QUERY PROCESSING CONCEPTS AND TECHNIQUES FOR SET CONTAINMENT TESTS

Von der Fakultät Informatik, Elektrotechnik und Informationstechnik
der Universität Stuttgart
zur Erlangung der Würde eines
Doktors der Naturwissenschaften (Dr. rer. nat.)
genehmigte Abhandlung

Vorgelegt von

RALF RANTZAU

aus Detmold

Hauptberichter: Univ.-Prof. Dr.-Ing. habil. Bernhard Mitschang
Mitberichter: Univ.-Prof. Rudolf Bayer, Ph. D.

Tag der mündlichen Prüfung: 22. Dezember 2003

Institut für Parallele und Verteilte Systeme (IPVS)
der Universität Stuttgart,
Bundesrepublik Deutschland

2003

"[…] the division operator does not have the same importance as the other operators—it is not needed as often, and database systems do not try to exploit the semantics of division by implementing it as a distinct operator (as, for example, is done with the join operator)."

R. Ramakrishnan, J. Gehrke [RG00]

Preface

The above citation is a good commentary on the division operator. This work investigates the division operator and related operators that all help to solve the set containment test problem. We do not try to show that division is as *important* as the other operators of the relational algebra, but we hope that we are able to demonstrate that it is nevertheless a truly *useful* operator.

Acknowledgements

I would like to thank several people who contributed to this work in special ways.

First, I thank Prof. Bernhard Mitschang, who encouraged me to work on this fascinating and somewhat exotic topic in his research group, for his continuous support over five years, bright ideas, and wise suggestions.

The analysis of my work by Prof. Rudolf Bayer and his valuable comments are greatly acknowledged. I warmly thank him for being a co-reviewer of my thesis.

I am grateful to Prof. Leonard Shapiro for sharing some of his insights with me, being a great host during my stays in Portland, and for many unforgettable discussions on query processing and optimization.

Clara Nippl's work on the "stream-join operator" has inspired me to work on relational division.

I thank my colleagues at the University of Stuttgart for fruitfull discussions and a lot of fun: Carmen Constantinescu, Clemens Dorda, Matthias Großmann, Uwe Heinkel, Nicola Hönle, Mihály Jakob, Kim Hyon Hee, Tobias Kraft, Lu Jing, Christoph Mangold, Marcello Mariucci, Daniela Nicklas, Viera Rewucki, Annemarie Roesler, Thomas Schwarz, Frank Wagner, Ralf Wagner, Mirka Zimmermann, as well as my former colleagues Aiko Frank, Michael Jaedicke, Peter Kutschera, Jochen Rütschlin, Kerstin Schneider, and Jürgen Sellentin.

I especially thank my colleague Holger Schwarz for numerous good pieces of advice, proofreading drafts of this work, and a great cooperation during six years.

Thank you, Dirk, Lisa, Marina, Simone, and Tim for supporting me always.

This work is dedicated to Karin and Otfrid.

Ralf Rantzau

Ludwigsburg, December 24, 2003

Abstract

Relational division is an operator of the relational algebra that realizes universal quantifications in queries against a relational database. Expressing a universal quantification problem in SQL is cumbersome. If the division operator would have a counterpart in a query language, a more intuitive formulation of universal quantification problems would be possible. Although division is a derived operator—it can be expressed using other basic algebra operators—the performance of queries involving a division problem can be significantly increased if it is implemented as a separate operator in a query processor.

In this work, we study relational division as well as operators related to it like *set containment join*. We present a classification of division algorithms and define algebraic laws to enable query optimization for queries involving division. A natural generalization of the division operator, called *set containment division*, is proposed and algorithms that realize it are discussed, including an algorithm that uses a new in-memory index data structure. Furthermore, we study strategies for parallelizing set containment division algorithms.

The most popular data mining sub-task called *frequent itemset discovery*, where one tries to find the number of transactions that contain a given set of items, is an excellent example of a problem that requires efficient set containment tests. In this work, we focus on frequent itemset discovery algorithms that exploit the query processing capabilities of a relational database system. After reviewing the best algorithms based on SQL-92, we suggest a new approach, called *Quiver*, that can be executed with the help of set containment division.

We report on performance results that cover experiments with frequent itemset discovery algorithms running on commercial database systems as well as experiments that highlight the most important properties of set containment test algorithms using a query processor prototype implemented in Java.

Zusammenfassung[1]

Die Division ist ein Operator der relationalen Algebra zur Realisierung der All-Quantifizierung innerhalb einer Anfrage an ein relationales Datenbanksystem. Die Division ist ein abgeleiteter Operator, wie zum Beispiel auch der Verbundoperator (Join), das heißt eine Division kann mit Hilfe anderer Operatoren der Algebra ausgedrückt werden. Da in der Anfragesprache SQL kein adäquates Sprachkonstrukt für die relationale Division existiert, ist es nicht möglich, eine SQL-Anfrage, die ein All-Quantifizierungsproblem enthält, auf intuitive Weise zu formulieren. Für den ebenfalls abgeleiteten Verbundoperator hingegen gibt es adäquate Sprachkonstrukte in SQL. In der Literatur wurden Divisionsalgorithmen vorgestellt, die deutlich effizienter sind als Realisierungen des Operators mit Hilfe von Basisoperatoren.

Diese Arbeit befasst sich sowohl mit der relationalen Division als auch mit verwandten Operatoren, wie beispielsweise dem Set Containment Join. Dabei steht die Frage im Zentrum, durch welche effizienten Algorithmen die logischen Operatoren realisiert werden können. Dazu gehören auch Überlegungen zur Integration der Operatoren in ein Datenbanksystem. So benötigt ein Datenbanksystem Transformationsregeln für Ausführungspläne während der Anfrageoptimierung und Strategien zur Parallelisierung der Pläne. Diese und weitere Fragen werden in dieser Dissertation erörtert, die in die nachfolgend skizzierten drei Teilbereiche gegliedert ist.

Erstens, schlage ich eine Klassifikation der Eingabedatencharakteristika für die Division vor und ordne jeder Klasse Algorithmen zu, die die jeweiligen Datencharakteristika für eine effiziente Verarbeitung voraussetzen. Darüber hinaus werden mehrere algebraische Gesetze (Transformationsregeln) formuliert, die einen algebraischen Ausdruck mit Divisionsoperator in einen äquivalenten Ausdruck überführen. Die Klassifikation der Eingabedaten mit den zugehörigen in Frage kommenden Divisionsrealisierungen ist zusammen mit den algebraischen Gesetzen die Voraussetzung dafür, dass ein Anfrageoptimierer einen effizienten Ausführungsplan einer Anfrage mit Division finden kann.

Zweitens, stelle ich einen erweiterten Divisionsoperators vor, Set-Containment-Division-Operator genannt, der nicht auf einem einzigen Divisor, sondern auf mehreren Gruppen von Divisoren operiert. Der Operator realisiert dabei eine Vereinigung von mehreren Divisionsausführungen. Wir zeigen die Äquivalenz dieses Operators mit anderen Vorschlägen aus der Literatur und stellen die Ähnlichkeit zum Set-Containment-Join-Operator heraus, dessen Verbundbedingung auf mengenwertigen Attributen basiert. Für all diese Operatoren werden verschiedene Algorithmen untersucht, es werden neue Ansätze für die Set Containment Division vorgeschlagen und Strategien zur parallelen Ausführung des Operators diskutiert. Zu den neuen Ansätzen gehört der Algorithmus Subset Index Set Containment Division, der auf einer Datenstruktur beruht, die die Ober- und Untermengenbeziehung zwischen den Mengen in einer Relation effizient verwaltet.

Drittens, untersuche ich als Einsatzgebiet für die relationale Division und ähnliche Operatoren die klassische Data-Mining-Technik Frequent Itemset Discovery, bei der häufig auftre-

[1] A summary of the dissertation in German

tende Kombinationen von Elementen (Frequent Itemsets) in einer großen Anzahl von Mengen (Transaktionen) gesucht wird. Insbesondere wird überprüft, welche Transaktionen eine Obermenge von bestimmten Itemsets sind. Dieses Problem wird zum Beispiel idealerweise durch den Set-Containment-Division-Operator gelöst. Wir stellen einen neuen Frequent-Itemset-Discovery-Algorithmus genannt Quiver vor, der von einem solchen Operator profitiert. Die Besonderheit des Algorithmus liegt darin, dass sowohl die Transaktionen als auch die Itemsets in Relationen verwaltet werden, bei denen ein Tupel ein Menge-Element-Paar repräsentiert. Üblicherweise werden in allen anderen Ansätzen, die auf SQL-92 beruhen, vollständige Itemsets durch ein einziges Tupel repräsentiert. Neben diesem neuen Algorithmus werden SQL-basierte Algorithmen analyisert und das Leistungsverhalten der Algorithmen sowohl an Hand von Messungen mit kommerziellen Datenbanksystemen als auch mit einer eigenen Implementierung von Anfrageausführungsplänen aufgezeigt.

Nachfolgend gehe ich auf jeden der eben erwähnten drei Teilbereiche dieser Arbeit detaillierter ein und geben einen kurzen Ausblick auf offene Fragen.

Relationale Division

Die Division ($\div$) ist ein binärer Operator der relationalen Algebra, der aus einer Dividendrelation r_1 und Divisorrelation r_2 eine Quotientrelation r_3 erzeugt: $r_1 \div r_2 = r_3$. Die zugehörigen Relationenschemas sind dabei $R_1(A \cup B)$, $R_2(B)$ sowie $R_3(A)$, wobei A und B nichtleere, disjunkte Attributmengen sind.

In der Literatur finden sich viele verschiedene, äquivalente Definitionen des logischen Divisionsoperators. Die ursprüngliche Definition geht auf Codd zurück [Cod70]. Sie macht schon deutlich, dass es bei der Division um "Untermengentests" oder "set containment tests" geht: $r_1 \div r_2 = \{t \mid t = t_1.A \wedge t_1 \in r_1 \wedge r_2 \subseteq i_{r_1}(t)\}$, wobei $i_{r_1}(x) = \{y \mid (x,y) \in r_1\}$ die Bildmenge von x unter r_1 ist.

Graefe [GC95] hat gezeigt, dass für die Division Algorithmen existieren, die im Allgemeinen eine bessere Leistung zeigen als Realisierungen des Operators mit Hilfe anderer Operatoren. Wir haben alle bekannten Algorithmen analysiert und danach klassifiziert, wie und nach welchen Attributen die Eingabedaten gruppiert sind. Darüber hinaus wurden neue Algorithmen vorgeschlagen und bekannte erweitert, um vorhandene Lücken in der Daten-Algorithmen-Klassifikation zu füllen. Zum Beispiel wurde zusammen mit Nippl [NRM00] ein Divisionsoperator namens *StreamJoin Division* vorgestellt, der besonders hauptspeichereffizient ist und eine Dividendtabelle r_1 als Eingabe benötigt, die nach den Attributwerten von B gruppiert ist.

Wenn ein Divisionsoperator in einem Datenbanksystem realisiert wird, sollten für den Anfrageoptimierer Regeln bereitgestellt werden, um Ausführungspläne mit einem enthaltenen Divisionsoperator transformieren zu können. Dies ist eine Voraussetzung dafür, einen guten oder sogar optimalen Plan zu finden. Als Grundlage hierfür stelle ich mehrere algebraische Gesetze vor, die einen algebraischen Ausdruck mit einem Divisionsoperator in einen äquivalenten Ausdruck umwandeln, der ebenfalls einen oder mehrere Divisionsoperatoren enthalten kann. Die algebraischen Gesetze decken Situationen ab, in denen einer der Operatoren Vereinigung, Durchschnitt, Differenz, Selektion, Kartesisches Produkt, Verbund oder Gruppierung zusammen mit

einem Divisionsoperator auftritt.

Eine Erweiterung des Divisionsoperators

Der Divisionsoperator prüft, welche Gruppen im Dividend alle Tupel des Divisors enthalten. Betrachtet man die Dividendengruppen und den Divisor als Mengen, kann man die Division als einen Untermengentest mit der Kardinalität $N : 1$ ansehen, wobei man N Dividendenmengen einer Divisormenge gegenüberstellt. Wenn man die Mengenelemente statt in Gruppen als mengenwertige Attribute in den Relationen verwaltet, entspricht die Division einem Verbundoperator, bei dem die Bedingung die Untermengenrelation ist: $r_1 \bowtie_{b_1 \supseteq b_2} r_2$. Wenn man nun statt dem einzelnen Divisortupel mehrere Tupel zulässt, das heißt, r_1 besitzt Scheman $R_1(A \cup \{b_1\})$ und r_2 hat Schema $R_2(\{b_2\} \cup C)$, dann ergibt sich ein Operator, der als *Set Containment Join* bezeichnet wird. Die Attributmengen A und C repräsentieren Mengenidentifikatoren und die Attribut b_1 und b_2 sind die zu vergleichenden Mengen. In der Literatur wurden verschiedene Algorithmen zur Realisierung des Set Containment Join vorgeschlagen.

Wenn man beim Set Containment Join die Mengenelemente wieder durch Gruppen repräsentiert und beim Ergebnis die Verbundattribute b_1 und b_2 weglässt, ergibt sich ein Operator, den ich als *Set Containment Division* ($\div^*$) bezeichne. Man kann ihn durch folgenden Ausdruck definieren:

$$r_1 \div^* r_2 = \bigcup_{t \in \pi_C(r_2)} (r_1 \div \pi_B(\sigma_{C=t}(r_2))) \times (t)\,.$$

Es zeigte sich, dass in der Literatur zwei äquivalente Operatoren unter den Namen Great Divide [DD92] und Generalized Division [Dem82] vorgeschlagen wurden. Die Äquivalenz dieser Definitionen wird in dieser Arbeit gezeigt. Im Unterschied zu diesen Vorschlägen, stelle ich zum ersten Mal drei Algorithmen für den Operator vor und diskutiere, wie die Ausführung des Operators parallelisiert werden kann.

Eine der Implementierungen von Set Containment Division basiert auf dem Divisionsoperator Merge-Sort Division, eine weitere auf Hash-Division. Ein dritter Ansatz beruht auf der Idee, eine Datenstruktur Subset Graph, die einem Hasse-Diagramm ähnlich ist, für die Gruppenelemente des Dividenden oder des Divisors zu verwenden. Hierbei werden Gruppen/Mengen durch Knoten repräsentiert. Eine gerichtete Kante des Graphen existiert zwischen den Knoten v_1 und v_2 genau dann, wenn die Menge in v_1 eine echte Untermenge der Menge in v_2 ist und keine Menge in der Relation existiert, die gleichzeitig eine echte Obermenge von v_1 und eine echte Untermenge von v_2 ist. Es werden also nur *direkte* Untermengenbeziehungen durch Kanten repräsentiert. Als eine Variation dieser Datenstruktur betrachte ich den Fall, dass die Elemente, die eine Unter- und Obermenge gemeinsam haben, nur in der Untermenge gespeichert wird. Diese Variante nenne ich Compressed Subset Graph. In Leistungsmessungen ließ sich zeigen, dass zwar der Speicherbedarf durch Kompression stark verringert werden kann, aber dies zu hohen Laufzeiteinbußen führt.

