    return metadata;

```

## B.7 SQL - Messergebnisse im Überblick

Tabelle	support	K-Way Join (orig.)	Subquery (orig.)	Subquery (mod.)	SOAP	Quiver	K-Way Join (vert.)
T10.I5.D100k (original)	250	144.853	871.819	906.145	438.703	OOM	missing
	100	217.027	1306.227	2039.641	703.688	OOM	missing
T10.I5.D100k (modified)	250	142898.026	7918.214	1516.507	825.966	OOM	missing
	100	468738.502	12854.537	3105.149	1858.057	OOM	missing
T5.I5.D100k (original)	250	868.361	1070.941	260.170	149.584	OOM	2124.062
	100	6080.271	1565.855	375.022	248.004	OOM	2825.508
T5.I5.D100k (modified)	250	77.974	486.871	473.603	320.777	OOM	467.580
	100	164.637	1096.164	924.169	752.779	OOM	677.545
T5.I5.D10k (original)	100	20.349	134.338	118.233	109.590	146654.674	39.310
	14	62.888	373.456	207.491	203.051	missing	missing
T5.I5.D5k (original)	50	11.160	72.155	64.889	59.235	409888.440	18.074
	14	26.678	165.492	130.648	126.474	missing	missing

## B.8 XXL - Messergebnisse im Überblick

### B.8.1 synthetische Datentabellen

C <sub>4</sub> (rows)	Tabellenname	K-Way Join mit Indizes	K-Way Join ohne Indizes	Set Con. Division	Quiver mit Indizes	Quiver ohne Indizes
128	t5_i5_d10k	12.226	15.303	148.298	50382.070	–
128	t5_i5_d5k	6.353	6.012	67.194	15754.413	–
128	t5_i5_d1k	0.632	1.395	13.300	2977.295	–
128	t5_i5_d500	0.239	0.447	6.990	1437.306	–
128	t5_i5_d50	0.049	0.045	1.983	113.771	136.010
64	t5_i5_d10k	11.668	13.981	70.936	21511.969	–
64	t5_i5_d5k	5.814	6.353	34.718	9920.838	–
64	t5_i5_d1k	0.635	0.976	6.563	1781.648	–
64	t5_i5_d500	0.337	0.459	3.342	844.398	798.876
64	t5_i5_d50	0.049	0.042	0.943	71.665	76.791
32	t5_i5_d10k	11.498	13.332	31.419	6942.693	–
32	t5_i5_d5k	6.264	5.868	18.337	6108.360	–
32	t5_i5_d1k	0.634	1.363	3.348	1505.430	1207.785
32	t5_i5_d500	0.246	0.410	1.648	527.385	471.169
32	t5_i5_d50	0.052	0.067	0.228	44.617	48.026
16	t5_i5_d10k	11.381	13.514	15.745	3511.635	–
16	t5_i5_d5k	5.738	5.828	9.140	3056.062	–
16	t5_i5_d1k	0.590	1.392	1.675	532.899	467.914
16	t5_i5_d500	0.325	0.413	0.812	259.134	221.766
16	t5_i5_d50	0.049	0.069	0.097	21.378	20.663
8	t5_i5_d10k	11.317	13.769	7.840	1835.854	–
8	t5_i5_d5k	6.272	6.192	5.021	1633.515	1521.040
8	t5_i5_d1k	0.988	1.027	0.830	230.342	197.346
8	t5_i5_d500	0.333	0.411	0.413	110.736	93.156
8	t5_i5_d50	0.046	0.069	0.053	6.154	6.371
4	t5_i5_d10k	11.886	14.830	3.942	1086.626	–
4	t5_i5_d5k	6.357	8.258	2.563	684.533	632.000
4	t5_i5_d1k	0.946	1.492	0.449	128.093	107.885
4	t5_i5_d500	0.618	0.665	0.212	61.164	51.749
4	t5_i5_d50	0.243	0.234	0.316	4.563	4.767

**B.8.2 Real life Datentabellen**

C <sub>4</sub> (rows)	Tabellenname	K-Way Join mit Indizes	K-Way Join ohne Indizes	Set Con. Division	Quiver mit Indizes	Quiver ohne Indizes
128	tbmswv2_d10k	10.717	11.698	84.383	10631.287	–
128	tbmswv2_d5k	3.720	4.558	42.019	5140151	–
128	tbmswv2_d1k	0.579	0.796	9.310	1026.990	1252.585
128	tbmswv2_d500	0.315	0.731	5.343	525.036	640.639
128	tbmswv2_d50	0.102	0.079	1.431	88.336	108.873
64	tbmswv2_d10k	10.354	10.431	41.991	5086.141	–
64	tbmswv2_d5k	4.681	4.665	20.745	2499.183	–
64	tbmswv2_d1k	0.570	1.305	4.417	459.249	500.776
64	tbmswv2_d500	0.323	0.423	2.496	251.834	281.943
64	tbmswv2_d50	0.059	0.049	0.497	41.899	47.378
32	tbmswv2_d10k	8.852	11.554	20.757	1893.221	–
32	tbmswv2_d5k	3.501	6.942	10.317	1062.230	1097.314
32	tbmswv2_d1k	0.572	1.140	2.165	181.808	199.134
32	tbmswv2_d500	0.301	0.730	1.170	104.339	111.663
32	tbmswv2_d50	0.058	0.049	0.199	18.560	19.477