Lösung eines Data-Mining-Problems mit Hilfe der Division

Frequent Itemset Discovery ist eine wichtige Data-Mining-Methode, die zum Beispiel ein Bestandteil der Suche nach Assoziationsregeln ist. Es gibt eine Vielzahl von Algorithmen für Frequent Itemset Discovery. Darunter gibt es SQL-basierte Algorithmen, bei denen ein Datenbanksystem für die Speicherung der Transaktionen und der Itemsets verwendet wird und die gesamte Anwendungslogik durch mehr oder weniger komplexe Anfragen ausgedrückt wird. In Untersuchungen in der Literatur wurde gezeigt, dass SQL-basierte Algorithmen für dieses Problem eine schwächere Leistung aufweisen als Algorithmen, die mit proprietären Datenstrukturen in Dateien und im Hauptspeicher operieren. Wenn man jedoch mit sehr großen Datenmengen nach Frequent Itemsets sucht, ist der Einsatz eines Datenbanksystems von Vorteil, weil die Ausführung des Algorithmus meist besser skaliert als mit einem dediziertem Data-Mining-System. Der Grund hierfür liegt darin, dass proprietäre Data-Mining-Systeme nicht dafür vorgesehen sind, auf Daten zu operieren, die die Größe des physischen Hauptspeichers überschreiten. Anders ist dies bei SQL-basierten Algorithmen, sofern der Anfrageoptimierer skalierbare Anfragepläne generieren kann.

Da unser Ansatz keine besondere Funktionalität der Anfragesprache SQL benötigt, beschränke ich mich bei der Analyse von SQL-basierten Algorithmen auf solche, die den Sprachumfang von SQL-92 abdecken, das heißt ich schließe objekt-relationale Erweiterungen wie benutzerdefinierte Funktionen und sehr große Objekte (binary large objects) aus. Allen diesen Algorithmen ist gemeinsam, dass die Itemsets in einem Tabellenschema gespeichert werden, das für jedes Item ein eigenes Attribut vorsieht. Die Itemsets werden durch das Schema $(s\#, i\#_1, \ldots, i\#_k)$ repräsentiert, wobei $s\#$ ein Itemset bezeichnet und $i\#_1, \ldots, i\#_k$ die Items (meist in lexikographisch aufsteigender Reihenfolge der Item-Werte). Eine andere Möglichkeit besteht darin, jedes Item eines Itemsets in einem eigenen Tupel zu halten, das heißt ein Schema $(s\#, i\#)$ zu verwenden. Dies ist das gleiche Schema, das für Transaktionen benutzt wird. Bei einem solchen *vertikalen* Schema für Transaktionen *und* Itemsets ist es möglich, den Set-Containment-Division-Operator einzusetzen: Man dividiert die Transaktionstabelle $t(t\#, i\#)$ durch die Kandidaten-Itemset-Tabelle $c(s\#, i\#)$: $t \div^* c$. Der Set-Containment-Division-Operator liefert als Ergebnis diejenigen Kombinationen von Transaktions- und Itemset-Identifikatoren $(t\#, s\#)$, bei denen das Itemset in der Transaktion enthalten ist. Wenn man das Ergebnis nach den Transaktionsidenfikatoren $t\#$ gruppiert und pro Gruppe die Zeilen summiert, findet man die Itemset-Identifikatoren, deren Items häufig genug auftreten, das heißt die Frequent Itemsets.

Wir schlagen einen neuen Algorithmus *Quiver* vor und beschreiben ihn mit Hilfe des relationalen Tupelkalküls, der relationalen Algebra sowie mit SQL. Die Ausdrücke für Quiver in der relationalen Algebra verwenden die Set Containment Division sowohl für die Kandidatengenerierungsphase als auch für die Support-Testphase. In Leistungsmessungen konnte gezeigt werden, dass der Quiver-Algorithmus wie erwartet schlechte Ausführungszeiten auf kommerziellen Datenbanksystemen liefert, dass aber eine Realisierung der Support-Testphase in einem selbstimplementierten Anfrageprozessor im Vergleich mit anderen Algorithmen höchstens um eine Größenordnung schlechtere Ausführungszeiten zeigte und in einigen Fällen sogar am besten abschnitt.

Ausblick

Es ist interessant zu untersuchen, in wie weit die vorgestellten Algorithmen auch für die Verarbeitung von XQuery-Anfragen eingesetzt werden können, die eine sogenannte Quantified Expression mit dem Schlüsselwort *every* enthalten, womit eine Allquantifizierung ermöglicht wird. Solche Ausdrücke können für die Formulierung einer Division benutzt werden. Insbesondere ist für die Semantik einiger XQuery-Anfragen wichtig, dass die Tupelreihenfolge, wie sie in den Eingabedokumenten vorliegt, auch im Anfrageergebnis beibehalten wird. Deshalb sind Überlegungen notwendig, wie diese Ordnungserhaltung bei quantifizierten Ausdrücken in XQuery garantiert werden kann und trotzdem eine effiziente Ausführung der Allquantifizierung erreicht wird.

Ein weiterer Bereich aktueller Forschung sind die sogenannten Continuous Queries und Data Streams. Wie für andere Operatoren stellt sich hier das Problem, wie man ressourcenschonend eine große Anzahl von Anfragen ausführt und optimiert, wenn mehr oder weniger kontinuierlich neue Daten erzeugt werden. Beispielsweise kann man sich ein Szenario vorstellen, bei dem viele Anfragen einen Set-Containment-Division-Operator enthalten, der jeweils eine andere, konstante Dividendtabelle als Eingabe besitzt aber dieselbe Tabelle, die kontinuierlich weitere Gruppen erzeugt, als Divisor dient. Wenn eine neue Divisorgruppe erzeugt wird, stellt sich das Problem, wie man die Division in allen Anfragen ausführt, ohne alle Dividendtabellen erneut vollständig lesen zu müssen.

Contents

"A bad beginning makes a bad ending."
Euripides (484 BC – 406 BC)

1 Introduction

This chapter highlights the key problem that we study in this thesis and outlines the structure of the document.

1.1 Problem Statement

The problem of testing if all elements in a set A are contained in another set B is a basic problem in data management. Suppose, the sets are represented as relations, where each element represents a tuple, the problem can be solved in two steps: first, join the two sets with an equality predicate on all columns of both relations and second, check if all elements of A have a join partner in B. The second step can be accomplished, e.g., by comparing the number of tuples in A with the number of tuples in the join result or by a set difference operation: if A minus the join result is an empty relation then $A \subseteq B$.

Generally speaking, the set containment test problem can be solved by a data management system using efficient implementations of relational operators and by exploiting access paths defined on the input data. However, what if we want to test many sets A against many sets B? A straightforward approach is to repeat the test in a nested-loop fashion, i.e., for each set A, test for each set B if $A \subseteq B$.

The set containment test is a special case of a universal quantification in predicate calculus where we check whether a predicate holds for all elements of a set. In our case, we test whether all elements $a \in A$ fulfill the set membership predicate $a \in B$. This special case of universal quantification has a counterpart in the relational algebra: the division operator.

In this thesis, we study this problem and applications of it and discuss query processing

strategies to solve the problem efficiently using database technology. We will show that current commercial database management systems do not always provide an efficient solution and therefore, we suggest novel query processing approaches for this problem.

1.2 Overview

The remainder of the document is organized as follows. In Chapter 2, we analyze the problem of universal quantification in the relational model. This comprises an overview of definitions for the relational division operator as well as algebraic laws that can be exploited by a relational query optimizer when rewriting an algebraic representation of a query involving a division operation. Chapter 3 discusses known division algorithms and presents several new approaches that improve the efficiency for special cases of input data. In Chapter 4, we review set containment *join* algorithms and discuss, how to realize the set containment *division* operator in a query execution engine. We discuss the important data mining problem of frequent itemset discovery as well as several SQL-based algorithms for it, including a new one that can exploit the division operator, in Chapter 5. In Chapter 6, we discuss performance experiments for the algorithms discussed in the previous chapters. This includes both an analysis of SQL-based algorithms using commercial database systems and stand-alone Java implementations of query execution strategies. We give a summary of the thesis as well as our conclusions in Chapter 7 and suggest directions for future work. Finally, Appendix A provides proofs of theorems and algebraic laws, Appendix B provides source code samples of the SQL-based algorithms as well as of the Java implementations of query execution plans, and Appendix C gives detailed information on some of the datasets used for performance experiments.

"It is by universal misunderstanding that all agree. For if, by ill luck, people understood each other, they would never agree."

C. Baudelaire (1821 – 1867)

2

Universal Quantification

This chapter introduces the terminology used in this document, it provides definitions of the division operator and related operators, highlights several algebraic laws for division, discusses how division can be expressed in SQL, and mentions applications that benefit from the division operator.

2.1 Introduction

Universal quantification is an important operation in the first order predicate calculus. This calculus provides existential and universal quantifiers, represented by $\exists$ and $\forall$, respectively. A universal quantifier that is applied to a variable x of a formula f specifies that the formula is true for all values of x. We say that x is *universally quantified* in formula f, and we write $\forall x : f(x)$ in calculus.

In relational database management systems, universal quantification is implemented by the division operator, represented by $\div$ in the relational algebra, which was introduced by Codd in 1970 [Cod70]. The division operator is important for databases because it appears often in practice, particularly in business intelligence applications, including online analytic processing (OLAP) and data mining. In this thesis, we will focus primarily on the division operator and variations of it but we will also discuss its relationship to the set containment join operator, which requires set-valued attributes as input.

Universal quantification is considered a useful functionality for database query languages. Darwen and Date suggest in "The Third Manifesto" [DD95] that any database language based on the relational model shall allow to easily express universal quantification: "If [...] [a database

language] includes a specific operator for relational projection, then it should also include a specific operator for the general form of relational division [...]."

Besides the relational world, universal quantification plays a role, e.g., in *XQuery*, the de-facto standard database query language for XML data. It provides a syntax for universally *quantified expressions* [W3C03].

The remainder of this chapter is organized as follows. First, we briefly summarize the terms used in this document regarding both the theoretical and practical view on relational databases: the relational model and the implementation of the model in a relational database management system. In Section 2.3, we present and compare several definitions of the relational division operator from the literature. In addition, we introduce an operator that generalizes division and discuss its relationship to other operators. We provide important algebraic laws involving division in Section 2.4. Section 2.5 focuses on the representation of relational division in SQL. Finally, in Section 2.6 we discuss applications of universal quantification.

2.2 Terminology

In this work, we separate the terminology used for database systems from that used for the relational model that underlies database systems. We do this to emphasize the importance of being unambiguous when speaking of a relational operator. In the relational algebra, i.e., in theory, an operator is defined in terms of its logical behavior based on a mathematical expression that characterizes the result relation. In the world of software systems, i.e., in practice, an operator embodies a particular algorithm. Such an algorithm may expose more properties than its mere logical behavior regarding the contents of the input and output relations. These properties are important when we discuss efficient implementations of the division operator in Chapter 3.

2.2.1 Relational Model

When we speak of the relational model, we consider a

- *database* as a collection of relations. A
- *relation* contains data that comply to a
- *relation schema*, which is defined by a list of
- *attributes* that represent values of a domain. A
- *domain* is a set of allowed values for an attribute. A relation is a
- *set* of
- *tuples*, each of which is an element of the Cartesian product of the attribute domains.

  The relational model provides languages like the relational algebra or the tuple relational calculus to formulate queries on a database. In the relational algebra, a query is formulated as an

- *algebra expression*, which is composed of
- *logical operators* that take one or more relations as input and produce one output relation. There are several
- *algebraic laws* that define how to transform one expression into an equivalent expression. Two expressions are equivalent if they describe the same set of output tuples for any contents of the input relations. These laws are used by a relational system to optimize the execution strategy for a query.

2.2.2 Relational System

When we speak of the software that realizes the relational model, called *relational database management system (RDBMS)*, we consider a

- *database* as a collection of tables. A
- *table* contains data that comply to a
- *table schema*, which is defined by a list of
- *columns* that represent values of a data type. A
- *data type* is a set of allowed values for a column. A table is a
- *multi-set* (or *bag*) of
- *rows*, each of which is an element of the Cartesian product of the column data types.

  An RDBMS provides a language like SQL to formulate queries on a database. The system answers a query by first transforming it into to a
- *query execution plan*, and then by executing the plan by an RDBMS component called
- *query execution engine*. A query execution plan is composed of
- *physical operators*, each of which is defined by an implementation of an algorithm. For each logical operator in the relational model, there can be one or more physical operators that all produce the same result as the logical operator but, in addition, a physical operator may exploit certain physical data characteristics of the input tables or produce certain data characteristics in the output table. Not every physical operator, like, e.g., the *sort* operator, must have a counterpart in the algebra of the relational model.

Table 2.1 summarizes analogous terms used for the relational model and a relational system.

Relational Model	RDBMS
database	database
relation	table
relation schema	table schema
attribute	column
domain	data type
set	multi-set
tuple	row
logical operator	physical operator
algebraic expression	query execution plan

Table 2.1: Terms used in theory and practice of the relational world

	Division	Set containment division/ great divide/ generalized division	Set containment join
Notation	$r_1 \div r_2$	$r_1 \div^* r_2$	$r_1 \bowtie_{b_1 \supseteq b_2} r_2$
Left input / dividend	$R_1(A \cup B)$, many groups	$R_1(A \cup B)$, many groups	$R_1(A \cup \{b_1\})$, many sets
Right input / divisor	$R_2(B)$, single group	$R_2(B \cup C)$, many groups	$R_2(\{b_2\} \cup C)$, many sets
Output / quotient	$R_3(A)$	$R_3(A \cup C)$	$R_3(A \cup \{b_1, b_2\} \cup C)$
Data layout	1NF	1NF	non-1NF

Table 2.2: Summary of operator characteristics

2.3 Relational Operators for Universal Quantification

This section discusses four operators used to realize universal quantification in the relational model: relational division, set containment join, set containment division, great divide, and generalized division. We will see that the latter three operators are equivalent. They have been proposed independently in the literature.

For ease of presentation, we give already here in Table 2.2 an overview of the operators to be discussed.

2.3.1 The Small Divide

In this section, we present several alternative definitions for division using tuple relational calculus and relational algebra. We present the original ideas described in the literature but we adapt the names of relations and operators for the sake of a uniform and hence more comprehensible presentation. Refer to Table 2.3 for details on the relational operators used in the following expressions.

Let $A = \{a_1, \ldots, a_m\}$ and $B = \{b_1, \ldots, b_n\}$ be nonempty disjoint sets of attributes. Let $R_1(A \cup$

Operator	Name	Description
$\cup$	Set union	$r_1 \cup r_2 = \{t \mid t \in r_1 \vee t \in r_2\}$
$\cap$	Set intersection	$r_1 \cap r_2 = \{t \mid t \in r_1 \wedge t \in r_2\}$
$-$	Set difference	$r_1 - r_2 = \{t \mid t \in r_1 \wedge t \notin r_2\}$
$\times$	Cartesian product	$r_1 \times r_2 = \{t_1 \circ t_2 \mid t_1 \in r_1 \wedge t_2 \in r_2\}$, where $\circ$ is the concatenation operator.
π_A	Projection	$\pi_A(r) = \{t.A \mid t \in r\}$, where A is a list of attributes, $\{A\}$ is the *set* of attributes in the *list* A, and $t.A$ is the concatenation of values from tuple t that appear in A.
π_E	Extended projection	Allows assigning expressions to attribute names, where E is a list of assignments. Example: $\pi_{a,b+c\rightarrow x}(r)$ for some relation r with schema $R(a,b,c)$ [GMUW02].
σ_C	Selection	$\sigma_C(r) = \{t \mid t \in r \wedge C(t)\}$, where C is a condition.
$\bowtie_C$	Theta-join	$r_1 \bowtie_C r_2 = \sigma_C(r_1 \times r_2)$ and C is a condition.
$\bowtie$	Natural join	$r_1 \bowtie r_2 = \pi_A(\sigma_C(r_1 \times r_2))$, where A is the set of attributes in the schema $R_1(r_1) \cup R_2(r_2)$, $C = \bigwedge_{i=1}^{n} r_1.a_i = r_2.a_i$, and $\{a_1, \ldots, a_n\}$ is the set of attributes appearing in the schema $R_1 \cap R_2$.
$\ltimes_C$	Left semi-join	$r_1 \ltimes_C r_2 = \pi_{[r_1]}(r_1 \bowtie_C r_2)$, C is a condition, $[r_1]$ denotes the attributes of $R_1(r_1)$.
$\overline{\ltimes}_C$	Left anti-semi-join	$r_1 \overline{\ltimes}_C r_2 = r_1 - (r_1 \ltimes_C r_2)$ and C is a condition.
$\supset\!\!\bowtie$	Left outer join	$r_1 \supset\!\!\bowtie r_2 = (r_1 \bowtie r_2) \cup ((r_1 \overline{\ltimes} r_2) \times (\times_1^n(NULL)))$, where n is the number of attributes in schema $R_2(r_2)$ [GK98].
$\rho_{r(A)}$	Rename	r is a relation name and A is a list of attributes; allows reassigning relation and attribute names. Example: $\rho_{r_2(x,y,z)}(r_1)$ for some relation r_1 with schema $R_1(a,b,c)$ [GMUW02].
${}_G\gamma_F$	Grouping	G is a list of grouping attributes and F is a list of aggregation functions applied to some attribute values. Example: ${}_{a,d}\gamma_{\text{sum}(b)\rightarrow total}(r_1)$ for some relation r_1 with schema $R_1(a,b,c,d)$ [GMUW02, SKS01].
$\div$	Division	Example: $r_1 \div r_2 = r_3$ for some relations r_1, r_2, and r_3 with the schemas $R_1(a,b,c)$, $R_2(b,c)$, and $R_3(a)$, respectively. Relation r_1 is called dividend, r_2 divisor, and r_3 quotient.
$\div^*$	Set containment division/ Great divide/ Generalized division	Example: $r_1 \div^* r_2 = r_3$ for some relations r_1, r_2, and r_3 with the schemas $R_1(a,b,c)$, $R_2(b,c,d,e)$, and $R_3(a,d,e)$, respectively.
$\bowtie_{\subseteq}$	Set containment join	Example: $r_1 \bowtie_{b\subseteq c} r_2 = r_3$ for some relations r_1, r_2, and r_3 with the schemas $R_1(a,b)$, $R_2(c,d,e)$, and $R_3(a,b,c,d,e)$, respectively, where b and c are set-valued attributes.

Table 2.3: Overview of the logical operators of the relational algebra used in this thesis

B) and $R_2(B)$ be relation schemas, and let $r_1(R_1)$ and $r_2(R_2)$ be relations on these schemas. We

call r_1 *dividend*, r_2 *divisor*, and r_3 *quotient* of the division operation $r_1 \div r_2 = r_3$. The schema of r_3 is $R_3(A)$. We call B, the subset of R_1's attributes that correspond to the attributes of R_2, the *dividend's divisor attributes* and A, the remaining attributes of R_1, the *dividend's quotient attributes* because they correspond to the attributes of the quotient schema R_3. Hence, we can write $r_1(A \cup B) \div r_2(B) = r_3(A)$.

The original definition of the division operator was given by Codd [Cod72], formulated as a query in tuple relational calculus:

DEFINITION 1 (CODD'S DIVISION): $r_1 \div r_2 = \{t \mid t = t_1.A \wedge t_1 \in r_1 \wedge r_2 \subseteq i_{r_1}(t)\}$, *where* $i_{r_1}(x)$ *is called the* image set of x under r_1 *and is defined by* $i_{r_1}(x) = \{y \mid (x,y) \in r_1\}$

In this calculus expression, the term $t = t_1.A$ means that a tuple in the result, i.e., in the quotient, consists of the attribute values for A of the dividend tuple t_1.

An equivalent definition of Codd's original definition using tuple relational calculus is given by Darwen and Date [DD92]:[1]

DEFINITION 2 (DARWEN'S DIVISION):
$$r_1 \div r_2 = \{t \mid \forall t_2 \in r_2 \exists t_1 \in r_1 : t = t_1.A \wedge t_1.B = t_2.B\}.$$

This definition shows why relational division is considered the algebraic counterpart of the universal quantifier ($\forall$).

Codd writes that "division is so-named because $(r_1 \times r_2) \div r_2 = r_1$." If we first apply the division and then the product, however, we have no equivalence, according to [Dat94], page 179: $(r_1 \div r_2) \times r_2 \subseteq r_1$. Definition 1 states that division is a set containment test problem. Codd gives another equivalent definition of the division operator, which he attributes to Paul Healy of IBM Research, San Jose, which uses other algebra operators. Maier [Mai83] also gives this equivalent definition in Exercise 3.3a, page 39:[2]

DEFINITION 3 (HEALY'S DIVISION): $r_1 \div r_2 = \pi_A(r_1) - \pi_A((\pi_A(r_1) \times r_2) - r_1)$.

In Exercise 3.3b of the same book, there is an alternative definition of the division operator:

DEFINITION 4 (MAIER'S DIVISION): $r_1 \div r_2 = \bigcap_{t \in r_2} \pi_A(\sigma_{B=t}(r_1))$.

This idea has been used for an algorithm called *stream-join division* that will be discussed in Section 3.3.3.5 on page 46.

In Claußen et al. [CKMP97], an equivalent definition of the division operator using semi-join ($\ltimes$) as well as anti-semi-join ($\overline{\ltimes}$) and left outer join ($\rtimes$), is given:

DEFINITION 5 (CLAUSSEN'S DIVISION): $r_1 \div r_2 = ((r_1 \ltimes r_2) \rtimes r_2) \,\overline{\ltimes}\, r_2$.

Abiteboul et al. [AHV95] give yet another definition of division, which is very similar to Codd's Definition 1:

[1] For simplicity, we name this definition and all defintions that follow after the first author only.

[2] In this definition, Maier actually uses the join symbol $\bowtie$ instead of the Cartesian product symbol $\times$. His definition of the join operator subsumes the Cartesian product [Mai83], page 17: "The definition of join does not require that R_1 and R_2 have a nonempty intersection. If $R_1 \cap R_2 = \emptyset$, then $r_1 \bowtie r_2$ is the Cartesian product of r_1 and r_2." We have adapted the original relation names to ours in this citation.

DEFINITION 6 (ABITEBOUL'S DIVISION): $r_1 \div r_2 = \{t \in \pi_A(r_1) \mid (t \ltimes r_2) \subseteq r_1\}$.

The previous definitions are all based on testing the existence (by means of join operators, etc.) or non-existence (by means of difference or anti-semi-join operators) of attribute values. These tests try to directly match values of the elements (B) in the sets of the dividend and values of the elements in the divisor. A different view on the division problem is an indirect one that is achieved by *counting* the number of elements belonging to a quotient group of the dividend r_1 and the number of tuples in the divisor relation r_2 (the cardinality of r_2). Then, a quotient group is the set of tuples having the same value for the quotient attributes A. Using an extended operator of the relational algebra as defined in [GMUW02], the division problem can be defined with the help of the *grouping operator* ${}_G\gamma_F(r_1)$, where G is a list of attributes of r_1 and F is a list of aggregation functions applied to an attribute of r_1, as sketched in Table 2.3. A name is assigned to the attribute corresponding to the aggregation using an arrow. The grouping operator first partitions the tuples of r_1 into groups, where each group is given by a unique value of all non-aggregated attributes in L. Then, one tuple is generated for each group by employing the aggregate functions on the respective attributes of all tuples of a group. If no attribute is given in L then the entire relation r_1 is one group. This counting approach has been described, e.g., in [GC95], however without using relational algebra. The division operator can thus be defined as follows:

DEFINITION 7 (GRAEFE'S DIVISION): $r_1 \div r_2 = \pi_A\left({}_A\gamma_{\text{count}(B)\to c}(r_1 \ltimes r_2) \bowtie \gamma_{\text{count}(B)\to c}(r_2)\right)$.

In Definition 7, the expression $r_1 \ltimes r_2$ makes sure that only those tuples of r_1 are aggregated that contain values that are also contained in r_2. Then, one compares the cardinality of r_2 with the number of tuples in each group defined by a distinct value of the quotient attributes of r_1. Finally, if the numbers are equal, the aggregate values are removed, i.e., the result relation comprises only the quotient attributes A.

If we know that r_1 contains no divisor values other than in r_2, i.e., $r_1 \ltimes r_2 = r_1$, which can be ensured by a foreign key constraint $R_1.B \to R_2$, then one can simplify Definition 7 to

$$r_1 \div r_2 = \pi_A\left({}_A\gamma_{\text{count}(B)\to c}(r_1) \bowtie \gamma_{\text{count}(B)\to c}(r_2)\right).$$

In [DD92], the basic division operator was called *small divide* to distinguish it from a generalization of it, called great divide, to be discussed next.

2.3.2 The Great Divide

Before we discuss three equivalent definitions of an extended division operator, we briefly consider another operator related to them. Let $R_1(A \cup B_1)$, $R_2(B_2 \cup C)$, and $R_3(A \cup B_1 \cup B_2 \cup C)$ be relation schemas, where $A = \{a_1, \ldots, a_m\}$, $B_1 = \{b_1\}$, $B_2 = \{b_2\}$, and $C = \{c_1, \ldots, c_o\}$ are attribute sets. The sets A and C are disjoint and may be empty, B_1 and B_2 are disjoint and nonempty, A and B_1 are disjoint, and B_2 and C are disjoint. The sets B_1 and B_2 consist of a single *set-valued* attribute, respectively. Let $r_1(R_1)$, $r_2(R_2)$, and $r_3(R_3)$ be relations on these schemas. The *set containment join* $r_1 \bowtie_{b_1 \supseteq b_2} r_2 = r_3$ is a join between the set-valued attributes b_1 and b_2, where we

a	b
1	1
1	4
2	1
2	2
2	3
2	4
3	1
3	3
3	4

(a) r_1 (dividend)

b
1
3

(b) r_2 (divisor)

a
2
3

(c) r_3 (quotient)

Figure 2.1: Division: $r_1 \div r_2 = r_3$

a	b
1	1
1	4
2	1
2	2
2	3
2	4
3	1
3	3
3	4

(a) r_1 (dividend)

b	c
1	1
2	1
4	1
1	2
3	2

(b) r_2 (divisor)

a	c
2	1
2	2
3	2

(c) r_3 (quotient)

Figure 2.2: Set containment division: $r_1 \div^* r_2 = r_3$

a	b_1
1	$\{1,4\}$
2	$\{1,2,3,4\}$
3	$\{1,3,4\}$

(a) r_1

b_2	c
$\{1,2,4\}$	1
$\{1,3\}$	2

(b) r_2

a	b_1	b_2	c
2	$\{1,2,3,4\}$	$\{1,2,4\}$	1
2	$\{1,2,3,4\}$	$\{1,3\}$	2
3	$\{1,3,4\}$	$\{1,3\}$	2

(c) r_3

Figure 2.3: Set containment join: $r_1 \bowtie_{b_1 \supseteq b_2} r_2 = r_3$

ask for the combinations of tuples in $t_1 \in r_1$ and $t_2 \in r_2$ such that set $t_1.b_1$ contains all elements of set $t_2.b_2$. Several efficient algorithms and strategies for realizing this operator in an RDBMS

have been proposed [HM97, MGM03, MGM02a, Ram02, RPNK00].

We have recently suggested a generalization of division that we called *set containment division*, denoted by $\div_1^*$, because of its similarity to the set containment join [RSMW03]. We have devised several algorithms for this operator and implemented them in a query execution engine. The algorithms will be discussed in Chapter 4. Let $R_1(A \cup B)$, $R_2(B \cup C)$, and $R_3(A \cup C)$ be relation schemas, where $A = \{a_1, \ldots, a_m\}$, $B = \{b_1, \ldots, b_n\}$, and $C = \{c_1, \ldots, c_o\}$ are nonempty sets of attributes, A and B are disjoint, and B and C are disjoint. Let $r_1(R_1)$, $r_2(R_2)$, and $r_3(R_3)$ be relations on these schemas. Although we define a new operator, we continue to use the terms dividend, divisor, and quotient for the relations r_1, r_2, and r_3, respectively. The dividend relation r_1 has the same schema as for the small divide. However, the divisor relation r_2 has additional attributes C. The set containment division operator is defined as follows:

DEFINITION 8 (SET CONTAINMENT DIVISION): $r_1 \div_1^* r_2 = \bigcup_{t \in \pi_C(r_2)} (r_1 \div \pi_B(\sigma_{C=t}(r_2))) \times (t)$

The idea is to iterate over the groups defined by the attributes $r_2.C$. Each group is a separate divisor for a division with dividend r_1. We "attach" the divisor group value to the resulting quotient tuples by a Cartesian product between each quotient group and a one-tuple relation (t).

The similarity to the set containment join operator is illustrated in Figures 2.2 and 2.3. Except for the fact that the input relations of set containment join are not in first normal form due to the set-valued attributes and that set containment division does not preserve the "join" attributes in B, the operators have the same semantics. Another slight difference is that set containment join allows empty sets as join attribute values whereas set containment division does not have the notion of an empty set. Furthermore, for the set containment join the attribute sets A and C may be empty.

Set containment division takes as input relations that are in the first normal form (1NF), which requires that the attributes have only atomic values. In contrast, a non-atomic domain has composite or set-valued values. A composite value occurs, e.g., for an *address* attribute that is composed of attribute values for *street*, *zip-code*, *country*, etc. An example of a set-valued attribute is *phone-numbers* to represent all (zero or more) phone numbers of a person. Set-valued attributes are required in the input of the set containment join operator, discussed next.[3]

In 1982, Robert Demolombe suggested a *generalized division* operator, denoted by $\div_2^*$, that is equivalent (see Theorem 1 below) to set containment division [Dem82]. Besides a definition of the operator in tuple relational calculus and predicate calculus, he gives an algebraic definition:

DEFINITION 9 (GENERALIZED DIVISION):
$r_1 \div_2^* r_2 = (\pi_A(r_1) \times \pi_C(r_2)) - \pi_{A \cup C}((\pi_A(r_1) \times r_2) - (r_1 \times \pi_C(r_2)))$

In 1988, Stephen Todd suggested—presumably independent from Demolombe—a generalized division operator but he did not publish it himself. However, it has been discussed by Darwen

[3]We oversimplify our presentation here. Strictly speaking, set containment division does *not* require input relations in 1NF, but it requires that each element of a "set" is represented by some attribute values of a *separate tuple*. In fact, these values might belong to a set-valued or composite attribute. For example, the set elements represent web site visits, where each visit is a set itself that consists of (URL, timestamp) pairs, indicating the time a user has requested the document with the URL. A relation in 1NF does not allow any set-valued or composite attributes at all.

and Date [DD92], where it was called *great divide*, denoted by $\div_3^*$. A definition in relational algebra is given by the following expression:

DEFINITION 10 (GREAT DIVIDE):
$r_1 \div_3^* r_2 = (\pi_A(r_1) \times \pi_C(r_2)) - \pi_{A \cup C}((\pi_A(r_1) \times r_2) - (r_1 \bowtie r_2))$.

Definition 10 differs only slightly from Definition 9 of generalized division. It uses a join instead of a Cartesian product. Darwen and Date write that great divide degenerates to small divide, as specified in Definition 3, if $C = \emptyset$ [DD92]. We prove in Appendix A on page 158 the following

THEOREM 1: *Set containment division ($\div_1^*$), generalized division ($\div_2^*$), and great divide ($\div_3^*$) are equivalent operators, i.e.,* $\div_1^* \equiv \div_2^* \equiv \div_3^*$.