16	tbmswv2_d10k	9.881	10.369	10.311	770.447	–
16	tbmswv2_d5k	3.700	4.470	5.131	407.415	–
16	tbmswv2_d1k	0.476	0.753	1.076	77.400	81.828
16	tbmswv2_d500	0.231	0.724	0.574	44.607	46.830
16	tbmswv2_d50	0.054	0.048	0.088	8.070	8.491
8	tbmswv2_d10k	10.257	10.904	5.128	318.704	–
8	tbmswv2_d5k	2.754	4.881	2.560	166.697	174.785
8	tbmswv2_d1k	0.489	0.881	0.543	32.603	34.587
8	tbmswv2_d500	0.232	0.399	0.284	18.929	19.907
8	tbmswv2_d50	0.054	0.048	0.039	6.716	3.776
4	tbmswv2_d10k	10.013	9.063	3.051	86.794	79.092
4	tbmswv2_d5k	4.950	4.796	1.328	36.316	42.695
4	tbmswv2_d1k	0.815	0.767	0.264	8.488	9.404
4	tbmswv2_d500	0.488	0.387	0.263	4.992	5.953
4	tbmswv2_d50	0.273	0.048	0.018	1.592	1.900



## C Abbildungsverzeichnis

### Abbildungsverzeichnis

1	Beispiel - Drei Kassenbons . . . . .	3
2	Transaktionstabelle T10.I5.D100k und T10.I5.D100k_modified . . . . .	7
3	Transaktionstabelle T5.I5.D100k und T5.I5.D100k_modified . . . . .	7
4	kleine Transaktionstabelle T5.I5.D10k . . . . .	8
5	kleine Transaktionstabelle T5.I5.D5k . . . . .	8
6	Cursor in XXL . . . . .	26
7	K-Way Join - Ausführungsplan für Berechnung von $S_4^h$ . . . . .	28
8	Quiver - Ausführungsplan für Berechnung von $S_4^h$ . . . . .	29
9	Division - graf. Ausführungsplan . . . . .	31
10	Messergebnisse SQL T10.I5.D100k support=250 . . . . .	37
11	Charakteristika der $F_k^h$ -und $C_k^h$ -Tabellen bei T10.I5.D100k und support=250 . . . . .	37
12	Messergebnisse SQL T10.I5.D100k support=100 . . . . .	37
13	Charakteristika der $F_k^h$ -und $C_k^h$ -Tabellen bei T10.I5.D100k und support=100 . . . . .	37
14	Messergebnisse SQL T10.I5.D100k_modified support=250 . . . . .	38
15	Charakteristika der $F_k^h$ -und $C_k^h$ -Tabellen bei T10.I5.D100k_modified und support=250 . . . . .	38
16	Messergebnisse SQL T10.I5.D100k_modified support=100 . . . . .	38
17	Charakteristika der $F_k^h$ -und $C_k^h$ -Tabellen bei T10.I5.D100k_modified und support=100 . . . . .	38
18	Ausführungsplan für K-Way Join - T10.I5.D100k und support=250 . . . . .	39
19	Ausführungsplan für K-Way Join - T10.I5.D100k_modified support=250 . . . . .	40
20	Messergebnisse SQL T5.I5.D100k support=250 . . . . .	41
21	Charakteristika der $F_k^h$ -und $C_k^h$ -Tabellen bei T5.I5.D100k und support=250 . . . . .	41
22	Messergebnisse SQL T5.I5.D100k support=100 . . . . .	41
23	Charakteristika der $F_k^h$ -und $C_k^h$ -Tabellen bei T5.I5.D100k und support=100 . . . . .	41
24	Messergebnisse SQL T5.I5.D100k_modified support=250 . . . . .	42
25	Charakteristika der $F_k^h$ -und $C_k^h$ -Tabellen bei T5.I5.D100k_modified und support=250 . . . . .	42
26	Messergebnisse SQL T5.I5.D100k_modified support=100 . . . . .	42
27	Charakteristika der $F_k^h$ -und $C_k^h$ -Tabellen bei T5.I5.D100k_modified und support=100 . . . . .	42
28	Messergebnisse SQL T5.I5.D10k support=100 . . . . .	43
29	Charakteristika der $F_k^h$ -und $C_k^h$ -Tabellen bei T5.I5.D10k und support=100 . . . . .	43
30	Messergebnisse SQL T5.I5.D5k support=50 . . . . .	43
31	Charakteristika der $F_k^h$ -und $C_k^h$ -Tabellen bei T5.I5.D5k und support=50 . . . . .	43
32	horiz. und vert. Kandidatentabellenerstellung . . . . .	44
33	Vergleich Quiver - K-Way Join vert. - K-Way Join orig. . . . .	44
34	Messergebnisse XXL T5.I5.D5k . . . . .	46
35	Messergebnisse XXL T5.I5.D10k . . . . .	46
36	Messergebnisse XXL T5.I5.D1k . . . . .	47
37	Messergebnisse XXL T5.I5.D500 . . . . .	47
38	Messergebnisse XXL T5.I5.D50 . . . . .	47