The three definitions have been suggested independently. However, while the publications on generalized division [Dem82] and great divide [DD92] solely focus on the relationship between the *logical* operator and the basic division operator, our work on the set containment division operator [RSMW03, Ran03] puts its emphasis on algorithms to implement *physical* operators and to investigate applications for it. In the rest of the document (except for the appendices), we will use our term *set containment division* for the operator $\div^*$.

2.3.3 Interchanging Operators

We can simulate set containment division by set containment join and vice versa. This is useful if an RDBMS offers an efficient implementation of only one of the two operators. By interchanging operators, we can solve problems for one operator by using the other. However, the interchange incurs a cost for pre- and post-processing of the dividend and divisor table.

Let b_1 and b_2 be attributes and let A and C be attribute sets. Let r_1, r_2, and r_3 be relations with the schemas $R_1(A \cup \{b\})$, $R_2(\{b\} \cup C)$, and $R_3(A \cup C)$, respectively. Let r_1', r_2', and r_3' be relations with the schemas $R_1'(A \cup \{b_1\})$, $R_2'(\{b_2\} \cup C)$, $R_3'(A \cup \{b_1, b_2\} \cup C)$, respectively, where b_1 and b_2 are set-valued attributes.

A set containment join problem $r_1' \bowtie_{b_1 \supseteq b_2} r_2' = r_3'$ can be solved by a set containment division operator as follows:

1. Unnest b_1 and b_2 into groups of tuples and rename attributes b_1 and b_2 to b: $r_1' \Rightarrow r_1$, $r_2' \Rightarrow r_2$.

2. Apply the set containment division: $r_1 \div^* r_2 = r_3$.

3. Incorporate the set values of b_1 and b_2 into the result: $r_3 \bowtie r_1' \bowtie r_2' = r_3'$.

Note that we can use the extended projection operator π, listed in Table 2.3 on page 7 to rename columns. By definition, set containment division requires to have the same set element attribute(s) to appear in both input relation schemas.

Analogously, a set containment division problem $\pi_{b_1 \to b}(r_1) \div^* \pi_{b_2 \to b}(r_2) = r_3$ can be simulated with the help of a set containment join operator as follows:

s#
S1
S2
S3

(a) *s*

s#	*p#*
S1	P1
S1	P2
S1	P4
S3	P2
S3	P3

(b) *sp*

p#	*color*
P1	green
P2	blue
P3	green
P4	blue

(c) *p*

Figure 2.4: An example illustrating the empty divisor problem

1. Nest the values b_1 and b_2 of each group defined by A and C, respectively, into a set value: $r_1 \Rightarrow r'_1$, $r_2 \Rightarrow r'_2$.

2. Apply the set containment join: $r'_1 \bowtie_{b_1 \supseteq b_2} r'_2 = r'_3$.

3. Remove the set values of b_1 and b_2 from the result: $\pi_{A \cup C}(r'_3) = r_3$.

2.3.4 The Empty Divisor Problem

When we formulate a query involving a universal quantification like "Find the suppliers who supply all parts of color *c*," we are led to believe that it can be easily solved using the division operator.

Suppose, we have the well-known supplier-parts database [DD92, Dat94] consisting of a supplier table *s*, a table *sp* indicating which supplier supplies which part, and a parts table *p*, as illustrated in Figure 2.4. Such tables typically occur in a data warehouse of a company [Tra02].

Now, when we try to find the suppliers that supply all blue parts, we can describe this problem by the algebraic expression $sp \div \pi_{p\#}(\sigma_{color=blue}(p)) = sp \div \{(\text{P2}), (\text{P4})\}$, which in our example correctly yields supplier number S1.

What happens if we want to find the suppliers supplying all *red* parts? Let us apply the previous expression, now using the red color: $sp \div \pi_{p\#}(\sigma_{color=red}(p)) = sp \div \emptyset = \{(\text{S1}), (\text{S3})\}$. Indeed, suppliers S1 and S3 both supply all red parts, i.e., they supply all parts in the empty set. But this is also true for supplier S2. *Any* set contains the empty set.

This empty divisor case is discussed in detail in [DD92]. The problem is that our intuition leads us to believe that our expression is equivalent to the given query. However, the expression actually only solved the problem "Find the suppliers who supply all parts with color *c and who supply at least one part*." The authors suggest a modified operator called *divide per* that solves the original problem. We do not give their algebraic definition here but mention that the new operator also takes the *suppliers* table *s* as parameter in addition to the relations *sp* and *p*. Using the new syntax, the correct expression for our example would be $s \div \pi_{p\#}(\sigma_{color=red}(p))\, per\ sp$. The authors also suggest a "divide per" version for the great divide operator.

2.3.5 The HAS Operator

Carlis proposed a generalization of the division operator, called *HAS* [Car86]. He argues that "division is misnamed" because there are more operators $\circ$ than division ($\div$) that fulfill the equation $(r_1 \times r_2) \circ r_2 = r_1$. He further claims that division is "hard to understand" because, among other arguments, "division is the only algebra operation that gives students any trouble." Finally, he writes that division is "insufficient" because it is not flexible enough, it allows only queries of the form "find the sets that contain *all* elements of a given set" but it does not help for queries asking for sets that contain, e.g., at least three elements of a given set.

The HAS operator involves three relations:

- r_1 contains entities about which we want the answer if it qualifies in the result,
- r_2 contains entities that are used for the qualification, and
- r_3 contains the relationships between the entities in r_1 and r_2.

For example, in the supplier-parts database in Figure 2.4, $r_1 = s$, $r_2 = p$, and $r_3 = sp$. In addition, the HAS operator uses a combination of six "adverbs," called *associations*, to describe the qualification: *strictly more than*, *strictly less than*, *some of but not all plus something else*, *exactly*, *none of plus something else*, and *none at all*. There are $2^6 - 1 = 63$ possible combinations to choose between one and six associations for a specific HAS operator. Such a combination is considered as a disjunction of the participating associations.

We illustrate the algebra syntax used in [Car86] by showing how relational division can be expressed by the HAS operator using one of the 63 association combinations:

r_1VIA r_3
HAS (*exactly* or *strictly more than*)
OF r_2

The combination "*exactly* or *strictly more than*" is equivalent to the adverb "at least," typically used to describe division.[4]

Another query that can be expressed using the HAS operator, but only with considerable effort using basic operators, is "Which suppliers do not supply all blue parts?":

sVIA sp
HAS (*strictly less than* or *some of plus something else* or *none of*)
OF $\pi_{color=blue}(p)$

The combination used in this query is equivalent to the adverb "not all of."[5]

[4] In the association combination overview in [Car86], this combination has number 9, denoted by the symbol $=>>$.

[5] In the association combination overview in [Car86], this combination has number 33, denoted by the symbol $<$.

2.4 Algebraic Laws

Let us briefly describe the role of algebraic laws for query optimization. Before a query is executed by the query execution engine of a relational database management system (RDBMS), the query optimizer rewrites the algebraic representation of the query according to transformation rules. Typically, one type of transformation rules is based on algebraic laws and the other maps a logical operator (e.g., join) to a physical operator (e.g., hash-join). An algebraic law is an equivalence between two different representations of an algebraic expression. Both representations describe the same set of result tuples for every possible database content. Together with heuristics and/or cost estimations, the optimizer applies transformation rules to subexpressions of the query such that the entire query can be evaluated with the minimal resource consumption or the shortest response time. Algebraic laws for the basic operators of the relational algebra are discussed, e.g., in [JK84, GMUW02]. The implementation of transformation rules (rewrite rules) in a commercial RDBMS are described, e.g., in [Loh88, PHH92]. Frameworks for building query optimizers, like Cascades [Gra95] and XXL [BBD+01], offer the possibility to study the code that is required to realize transformation rules in an RDBMS.

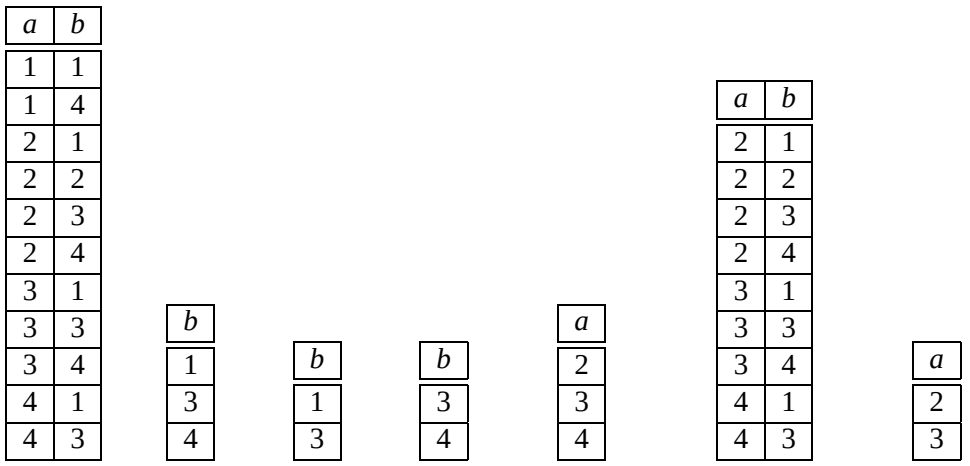
To the best of our knowledge, no commercial RDBMS has an implementation of relational division. One reason is that there is no keyword in SQL that would allow to express universal quantification intuitively. Another reason is that set containment tests are not considered as important as existential element tests that is realized by the join operator. However, special applications like frequent itemset discovery could be processed efficiently and formulated more intuitively if division would be a first-class operator. Let us suppose that an RDBMS offers one or more efficient implementations of division, i.e., physical division operators like hash-division or merge-sort division [GC95, RSMW03]. An optimizer could replace the division operator by an expression that simulates the operator and apply transformation rules on the operators in the expression. In addition, it should also be able to apply rewrite rules to the division operator directly since efficient implementations are available in the query execution engine.

Therefore, in the following we present several algebraic laws that either preserve the division operator or produce some non-trivial rewrite result. Note that there are an infinite number of equivalent expressions for any given algebraic expression. We have tried to distill the most useful and interesting laws such that a rule-based optimizer might indeed decide to employ them to achieve an efficient query processing strategy.

No previous work has covered the optimization of queries involving division or generalized division. However, applications like frequent itemset discovery would benefit from a division syntax in SQL, an efficient implementation of the operator in a query execution engine, and the integration of appropriate transformation rules into an optimizer.

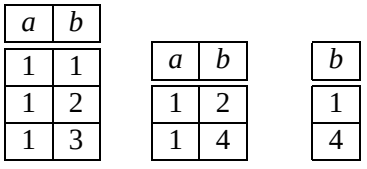
Some of the algebraic laws discussed in this section require the notion of a *partitioned* relation. We use the following notations and definitions for partitions. Let r_i' and r_i'' denote nonempty *horizontal* partitions of r_i such that $r_i' \cup r_i'' = r_i$, where $i \in \{1,2\}$, i.e., we define a decomposition of r_i's *tuples*. Let r_i^* and r_i^{**} denote relations that conform to the schemas of the *vertical* partitions R_i^* and R_i^{**} of R_i, respectively, such that $R_i^* \cup R_i^{**} = R_i$, where $i \in \{1,2\}$. Hence, we define a decomposition of R_i's *attributes*.

For the laws that follow, we will indicate when we require partitions to be disjoint or not.



(a) r_1 (b) r_2 (c) r_2' (d) r_2'' (e) $r_1 \div r_2'$ (f) $r_1 \ltimes (r_1 \div r_2')$ (g) r_3

Figure 2.5: An example for Law 1



(a) r_1' (b) r_1'' (c) r_2

Figure 2.6: The precondition of Law 2 is not fulfilled

The proofs of the laws and theorems can be found in Appendix A on page 153.

2.4.1 Algebraic Laws for Division

Before we present the laws, we state two theorems.

THEOREM 2: *Division is non-commutative, i.e.,* $r_1 \div r_2 \neq r_2 \div r_1$ *for relations* r_1 *and* r_2.

THEOREM 3: *Division is non-associative, i.e.,* $r_1 \div (r_2 \div r_3) \neq (r_1 \div r_2) \div r_3$ *for nonempty relations* r_1, r_2, *and* r_3.

2.4.1.1 Union

When the *divisor* r_2 is decomposed into horizontal partitions then one can divide by these divisors separately:

LAW 1: $r_1 \div (r_2' \cup r_2'') = (r_1 \ltimes (r_1 \div r_2')) \div r_2''$.

This law holds also for overlapping divisor partitions, as illustrated in the example in Figure 2.5. It can help an RDBMS to employ pipeline parallelism as follows. Suppose, r_1 is grouped on A. We can employ efficient group-preserving algorithms for the inner division $r_1 \div r_2'$ and the semi-join and deliver the result as the dividend of the outer division, which again can employ a group-preserving algorithm.

When we decompose the *dividend* instead of the divisor, we must take care of the situation sketched in Figure 2.6. There is a quotient candidate value ($a = 1$) whose tuples are dispersed across the dividend relations but none of the groups contains *all* values of the divisor. However, the union of the groups does. In other words, $r_1' \div r_2 = \emptyset$ and $r_1'' \div r_2 = \emptyset$ but $(r_1' \cup r_1'') \div r_2 \neq \emptyset$. We have to exclude this situation in the precondition of Law 2. Formally, the following precondition must be true:

$$c_1(r_1', r_1'') \equiv \forall a \in \pi_A(r_1') \cap \pi_A(r_1'') : r_2 \subseteq \pi_B\left(\sigma_{A=a}\left(r_1'\right)\right) \vee r_2 \subseteq \pi_B\left(\sigma_{A=a}\left(r_1''\right)\right) \vee r_2 \not\subseteq \pi_B\left(\sigma_{A=a}\left(r_1'\right) \cup \sigma_{A=a}\left(r_1''\right)\right) \tag{2.1}$$

LAW 2: If condition $c_1\left(r_1', r_1''\right) \equiv \mathit{true}$ then $(r_1' \cup r_1'') \div r_2 = (r_1' \div r_2) \cup (r_1'' \div r_2)$.

Since testing condition c_1 can be expensive, we can use a stricter condition c_2 that is easier to check:

$$c_2(r_1', r_1'') \equiv \pi_A\left(r_1'\right) \cap \pi_A\left(r_1''\right) = \emptyset. \tag{2.2}$$

It can be shown easily that for any relations $r_1 = r_1' \cup r_1''$ and r_2 as defined before, if c_2 holds then also c_1 holds. By using condition c_2 instead of c_1 with Law 2, an RDBMS can parallelize a query execution with degree 2 as follows. Suppose that the query execution engine can access the data in table r_1 via an index on A. We can employ two parallel scans: one that starts with the lowest value of A and scans the leaves of the index in ascending order of A and another that starts with the highest value of A and retrieves data in descending order of A. Both scans stop as soon as they encounter the same value for A. Exactly one of them has to process the entire last group. Higher degrees of parallelism can be achieved by partitioning r_1 into $n > 2$ partitions.

2.4.1.2 Selection

Let $p(X)$ denote a predicate involving only elements of a set of attributes X. Since only r_1 contains the attribute set A, we can state the following "selection push-down" law:

LAW 3: $\sigma_{p(A)}\left(r_1 \div r_2\right) = \sigma_{p(A)}\left(r_1\right) \div r_2$.

For a predicate that involves only attributes in B, the following "replicate-selection" law holds:

LAW 4: $r_1 \div \sigma_{p(B)}\left(r_2\right) = \sigma_{p(B)}\left(r_1\right) \div \sigma_{p(B)}\left(r_2\right)$.

EXAMPLE 1: For a predicate that involves only attributes in B, but that is applied to the dividend

only, the following expressions are equivalent:

$$\sigma_{p(B)}(r_1) \div r_2 = \left(\sigma_{p(B)}(r_1) \div \sigma_{p(B)}(r_2)\right) - \pi_A\left(\pi_A(r_1) \times \sigma_{\neg p(B)}(r_2)\right).$$

This expression is very similar to Law 4. We only have to take care of the situation when $\sigma_{\neg p(B)}(r_2) \neq \emptyset$. In this case, the expression $\sigma_{p(B)}(r_1) \div r_2$ is equal to the empty set because no dividend tuple has a value of B that can match a tuple in $\sigma_{\neg p(B)}(r_2)$. Hence, if $\sigma_{\neg p(B)}(r_2)$ contains at least one tuple, we achieve that the result relation is empty by simply removing all A values in r_1 from the quotient candidates in $\sigma_{p(B)}(r_1) \div \sigma_{p(B)}(r_2)$. The Cartesian product is merely used to "switch" $\pi_A(r_1)$ "on or off."[6] To make our argumentation clearer, we could rewrite our expression as follows:

$$\sigma_{p(B)}(r_1) \div r_2 = \begin{cases} \sigma_{p(B)}(r_1) \div \sigma_{p(B)}(r_2) & \text{if } \sigma_{\neg p(B)}(r_2) = \emptyset, \text{ and} \\ \emptyset & \text{otherwise.} \end{cases}$$

To illustrate our first expression, Figure 2.7 shows an example where the division yields an empty quotient relation. □

2.4.1.3 Intersection

We can push division into intersections or differences of dividend relations.

LAW 5: $(r_1' \cap r_1'') \div r_2 = (r_1' \div r_2) \cap (r_1'' \div r_2)$.

2.4.1.4 Difference

For the difference operator we require the precondition that $\pi_A(r_1')$ and $\pi_A(r_1'')$ are disjoint.[7]

LAW 6: If $\pi_A(r_1') \cap \pi_A(r_1'') = \emptyset$ then $(r_1' \div r_2) - (r_1'' \div r_2) = r_1' \div r_2$.

2.4.1.5 Cartesian Product

Let A_1 and A_2 be disjoint subsets of the attribute set A such that $A_1 \cup A_2 = A$. Let r_1^* be a relation with schema $R_1^*(A_1)$ and r_1^{**} be a relation with schema $R_1^{**}(A_2 \cup B)$. As usual, let $R_2(B)$ be the schema of the divisor r_2. Then it suffices to apply the division only to some of the attributes of the dividend:

LAW 7: $(r_1^* \times r_1^{**}) \div r_2 = r_1^* \times (r_1^{**} \div r_2)$.

Figure 2.8 illustrates Law 7 with an example. The law can help when the query optimizer finds that a predicate θ of a theta-join $\bowtie_\theta$ is always true since $\bowtie_{true} \equiv \times$.

[6] Of course, it would suffice to combine $\pi_A(r_1)$ with only a single tuple of $\sigma_{\neg p(B)}(r_2)$ by the Cartesian product.

[7] This not the weakest precondition. For the equivalence of the law to hold, it would suffice to require that $\forall a \in \sigma_{A=a}(\pi_A(r_1') \cup \pi_A(r_1'')) : r_2 \subseteq \sigma_{A=a}(\pi_A(r_1')) \vee r_2 \subseteq \sigma_{A=a}(\pi_A(r_1'')) \vee r_2 \not\subseteq \sigma_{A=a}(\pi_A(r_1') \cup \pi_A(r_1''))$. However, we prove the law only for the stronger precondition $\pi_A(r_1') \cap \pi_A(r_1'') = \emptyset$.

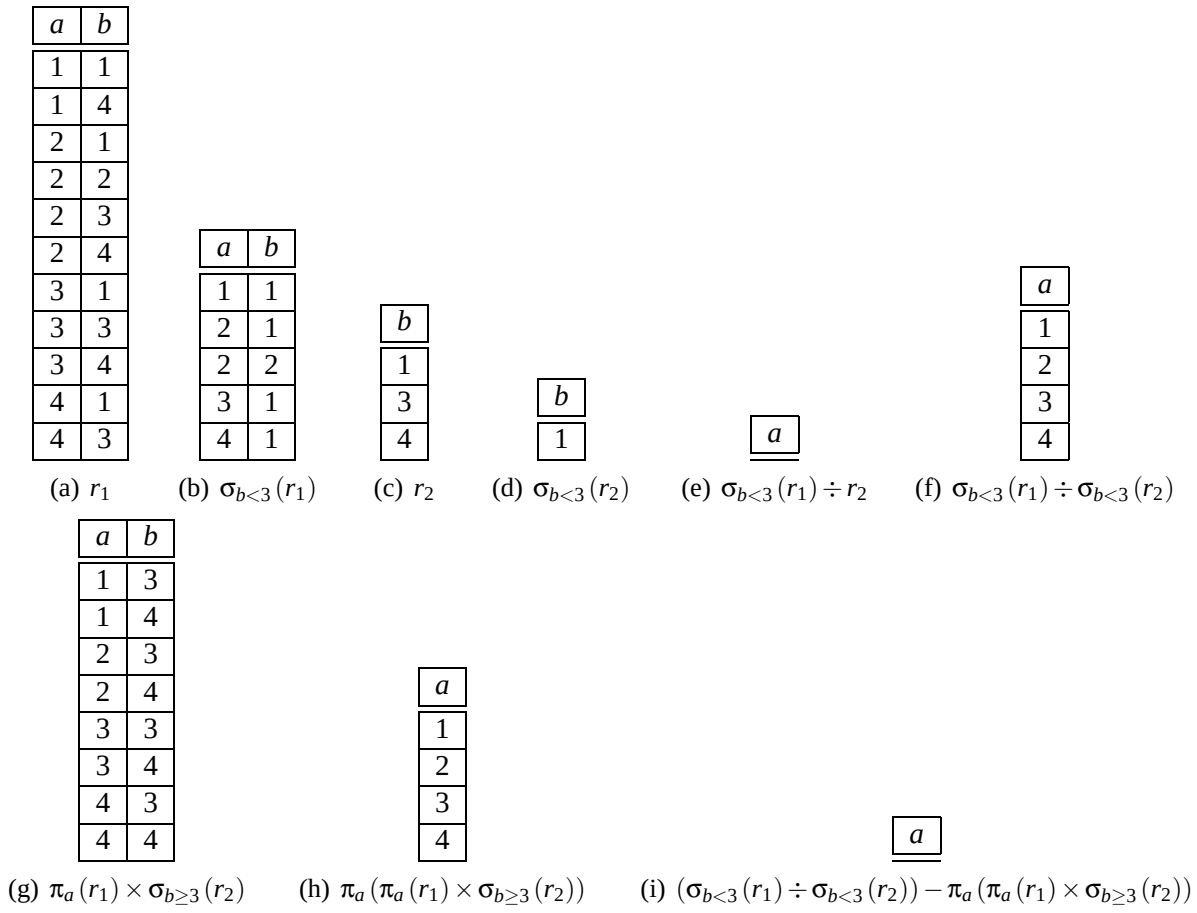


Figure 2.7: An illustration for Example 1

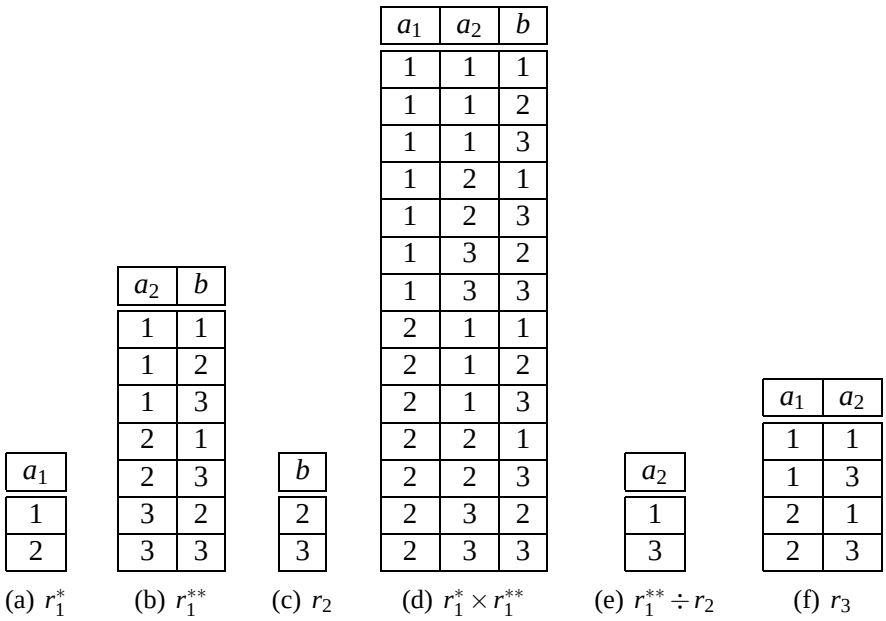
Let B_1 and B_2 be disjoint nonempty subsets of the attribute set B such that $B_1 \cup B_2 = B$. Let r_1^* be a relation with schema $R_1^*(A \cup B_1)$ and r_1^{**} be a relation with schema $R_1^{**}(B_2)$. Again, let $R_2(B)$ be the schema of the divisor r_2. Then, we can state the following

LAW 8: If $\pi_{B_2}(r_2) \subseteq r_1^{**}$ then $(r_1^* \times r_1^{**}) \div r_2 = r_1^* \div \pi_{B_1}(r_2)$.

Figure 2.9 illustrates Law 8 with an example.

EXAMPLE 2: With the help of Law 8 we can prove that $(r_1 \times s) \div (r_2 \times s) = r_1 \div r_2$. Let $B = B_1 \cup B_2$. We have $R_1^*(A \cup B_1)$, $R_1^{**}(B_2)$, $R_2^*(B_1)$, $R_2^{**}(B_2)$ and thus $R_1(A \cup B_1 \cup B_2)$ as the dividend schema and $R_2(B_1 \cup B_2)$ as the divisor schema. We define $s = r_1^{**} = r_2^{**}$. The condition $r_1^{**} \subseteq \pi_{R_1^{**}}(r_2)$ is fulfilled since $r_1^{**} = r_2^{**} = \pi_{R_2^{**}}(r_2) = \pi_{R_1^{**}}(r_2)$. Hence, we have

$$\begin{aligned}(r_1^* \times s) \div (r_2^* \times s) &= (r_1^* \times r_1^{**}) \div (r_2^* \times r_2^{**}) \qquad \text{(Definition of } s\text{)} \\ &= (r_1^* \times r_1^{**}) \div r_2 \qquad \text{(Definition of } R_2\text{)} \\ &= r_1^* \div \pi_{B_1}(r_2) \qquad \text{(Law 8)}\end{aligned}$$



a_1
1
2

(a) r_1^*

a_2	b
1	1
1	2
1	3
2	1
2	3
3	2
3	3

(b) r_1^{**}

b
2
3

(c) r_2

a_1	a_2	b
1	1	1
1	1	2
1	1	3
1	2	1
1	2	3
1	3	2
1	3	3
2	1	1
2	1	2
2	1	3
2	2	1
2	2	3
2	3	2
2	3	3

(d) $r_1^* \times r_1^{**}$

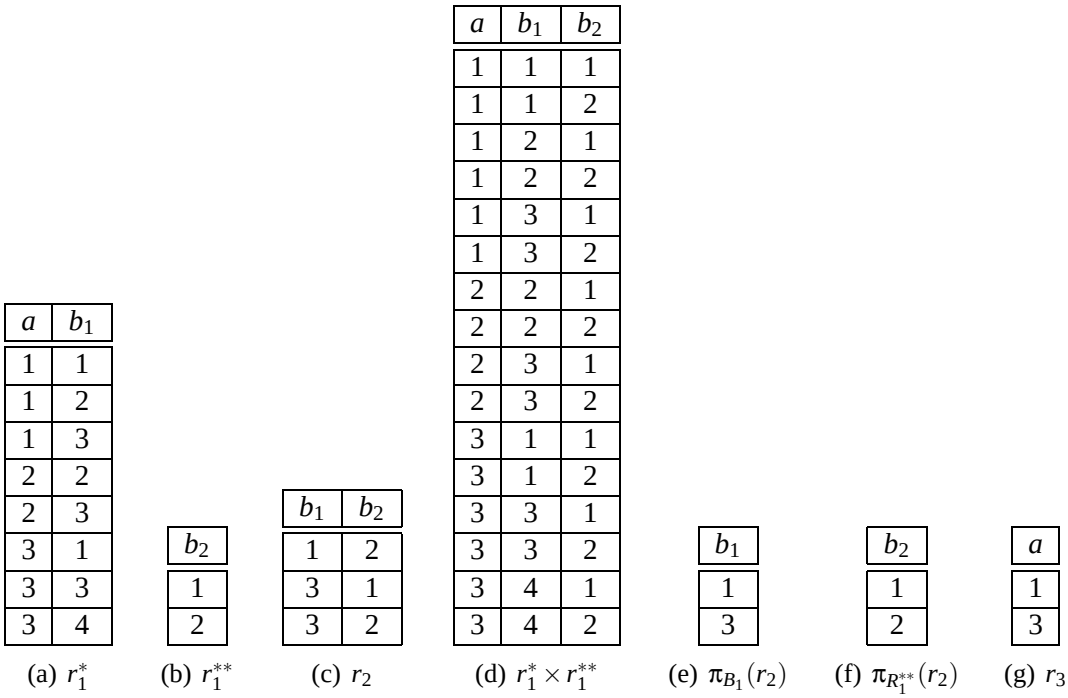
a_2
1
3

(e) $r_1^{**} \div r_2$

a_1	a_2
1	1
1	3
2	1
2	3

(f) r_3

Figure 2.8: An example for Law 7



a	b_1
1	1
1	2
1	3
2	2
2	3
3	1
3	3
3	4

(a) r_1^*

b_2
1
2

(b) r_1^{**}

b_1	b_2
1	2
3	1
3	2

(c) r_2

a	b_1	b_2
1	1	1
1	1	2
1	2	1
1	2	2
1	3	1
1	3	2
2	2	1
2	2	2
2	3	1
2	3	2
3	1	1
3	1	2
3	3	1
3	3	2
3	4	1
3	4	2

(d) $r_1^* \times r_1^{**}$

b_1
1
3

(e) $\pi_{B_1}(r_2)$

b_2
1
2

(f) $\pi_{R_1^{**}}(r_2)$

a
1
3

(g) r_3

Figure 2.9: An example for Law 8

$$= r_1^* \div r_2^* \qquad \text{(Definition of } R_2\text{)}$$

□

2.4.1.6 Join

Join, like division, is a derived operator. When a division operators occurs together with a join operator in an expression, it may be beneficial for the execution strategy of an RDBMS to rewrite the join operator and subsequently apply algebraic laws to rewrite result in combination with division. The laws involving the selection operator in Section 2.4.1.2 as well as the laws concerning the Cartesian product in Section 2.4.1.5 can be used to rewrite expressions involving join and division, since $r \bowtie_\theta s = \sigma_\theta(r \times s)$, where $\bowtie_\theta$ is a theta-join with the condition θ. The following example illustrates such a rewrite.