39	Messergebnisse XXL K-Way Join T5.I5.D10k . . . . .	48
40	Messergebnisse XXL Quiver T5.I5.D500 . . . . .	48
41	Messergebnisse XXL Quiver T5.I5.D50 . . . . .	48
42	Erstellung des Index (Item, TID) . . . . .	49
43	Erstellung des Index (TID) . . . . .	49
44	K-Way Join - XXL - T5.I5.D10k und tbmswv2_D10k . . . . .	49
45	Division - XXL - T5.I5.D10k und tbmswv2_D10k . . . . .	49
46	Quiver - XXL- T5.I5.D10k und tbmswv2_D10k . . . . .	50
47	Messergebnisse XXL Quiver tbmswv2_D1k . . . . .	50

48	Messergebnisse XXL Vergleich K-Way Join - Division - Quiver mit tbmswv2_D10k . . .	50
----	--	----

## D Tabellenverzeichnis

### Tabellenverzeichnis

1	Transaktionstabellen Informationen . . . . .	8
2	horizontaler Tabellenaufbau . . . . .	11
3	vertikaler Tabellenaufbau . . . . .	11
4	verwendete horizontale Tabellen . . . . .	12
5	verwendete vertikale Tabellen . . . . .	12
6	Angelegte Indizes für jede Tabelle . . . . .	13
7	Berechnungsreihenfolge der einzelnen Algorithmen . . . . .	14
8	Division durch SQL Statements ersetzt . . . . .	22
9	Übersicht der Operatoren der Ausführungspläne . . . . .	27
10	Charakteristika der Transaktionstabellen . . . . .	46

## Literaturverzeichnis

### Literatur

- [1] R. Agrawal and R. Srikant. Fast Algorithms for Mining Association Rules. In J. B. Bocca, M. Jarke, and C. Zaniolo, editors, *Proc. 20th Int. Conf. Very Large Data Bases, VLDB*, pages 487–499. Morgan Kaufmann, 12–15 1994.
- [2] J. V. den Bercken, J.-P. Dittrich, and B. Seeger. javax.XXL: A prototype for a Library of Query processing Algorithms. In W. Chen, J. F. Naughton, and P. A. Bernstein, editors, *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA*, page 588. ACM, 2000.
- [3] M. Exner. JavaDoc Einfuehrung, April 2002.
- [4] R. Husser. Entwurf und Integration von Set Containment Division Operatoren in einen Anfrageprozessor. Studienarbeit Nr. 1866, Universität Stuttgart, Fakultät Informatik, 2003.
- [5] K. Rajamani and A. Cox. Efficient Mining for Association Rules with Relational Database Systems. IBM Santa Teresa Labs, August 1999.
- [6] R. Rantzau. Frequent Itemset Discovery with SQL Using Universal Quantification. Computer Science Department, University of Stuttgart, 2002.
- [7] R. Rantzau. Processing Frequent Itemset Discovery Queries by Division and Set Containment Join Operators. Computer Science Department, University of Stuttgart, 2003.
- [8] R. Rantzau, L. D. Shapiro, B. Mitschang, and Q. Wang. Algorithms and Applications for Universal Quantification in Relational Databases. Computer Science Department, University of Stuttgart and Portland State University, 2001.
- [9] S. Sarawagi, S. Thomas, and R. Agrawal. Integrating association rule mining with relational database systems: alternatives and implications. 1998.
- [10] S. Thomas and S. Chakravarthy. Performance Evaluation and Optimization of Join Queries for Association Rule Mining. Database Systems Research and Development Center, Computer and Information Science and Engineering Department, University of Florida, Gainesville FL 32611, 1999.
- [11] Z. Zheng, R. Kohavi, and L. Mason. Real World Performance of Association Rule Algorithms. In F. Provost and R. Srikant, editors, *Proceedings of the Seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 401–406, 2001. This is the long version of the original paper.

## **Erklärung**

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

---

(Daniel Böck)