EXAMPLE 3: Let r_1^*, r_1^{**}, and r_2 be relations with the schemas $R_1^*(a, b_1)$, $R_1^{**}(b_2)$, and $R_2(b_1, b_2)$, respectively. Furthermore, let $r_1^{**}.b_2$ be a unique attribute and let $r_2.b_2$ be a foreign key that references r_1^{**}, i.e., $\pi_{b_2}(r_2) \subseteq r_1^{**}$. Suppose, we want to compute relation $r_3 = (r_1^* \bowtie_{b_1<b_2} r_1^{**}) \div r_2$. We can derive the following expressions:

$$\begin{aligned}
r_3 &= (r_1^* \bowtie_{b_1<b_2} r_1^{**}) \div r_2 \\
&= \sigma_{b_1<b_2}(r_1^* \times r_1^{**}) \div r_2 && \text{(Definition of theta-join)} \\
&= (\sigma_{b_1<b_2}(r_1^* \times r_1^{**}) \div \sigma_{b_1<b_2}(r_2)) - \pi_a(\pi_a(r_1^* \times r_1^{**}) \times \sigma_{b_1\geq b_2}(r_2)) && \text{(Example 1)} \\
&= ((r_1^* \times r_1^{**}) \div \sigma_{b_1<b_2}(r_2)) - \pi_a(\pi_a(r_1^* \times r_1^{**}) \times \sigma_{b_1\geq b_2}(r_2)) && \text{(Law 4)} \\
&= (r_1^* \div \pi_{b_1}(\sigma_{b_1<b_2}(r_2))) - \pi_a(\pi_a(r_1^* \times r_1^{**}) \times \sigma_{b_1\geq b_2}(r_2)) && \text{(Law 8)} \\
&= (r_1^* \div \pi_{b_1}(\sigma_{b_1<b_2}(r_2))) - \pi_a(\pi_a(r_1^*) \times \sigma_{b_1\geq b_2}(r_2)) && \text{(since } a \in R_1^* \text{ but } a \notin R_1^{**}\text{)}
\end{aligned}$$

Note that the term $\pi_a(r_1^*) \times \sigma_{b_1\geq b_2}(r_2)$ is merely used to test if $\sigma_{b_1\geq b_2}(r_2)$ contains at least one tuple. If yes, the result of the given problem is the empty set. Otherwise, it is simply $r_1^* \div \pi_{b_1}(\sigma_{b_1<b_2}(r_2))$. Figure 2.10 sketches some intermediate results that occur during the computation of our example expression.

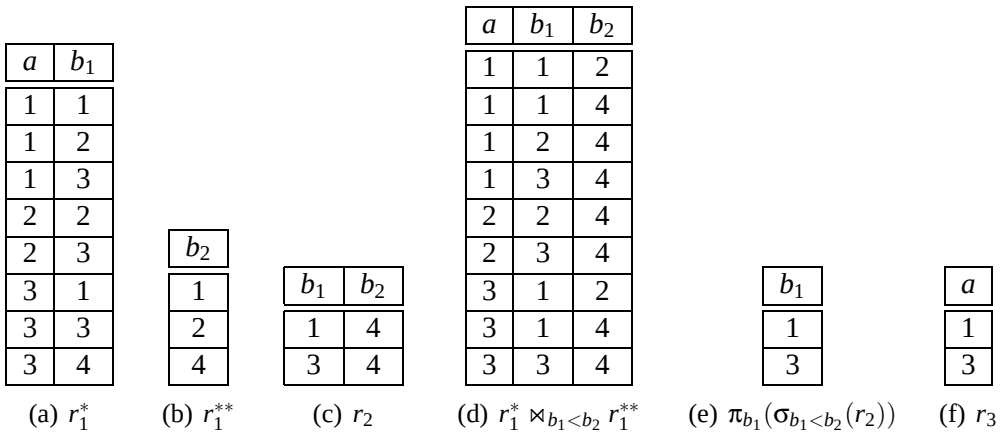
An RDBMS might be able to execute a plan based on this expression more efficiently than a plan based on the original expression because no join between r_1^* and r_1^{**} is required. Such a situation occurs, e.g., when there is no index available on $r_1^*.b_1$ and no index on $r_1^{**}.b_2$, but when there are two indexes defined on the columns b_1 and b_2 of table r_2, respectively.

We are not aware of straightforward laws that could help to rewrite join expressions involving also quotient attributes of A like in $(r_1^* \bowtie_{a+b_1<b_2} r_1^{**}) \div r_2$. □

Let us focus on a special type of join: the semi-join. Let r_3 be a relation with schema $R_3(A)$. Then we can state the following

LAW 9: $(r_1 \div r_2) \ltimes r_3 = (r_1 \ltimes r_3) \div r_2$.

This law can help an RDBMS if r_3 has few tuples and r_1 and r_2 have many tuples. It may be cheaper to keep r_3 in memory and to compute the semi-join in one scan over r_1, especially if the join is highly selective and removes many tuples from r_1. Then, the division of the join result



a	b_1
1	1
1	2
1	3
2	2
2	3
3	1
3	3
3	4

(a) r_1^*

b_2
1
2
4

(b) r_1^{**}

b_1	b_2
1	4
3	4

(c) r_2

a	b_1	b_2
1	1	2
1	1	4
1	2	4
1	3	4
2	2	4
2	3	4
3	1	2
3	1	4
3	3	4

(d) $r_1^* \ltimes_{b_1<b_2} r_1^{**}$

b_1
1
3

(e) $\pi_{b_1}(\sigma_{b_1<b_2}(r_2))$

a
1
3

(f) r_3

Figure 2.10: An illustration of Example 3

with r_2 is likely to be cheap.

2.4.1.7 Grouping

We consider two special cases involving the grouping operator. Concerning the first special case, let r_0 be a relation with schema $R_0(A \cup X)$ for some nonempty attribute set X. Let $r_1 = {}_A\gamma_{f(X)\to B}(r_0)$, where f is an aggregate function and its result is assigned to the attributes in B.[8] In other words, each quotient candidate group of the dividend consists of a single tuple. Hence, in order to find a quotient, the divisor cannot have more than one tuple. For this special case, we can formulate

$$\text{LAW 10: } r_1 \div r_2 = \begin{cases} r_1 & \text{if } \sigma_{c=0}\left(\gamma_{\text{count}(B)\to c}\left(r_2\right)\right) \neq \emptyset, \\ \pi_A\left(r_1 \ltimes r_2\right) & \text{if } \sigma_{c=1}\left(\gamma_{\text{count}(B)\to c}\left(r_2\right)\right) \neq \emptyset, \text{ and} \\ \emptyset & \text{otherwise.} \end{cases}$$

Figure 2.11 illustrates an example for this law.

Now, let us consider another special case. Let r_0 be a relation with schema $R_0(X \cup B)$ for some nonempty attribute set X. Let $r_1 = {}_B\gamma_{f(X)\to A}(r_0)$, where f is an aggregate function and its result is assigned to the attributes in A.[8] In other words, each divisor attribute value B of the dividend occurs in a single tuple, i.e., the groups defined by B have size one. Furthermore, let $r_2.B$ be a foreign key referencing $r_1.B$, i.e., $r_2.B \subseteq \pi_B\left(r_1\right)$.

Hence, there can be at most one dividend tuple for each B value. We simply have to check if $\pi_A(r_1 \ltimes r_2)$ contains a single value. If it does, then this value is the quotient. Otherwise, there is no quotient.

[8]The assignment $f(X) \to B$ is a simplification. In general, f is a list of aggregate functions $f_1, \ldots, f_n$, where $n = |B|$, such that $f(X) = (f_1(e_1(X)), \ldots, f_n(e_n(X))) = (b_1, \ldots, b_n) = B$ and $e_i(X)$ is an arithmetic expression using attributes of X, e.g., $e_5 = 7x_3 - \sqrt{x_5}$. The set X may have any number of attributes, it need not be equal to B.

a	x
1	1
1	2
1	3
2	1
2	3
3	1
3	3
3	4

(a) r_0

a	b
1	6
2	4
3	8

(b) $r_1 = {}_a\gamma_{\text{sum}(x)\to b}(r_0)$

b
4

(c) r_2

a	b
2	4

(d) $r_1 \ltimes r_2$

a
2

(e) $\pi_A(r_1 \ltimes r_2)$

Figure 2.11: An example for Law 10

x	b
1	1
1	2
1	3
2	1
2	3
3	1
3	3
3	4

(a) r_0

a	b
6	1
1	2
6	3
3	4

(b) $r_1 = {}_b\gamma_{\text{sum}(x)\to a}(r_0)$

b
1
3

(c) r_2

a	b
6	1
6	3

(d) $r_1 \ltimes r_2$

a
6

(e) $\pi_A(r_1 \ltimes r_2)$

Figure 2.12: An example for Law 11

$$\text{LAW 11: } r_1 \div r_2 = \begin{cases} \pi_A(r_1 \ltimes r_2) & \text{if } \sigma_{c=1}\left(\gamma_{\text{count}(A)\to c}\left(\pi_A(r_1 \ltimes r_2)\right)\right) \neq \emptyset, \text{ and} \\ \emptyset & \text{otherwise.} \end{cases}$$

Figure 2.12 illustrates an example for this law.

The two laws involving the grouping operator can improve the query execution time considerably because the division operation is replaced by a single join operation and a projection on the join result. However, since the laws apply only in special cases, we believe that it is not very likely that RDBMS implementors would opt to consider these laws.

2.4.2 Algebraic Laws for Generalized Division

We have identified several laws for the extended division operator $\div^*$. In the following, we show some of the laws that we consider as important.

2.4.2.1 Union

When the *divisor* r_2 is decomposed into horizontal partitions then one can divide by these divisors separately:

LAW 12: If $\pi_C(r_2') \cap \pi_C(r_2'') = \emptyset$ then $r_1 \div^* (r_2' \cup r_2'') = (r_1 \div^* r_2') \cup (r_1 \div^* r_2'')$.

This law allows to parallelize the execution of a query. Suppose that the dividend r_1 is replicated on n nodes of a query execution engine and that the divisor is equally distributed according to a hash function on $r_2.C$ across the nodes. Then it is possible to reduce the execution time to $\frac{1}{n}$ of the original time provided that the division execution is considerably more expensive than the final union/merge operator plus the cost for data shipping to and from the nodes.

2.4.2.2 Selection

The following law is the same as Law 3 for the division operator.

LAW 13: $\sigma_{p(A)}(r_1 \div^* r_2) = \sigma_{p(A)}(r_1) \div^* r_2$.

A similar law holds for attribute C of the divisor relation:

LAW 14: $\sigma_{p(C)}(r_1 \div^* r_2) = r_1 \div^* \sigma_{p(C)}(r_2)$.

The following law is the same as Law 4 for division:

LAW 15: $r_1 \div^* \sigma_{p(B)}(r_2) = \sigma_{p(B)}(r_1) \div^* \sigma_{p(B)}(r_2)$.

2.5 Universal Quantification and SQL

In this section, we show SQL expressions for division and explain how they give rise to two classes of algorithms based on the kind of data structures employed.

2.5.1 Classical Division

The commonly used approach to express universal quantification uses two "NOT EXISTS" clauses due to the mathematical equivalence $\forall x \exists y : f(x,y) \equiv \neg \exists x \neg \exists y : f(x,y)$, where f is a predicate involving the variables x and y. The following example query based on the supplier-parts database illustrated in Figure 2.4 (page 13) asks for the suppliers who supply all blue parts.[9]

```
SELECT DISTINCT s#
FROM   sp AS sp1
WHERE  NOT EXISTS (
         SELECT *
         FROM   p AS p1
```

[9] We actually ask only for those suppliers who supply at least one part, i.e., those *s#* values in *s*, where there exists a tuple in *sp* with that *s#* value (see Section 2.3.4).

```
WHERE  p.color = 'blue' AND
       NOT EXISTS (
         SELECT *
         FROM   sp AS sp2
         WHERE  sp2.s# = sp1.s# AND
                sp2.p# = p1.p#))
```

A direct translation of this query asks for each supplier such that there is no blue part that is not supplied by the supplier. Note that SQL operates on tables, i.e., multi-sets/bags of rows as opposed to sets of tuples in the relational model, as mentioned in Section 2.2. We use the keyword DISTINCT in the outermost SELECT clause to remove duplicates from the result. Otherwise, we would get the *s#* value as many times as it occurs in *sp*, including the case of duplicate dividend rows.

This approach is not very intuitive to formulate. Another way to express division queries has been proposed in the past, using a special syntax for universal quantification. The quantifier "FOR ALL," which is part of a so-called *quantified predicate* [GP99], was planned to be included in the SQL:1999 standard [ISO02] but it was finally excluded for reasons unknown to the author. We could phrase queries using the quantifier for division queries in an intuitive way. For example, the following SQL query employing a quantified predicate is equivalent to the above query:

```
SELECT DISTINCT s#
FROM   sp AS sp1
WHERE  FOR ALL (
         SELECT *
         FROM   p AS p1
         WHERE  color = 'blue')
       (EXISTS (
         SELECT *
         FROM   sp AS sp2
         WHERE  sp2.s# = sp1.s# AND
                sp2.p# = p1.p#))
```

There is a third way mentioned in the literature that uses aggregation [GC95]. Our example query can be phrased in SQL using aggregation as follows:

```
SELECT   s#
FROM     sp
GROUP BY s#
HAVING   COUNT(DISTINCT p#) = (
           SELECT COUNT(DISTINCT p#)
           FROM   p
           WHERE  color = 'blue')
```

There is a problem with this approach because it is not equivalent to the previous two approaches. It returns the same result as the other queries only if two conditions are met. First, each *p#* (*B*) value in *sp* (the dividend) is also contained in the *p* table (the divisor). Defining a foreign key *sp.p#* that references *p* and enforcing referential integrity can fulfill this condition. Another way to guarantee referential integrity is to preprocess the dividend by a semi-join of dividend and divisor. The semi-join returns only those dividend rows whose *B* values are contained in the divisor.

The second condition of this approach requires that the *p#* (*B*) values and the divisor rows are unique. Possible duplicates have to be removed before the division. Hence, the SQL query above contains the SQL keyword "DISTINCT" when counting *p#* values to avoid any duplicates. Note that when the divisor is grouped on all of its attributes, each group consists of a single row because of the required absence of duplicate rows. The same is true for the dividend if it is grouped on both *A* and *B*.

We have seen that the two approaches (based on existence and based on aggregation) actually realize two logical operators that give rise to two classes of algorithms, aggregate and scalar. The *scalar* class of algorithms relies on direct row matches between the dividend's divisor attributes *B* and the divisor table. The second class, *aggregate* algorithms, use counters to compare the number of rows in a dividend's quotient group to the number of divisor rows. In [Bry89], scalar and aggregate algorithms are called *direct* and *indirect* algorithms, respectively.

Aggregate algorithms are often described as alternative ways to scalar algorithms (for the real division operator) but they are prone to errors because one has to take care of duplicates, NULL values, and referential integrity, as mentioned before.

Some query languages for non-relational data models also offer support to express quantification. For example, there is a working draft specification of *XQuery* [W3C03], a query language for XML data, where universal quantification can be expressed by a so-called *every expression*. For example, the following query solves our division problem in an XML database:

```
for $sp1 in doc("supplier-parts.xml")//sp
where
  every $p in //p[color="blue"] satisfies
    some $sp2 in //sp satisfies
      $sp2/p# = $p/p# and
      $sp2/s# = $sp1/s#
return $sp1/s#
```

A query language syntax dedicated to universal quantification allows us to map the query directly to a query execution that uses a division algorithm. It is, however, nontrivial to map a query formulated in an indirect way (e.g. by using nested negations as in the first approach) to a query execution that uses a division algorithm.

2.5.2 Set Containment Division

We have devised a straightforward hypothetical syntax for the set containment division operator in SQL. We illustrate the syntax using an example. The following query delivers for each color the suppliers who supply all parts with that color:[10]

```
SELECT s#, color
FROM   (
         SELECT DISTINCT s#, p#
         FROM   sp
       ) AS sp1
       GREAT DIVIDE BY (
         SELECT DISTINCT p#, color
```

[10] see footnote 9 on page 24

```
    FROM   p
) AS p1
ON (sp1.p# = p1.p#)
```

The syntax has the same structure as the join in the following query, which returns for each color the suppliers who supply at least one part with that color:

```
SELECT s#, color
FROM   sp INNER JOIN p
       ON (sp.p# = p.p#)
```

According to the SQL standard ([ISO02], Section 7.7, "Joined Table"), the latter query includes a *join condition* introduced by the keyword ON. The INNER JOIN clause is a *qualified join*, where the keyword INNER denotes the *join type*. Analogously, we use a "qualified division," where the keyword GREAT enables the set containment division. If we use the keyword SMALL instead of GREAT, we enable the classical division. The terms GREAT and SMALL in combination with the division operator originate from the article [DD92], as mentioned in Section 2.3.2.

Concerning the power of the SQL syntax, one could allow a more general join condition than equality between columns in the ON clause like

```
ON (INTEGER(SUBSTR(sp1.p#, 2, 10))
    BETWEEN 3 AND CEILING(SQRT(INTEGER(SUBSTR(p1.p#, 2, 10)))))
```

However, the result of such a query would have a semantics that is completely different from division.

2.6 Applications and Beyond

Set containment join is considered an important operator for queries involving set-valued attributes [Hel00, HM02, Mam03, MGM03, MGM02a, Ram02, RPNK00, ZND+01]. For example, set containment test operations have been used for optimizing a workload of continuous queries, in particular for checking if one query is a subquery of another. In particular, Chen and DeWitt [CD02] suggested an algorithm that re-groups continuous queries to maintain a close-to-optimal global query execution plan. Each group consists of a set of queries that share some common join expression. Among the data structures used for the algorithm are query strings. A query string is an ASCII text string representation of an AND-OR DAG of a query. The query string is mapped to a signature using a hash-function. The signatures are used during a set containment test between queries to check if one query is a subquery of another.

Another example of set containment joins is content-based retrieval (using a "search engine") in document databases, where a huge set of documents is tested against a set of keywords that all have to appear in the document.

Frequent itemset discovery, one of the most important data mining operations, is an excellent example of a set containment test problem: Given a set of transactions T and a single (candidate) itemset i, how many transactions $t \in T$ fulfill the condition $i \subseteq t$? If this number is beyond the minimum support threshold, the itemset is considered frequent. In general, we would like to test a whole set of itemsets I for containment in T:

Find the number of tuples in $I \bowtie_{\subseteq} T$ for each distinct transaction in T.

In this work, we argue that if SQL would allow expressing set containment join and set containment division problems in an intuitive manner and if several algorithms implementing these operators were available in a RDBMS, this would greatly facilitate the processing of queries for frequent itemset discovery.

The relational division operator has been studied in the context of *fuzzy relations*, e.g., [BP82]. In a fuzzy relation, the tuples are weighted by a number between 0 and 1. One interpretation of an extended division operator for fuzzy relations, the *fuzzy quotient operator* [Yag91], is based on one of several relaxed versions of the universal quantifier, called "almost all," which is realized by a so-called *ordered weighted average operator*. The fuzzy quotient operator produces those values of $a \in \pi_A(r_1)$, where for "almost all" elements $b \in \pi_B(r_2)$ the tuple $((a) \times (b))$ is in r_1 for some fuzzy relations r_1 and r_2 with schemas $R_1(A \cup B)$ and $R_2(B)$, respectively. Other interpretations of a "fuzzy" version for division are discussed, e.g., in [BDPP97, Bos97]. We do not further study the division operator for fuzzy relations in this work.

2.7 Summary

Division is an operator that represents the universal quantifier in the relational model. We gave an overview of the multitude of definitions of the division operator in the literature. Then, we introduced a generalization of division called set containment division or great divide or generalized division. The three names have been suggested independently in the literature. Next, we illustrated the close relationship between set containment division and set containment join, which requires set-valued join attributes. Division is a derived operator but there are efficient division algorithms, i.e., physical operators, that do not rely on basic algebra operators. Hence, we presented important algebraic laws involving the division operator to enable transformations of a query execution plan during query optimization. Finally, we mentioned applications where queries occur that can benefit from an efficient implementation of the division operator. The algorithms that underly such implementations are the topic of the following chapter.

"Relational division tends to be one of the least well understood operators of the relational algebra."

H. Darwen, C. Date [DD92]

3

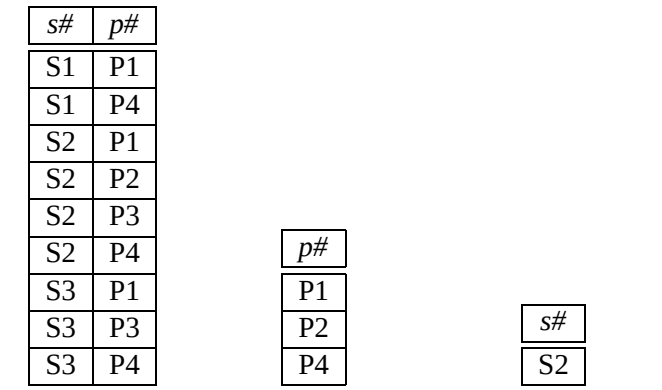
Algorithms for Relational Division

In the previous chapter, we have discussed the *logical* behavior of relational division and of two operators related to it, set containment division and set containment join. In this chapter, we focus on *physical* division operators by presenting a comprehensive survey of the structure and performance of division algorithms and by introducing a framework that results in a complete classification of input data for division [RSMW02]. Then we go on to identify the most efficient algorithm for each such class. One of the input data classes has not been covered so far. For this class, we propose several new algorithms. Thus, for the first time, we are able to identify the optimal algorithm to use for any given input dataset.

These two classifications of optimal algorithms and input data are important for query optimization. They allow a query optimizer to make the best selection when optimizing at intermediate steps for the quantification problem and they complement the algebraic laws involving division that we discussed in Section 2.4.

3.1 Introduction

Several algorithms have been proposed to implement relational division efficiently. They are presented in an isolated manner in the research literature—typically, no relationships are shown between them. Furthermore, each of these algorithms claims to be superior to others, but in fact



s#	*p#*
S1	P1
S1	P4
S2	P1
S2	P2
S2	P3
S2	P4
S3	P1
S3	P3
S3	P4

(a) *sp* (dividend)

p#
P1
P2
P4

(b) *p* (divisor)

s#
S2

(c) *result* (quotient)

Figure 3.1: $sp \div p = result$, representing the query "Which suppliers supply all parts?"

each algorithm has optimal performance only for certain types of input data.

To illustrate the division operator we will use a simple example throughout the chapter, illustrated in Figure 3.1, representing data from a company's supplier-parts database [DD92, Dat94] like the one we used in Section 2.3.4. A *p* row represents a part that is required for a certain project and an *sp* row indicates which supplier can supply which part. The following query can be expressed by the division operator:[1]

"Which suppliers supply *all* parts required by the project?"

As indicated in the table *result*, only S2 supplies all parts. S2 supplies also part P3 but this does not affect the result. Both supplier S1 and S3 do not supply part P2. Therefore, they are not included in the result.

Remember from Section 2.3.1 that the division operator takes two tables for its input, the *dividend* and the *divisor*, and generates one table, the *quotient*. All the data elements in the divisor must appear in the dividend, paired with any element (such as S2) that is to appear in the quotient.

In the example of Figure 3.1, the divisor and quotient have only one column each, but in general, they may have an arbitrary number of columns. In any case, the set of attributes of the dividend is the disjoint union of the attributes of the divisor and the quotient. To simplify our exposition, we assume that the names of the dividend attributes are the same as the corresponding attribute names in the divisor and the quotient.

The remainder of this chapter is organized as follows. In Section 3.2, we present a classification of input data for algorithms that evaluate division within queries. Section 3.3 gives an overview of known and new algorithms to solve the universal quantification problem and classifies them according to two general approaches for division. In Section 3.4, we evaluate the algorithms according to both applicability and effectiveness for different kinds of input data, based on a performance analysis. We discuss related work in Section 3.5 and summarize this chapter in Section 3.6.

[1] see footnote 9 on page 24

3.2 Classification of Data

This section presents an overview of the input data for division. We identify all possible classes of data based on whether it is grouped on certain columns. For some of these classes, we will present efficient algorithms in Section 3.3 that exploit the specific data properties of a class.

3.2.1 Input Data Characteristics

The goal of this chapter is to identify optimal algorithms for the division operator, for all possible inputs. Several papers compare new algorithms to previous algorithms and claim superiority for one or more algorithms, but they do not address the issue of which algorithms are optimal for which types of data [Bry89, CKMP97, GC95]. In fact, the performance of any algorithm depends on the structure of its input data. If we know about the structure of input data, we could employ an algorithm that exploits this structure, i.e., the algorithm does not have to restructure the input before it can start generating output data. Of course, there is no guarantee that such an algorithm is always "better" than an algorithm that requires previous restructuring. However, the division operator offers a variety of alternative algorithms that can exploit such a structure for the sake of good performance and low memory consumption.

Suppose we are fortunate and the input data is highly structured. For example, suppose the data has the schema of Figure 3.1 but is of much larger size, and suppose:

- *sp* is sorted by *s#* and *p#* and resides on disk, and
- *p* is sorted by *p#* and resides in memory.

Then the example query can be executed with one scan of the *sp* table. This is accomplished by reading the *sp* table from disk. As each supplier appears, the *p#* values associated with that supplier are merged with the *p* table. If all parts match, the *s#* value is copied to the result.

The single scan of the *sp* table is obviously the most efficient possible algorithm in this case. In the remainder of this chapter, we will describe similar types of structure for input datasets, and the optimal algorithms that are associated with them. The notion of "optimality" will be further discussed in the next section.

Revisiting our example in Figure 3.1, how could this careful structuring of input data, such as sorting by *s#* and *p#*, occur? It could happen by chance, or for two other more commonly encountered reasons:

1. The data might be stored in tables in a certain order. In an RDBMS, one can define that some columns form the primary key of a table. This is usually realized by a clustered index like a B-tree [BM72]. The indexed rows are stored on disk in the order of the primary key column values.

2. The data might have undergone some previous processing, because the division operator query is part of a more complex query. The previous processing might have been a merge-join operator, for example, which requires that its inputs be sorted and produces sorted output data.

3.2.2 Choice of Algorithms

A query processor of a database system typically provides several algorithms that all realize the same operation. An optimizer has to choose one of these algorithms to process the given data. If the optimizer knows the structure of the input data for an operator, it can pick an algorithm that exploits the structure. Many criteria influence the decision why one algorithm is preferred over others. Some of these choice criteria are: the time to deliver the first/last result row, the amount of memory for internal, temporary data structures, the number of scans over the input data, or the ability to be non-blocking, i.e., to return some result rows before the entire input data are consumed.

Which algorithm should we use to process the division operation, given the dividend and divisor tables shown in Figure 3.1? Several algorithms are applicable but they are not equally efficient. For example, since the dividend and divisor are both sorted on the column *p#* in Figure 3.1, we could select a division algorithm that exploits this fact by processing the input tuples in a way that is similar to the merge-join algorithm, as we have sketched in the previous section.

What algorithm should we select when the input tables are *not* sorted on *p#* for each group of *s#*? One option is to sort both input tables first and then employ the algorithm similar to merge-join. Of course, this incurs an additional computational cost for sorting in addition to the cost of the division algorithm itself. Another option is to employ an algorithm that is insensitive to the ordering of input tuples. One such well-known algorithm is hash-division and is discussed in detail in Section 3.3.2.4.

We have seen that the decision which algorithm to select among a set of different division algorithms depends on the structure of the input data. This situation is true for any class of algorithms, including those that implement database operators like join, aggregation, and sorting.

It is possible that division is only a portion of a larger query that contains many additional query parts. Hence, the input of a division operation is not restricted to base tables, like in the example of Figure 3.1, but it can be derived tables which are the result of another operation like a join, for example. Furthermore, the output of the division could be an intermediate result itself that is further processed within the query. For example, the quotient table *result* in Figure 3.1 could be the input of an aggregation that counts the number of suppliers. The meaning of the resulting aggregate is the number of suppliers who supply all parts of the project. Alternatively, the result in Figure 3.1 could be an input of a join with a table $s(s\#, name, address, \dots)$ to retrieve a supplier's name, address, etc. instead of a meaningless ID. Thus, the result table produced by the selected division algorithm can have certain data properties that influence the choice of additional algorithms, here a join, that are used to process the overall query.

3.2.3 Grouping

Relational database systems have the notion of grouped rows in a table. Let us briefly look at an example that shows why grouping is important for query processing. Suppose we want to find for each part the number of suppliers in the *sp* table of Figure 3.1. One way to compute the aggregates involves *grouping*: after the table has been grouped on *p#*, all rows of the table with the same value of *p#* appear next to each other. The ordering of the group values is not specified,

i.e., any group of rows may follow any other group. *Group-based aggregation* groups the data first, and then it scans the resulting table once and computes the aggregates during the scan.

Another way to process this query is *nested-loop aggregation*. We pick any part number as the first group value and then search through the whole table to find the rows that match this value and compute the sum. Then, we pick a second p# value, search for matching rows, compute the second aggregate, pick the third value, etc. If no suitable search data structure (index) is available, this processing may involve multiple scans over the entire dataset.

The aggregation step of the group-based approach is obviously more efficient than the second approach because it can make an assumption about some ordering of the rows. However, the more efficient processing is paid with the overhead of the preceding grouping.

When a table is to be grouped on a list $(a_1,\ldots,a_n)$ of more than one column, the result is equal to grouping on a single column in an iterative way: We first group on a_1, then for each subset of rows defined by a_1, we group on a_2, and for each such subset determined by a_2, we group on a_3, etc. Hence, if we want to compare two tables that are grouped on the same set of columns, we have to be aware of the *column list ordering*, because the resulting grouped table has a different structure for each ordering. This fact is important for division when we match some of the dividend's divisor columns with all of the divisor's columns.

Sorted data appears frequently in query processing. Note that sorting is a special grouping operation. For example, grouping only requires that suppliers supplying the same part are stored next to each other (in any order), whereas sorting requires more effort, namely that they be in a particular order (ascending or descending). The overhead of sort-based grouping is reflected by the time complexity $O(n\log n)$ as opposed to the nearly linear time complexity for hash-based grouping. Hash-based grouping inserts all rows into a hash table and then scans the hash table. All rows with of the same group are hashed to the same hash-bucket chain. Sometimes more than one group are hashed to the same chain, which causes a "slightly" more than linear complexity. Though sort-based grouping algorithms do more than necessary, both hash-based and sort-based grouping perform well for large datasets [Gra93, GC95].

3.2.4 Grouped Input Data for Division

Relational division has two input tables, a dividend and a divisor, and it returns a quotient table. According to the definition of the division operator in Section 2.3.1, we can partition the attributes of the dividend r_1 into two sets, which we denote A and B, where B corresponds to the attributes of the divisor and A are the quotient attributes. As already mentioned, for simplicity, we assume that the names of attributes in the quotient r_3 are the same as the corresponding attribute names in the dividend r_1 and the divisor r_2. Thus, we write a division operation as $r_3(A) = r_1(A \cup B) \div r_2(B)$. In Figure 3.1, $A = \{s\#\}$ and $B = \{p\#\}$.

Our classification of division algorithms is based on whether certain columns are grouped or even sorted. Several reasons justify this decision. Grouped input can reduce the amount of memory needed by an algorithm to temporarily store rows of a table because all rows of a group have a constant group value. Furthermore, grouping appears frequently in query processing. Many database operators require grouped or sorted input data (e.g., merge-join) or produce such output data (e.g., index-scan): If there is an index defined on a base table, a query processor

Class	Dividend r_1		Divisor r_2	Description of grouping
	A	B	B	
0	$-$	$-$	$-$	
1	$-$	$-$	$+$	
2	$-$	$+$	$-$	
3	$-$	$+$	$+$	arbitrary ordering of groups in $r_1.B$ and $r_2.B$
4	$-$	$+^*$	$+^*$	same ordering of groups in $r_1.B$ and $r_2.B$
5	$+$	$-$	$-$	
6	$+$	$-$	$+$	
7	$+_1$	$+_2$	$-$	A major, $r_1.B$ minor
8	$+_2$	$+_1$	$-$	$r_1.B$ major, A minor
9	$+_1$	$+_2$	$+$	A major, $r_1.B$ minor; arbitrary ordering of groups in $r_1.B$ and $r_2.B$
10	$+_1$	$+_2^*$	$+^*$	A major, $r_1.B$ minor; same ordering of groups in $r_1.B$ and $r_2.B$
11	$+_2$	$+_1$	$+$	$r_1.B$ major, A minor; arbitrary ordering of groups in $r_1.B$ and $r_2.B$
12	$+_2$	$+_1^*$	$+^*$	$r_1.B$ major, A minor; same ordering of groups in $r_1.B$ and $r_2.B$

Table 3.1: A classification of dividend and divisor

can retrieve the rows in sorted order, specified by the index column list. Thus, algorithms may exploit for the sake of efficiency the fact that base tables or derived tables are grouped if the system *knows* about this fact.

In Table 3.1, we show all possible classes of input data based on whether or not interesting column sets are grouped, i.e., grouped on all columns of A and/or all columns of B. As we will see later in this chapter, some classes have no suitable algorithm that can exploit its specific combination of data properties. The classes that have at least one algorithm exploiting exactly its data properties are shown in bold font. In class 0, for example, no table is grouped on an interesting column set. Algorithms for this class have to be insensitive to whether the data is grouped or not. Another example scenario is class 10. Here, the dividend is *first* grouped on the quotient columns A (denoted by $+_1$, the major group) and for each group, it is grouped on the divisor columns B (denoted by $+_2$, the minor group). The divisor is grouped in the same ordering (denoted by a superscript asterisk $+^*$) as the dividend.

Our classification is based on grouping only. As we have seen, some algorithms may require that the input is even sorted and not merely grouped. We consider this a minor special case of our classification, so we do not reflect this data property in Table 3.1, but the algorithms in Section 3.3 will refer to this distinction. We do not consider any data property other than grouping in this chapter because our approach is complete and can easily and effectively be exploited by a query optimizer and query processor. A further data property is, e.g., whether the data is partitioned across tables and how it is partitioned (vertically or horizontally). Partitioning is important when we want to parallelize the query execution.

In Table 3.1, columns are either grouped ($+$) or not grouped ($-$). We use the same (a different) subscript of $+$ when $r_1.B$ and the divisor have the same (a different) ordering of groups in

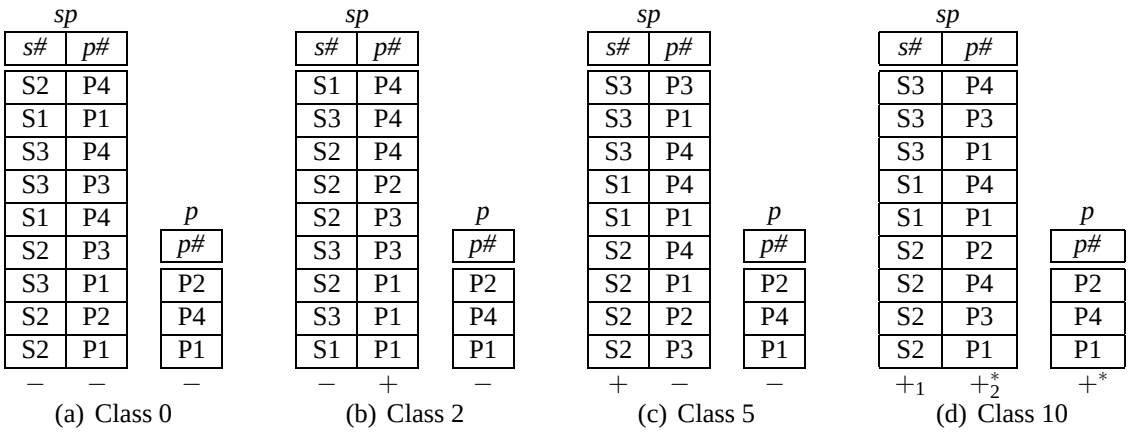
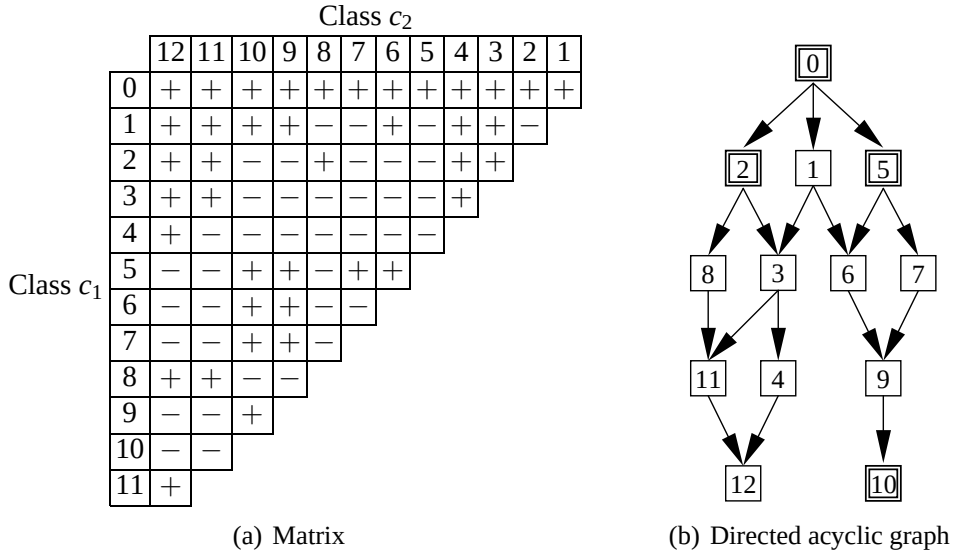


Figure 3.2: Four important classes of input data, based on the example of Figure 3.1



(a) Matrix (b) Directed acyclic graph

Figure 3.3: A matrix and a DAG representing the input data classification described in Table 3.1

classes 3, 4, 9–12. In addition, when the dividend is grouped on both A and $r_1.B$ in classes 7–12, then $+_1$ ($+_2$) denotes the columns that the table is grouped on first (second).

Figure 3.2 illustrates four classes of input data for division, based on the example data of Figure 3.1. These classes, which are shown in bold font in Table 3.1, are important for several algorithms that we present in the following section. The tables are either grouped ($+$) or not grouped ($-$) on the respective column. Notice that for class 10 both tables are grouped in the same order on *p#*. If the value P3 is present in a quotient group then it always appears after P4 and before P1. Figure 3.1 shows another example instance of class 10, where the quotient order as well as the divisor group order is ascending. The benefit of knowing about such an input data property will be clarified when we discuss algorithms exploiting this specific property in Sections 3.3.2.2 and 3.3.2.3.

If we know that an algorithm can process data of a specific class, it is useful to know which other classes are also covered by the algorithm. This information can be represented, e.g., by a Boolean matrix like the one on the left in Figure 3.3. One axis indicates a given class c_1 and the other axis shows the other classes c_2 that are also covered by c_1. Alternatively, we can use a directed acyclic graph representing the input data classification, sketched on the right of Figure 3.3. If a cell of the matrix is marked with a plus sign (+), or equivalently, if there is a path in the graph from class c_1 to c_2, then an algorithm that can process data of class c_1 can also process data of class c_2. The graph clearly shows that the classification is a partial order of classes, not a strict hierarchy. The source node of the graph is class 0, which requires no grouping of columns *A* or *B*. Any algorithm that can process data of class 0 can process data of any other class. For example, an algorithm processing data of class 6 is able to process data of classes 9 and 10. All algorithms to be discussed in Section 3.3 assume data properties of either class 0, 2, 5, or 10.

For the subsequent discussion of division algorithms, we define two terms to refer to certain row subsets of the dividend. Let the dividend r_1 be grouped on A (B) as the first or the only set of group columns, i.e., let the dividend belong to class 5 (2) and all its descendants in Figure 3.3. Furthermore, let v be one specific value of such a group. Then, the set of rows defined by $\sigma_{A=v}(r_1)$ ($\sigma_{D=v}(r_1)$) is called the *quotient group* (*divisor group*) of v. For example, in the *sp* table of class 5 in Figure 3.2(c), the quotient group of S1 consists of the rows {(S1, P4), (S1, P1)}. Similarly, the divisor group of P2 in class 2 in Figure 3.2(b) consists of the single row (S2, P2).

3.3 Overview of Algorithms

In this section, we present algorithms for relational division proposed in the database literature together with several new variations of the well-known hash-division algorithm. For the sake of a concise presentation, we will frequently use abbreviations for the algorithms that we summarize in Table 3.2.

In Section 3.4, we will analyze and compare the effectiveness of each algorithm with respect to the data classification of Section 3.2.

3.3.1 Complexity of Algorithms

During the evaluation of relevant literature, we found that it is necessary to clarify that each division algorithm (analogous to other classes of algorithms, like joins, for example) has performance advantages for certain data characteristics. No algorithm is able to outperform the others for every input data conceivable.

The following algorithms assume that the division's input consists of a dividend table $r_1(A \cup B)$ and a divisor table $r_2(B)$, where A is a set of quotient columns and B is the set of divisor columns, as defined in Section 3.2.4.

During the presentation of the algorithms, we analyze the worst and typical case complexities of processing time and memory consumption in *O*-notation, based on the size (number of rows)

Division algorithm	Abbreviation
Hash-division	HD
Hash-division for divisor groups	HDD
Hash-division for quotient groups	HDQ
Merge-count division	MCD
Merge-group division	MGD
Merge-sort division	MSD
Nested-loop division	NLD
Nested-loop counting division	NLCD
Transposed hash-division	HDT
Transposed hash-division for divisor groups	HDTD
Transposed hash-division for quotient groups	HDTQ
Stream-join division	SJD

Table 3.2: Abbreviations for division algorithms

of the dividend $|r_1|$ and the size of the divisor $|r_2|$. We use $|A|$, the number of distinct values of quotient columns A in the dividend, for some algorithms to derive a complexity formula. Note that always $|A| \leq |r_1|$, and in the *worst* case $|A| = |r_1|$, i.e., each single row of r_1 is a potential (candidate) quotient. To derive formulas for the *typical* time and memory complexities, we use the assumption that $|r_1| \gg |r_2|$, i.e., there are many quotient candidates and/or the number of rows of a typical quotient candidate is much larger than the number of divisor rows. We consider this situation as the typical case because relational division is defined to compute a *set* of result rows and in real-life scenarios this set is of considerable size. A large result size occurs only if the dividend contains many more rows than the divisor.

In addition to time and memory complexity, it is useful to analyze the I/O cost of each algorithm, as it has been done in detail for some of the following algorithms in [GC95]. However, since the focus of this chapter is to describe the fundamental structure of input data and algorithms involved in relational division, we restrict our analysis to memory and processing complexities and we do not give I/O formulas.

3.3.2 Scalar Algorithms

This section presents division algorithms that use data structures to directly match dividend rows with divisor rows.

3.3.2.1 Nested-Loop Division

This algorithm is the most naïve way to implement division. However, like nested-loop join, an operator using *nested-loop division (NLD)* has no required data properties on the input tables and thus can always be employed, i.e., NLD can process input data of class 0 and thus any other class of data, according to Figure 3.3.

We use two set data structures, one to store the set of divisor values of the divisor table, called *seen_divisors*, and another to store the set of quotient candidate values that we have found so far in the dividend table, called *seen_quotients*. We first scan the divisor table to fill *seen_divisors*. After that, we scan the dividend in an outer loop. For each dividend row, we check if its quotient value (A) is already contained in *seen_quotients*. If not, we append it to the *seen_quotients* data structure and scan the remainder of the dividend iteratively in an inner loop to find all rows that have the same quotient value as the dividend row of the outer loop. For each such row found, we check if its divisor value is in *seen_divisors*. If yes, we mark the divisor value in *seen_divisors*. After the inner scan is complete, we add the current quotient value to the output if all divisors in *seen_divisors* are marked. Before we start processing the next dividend row of the outer loop, we unmark all elements of *seen_divisors*.

Note that NLD can be very inefficient. For each row in the dividend table, we scan the dividend at least partially to find all the rows that belong to the current quotient candidate. All divisor rows and quotient candidate rows are stored in an in-memory data structure. NLD can be an efficient algorithm for small ungrouped datasets when the overhead of sorting or hash-based grouping, which involves allocating and filling a hash table, is too high.

This algorithm can make use of any set data structure like hash tables or sorted lists to represent *seen_divisors* and *seen_quotients*. Let us assume that this algorithm uses hash tables or any very efficient data structure with a (nearly) constant access time. Then, the worst case time complexity of this algorithm is $O(|r_1|^2 + |r_2|)$ and the typical time complexity is $O(|r_1|^2)$. The memory complexity is $O(|A| + |r_2|)$. Since in the extreme case $|A| = |r_1|$, the worst case memory complexity is $O(|r_1| + |r_2|)$ and the typical memory complexity is $O(|r_1|)$.

The pseudo code of the nested-loop division algorithm is shown in Appendix B on page 172. There, the *seen_divisors* and *seen_quotients* data structures are represented by the divisor hash table *dht* and the quotient hash table *qht*, respectively.

Figure 3.4(a) illustrates the two hash tables used in this algorithm: the divisor/quotient hash table represents *seen_divisors*/*seen_quotients*, respectively. The value setting in the hash tables is shown for the time when all dividend rows of S1 and S2 (in this order) have been processed and we have not yet started to process any rows of S3 in the outer loop. We find that S2 is a quotient because all bits in the divisor hash table are equal to 1.

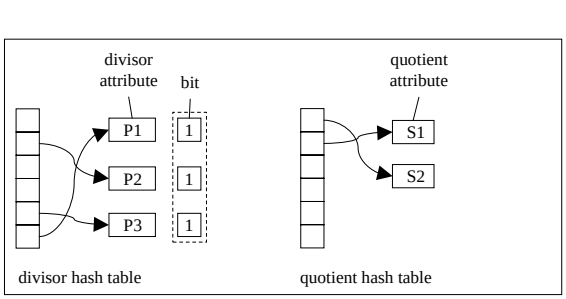
The value setting of all illustrations in Figure 3.4 are based on the example from Figure 3.1. Except for MSD and MGD, broken lined boxes indicate that a quotient is found.

3.3.2.2 Merge-Sort Division

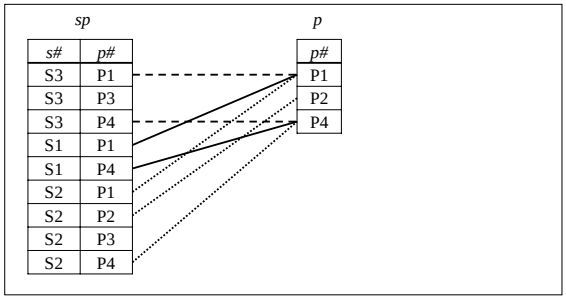
The *merge-sort division (MSD)* algorithm assumes that

- the divisor r_2 is sorted, and that
- the dividend r_1 is grouped on A, and for each group, it is sorted on B in the same order (ascending or descending) as r_2.

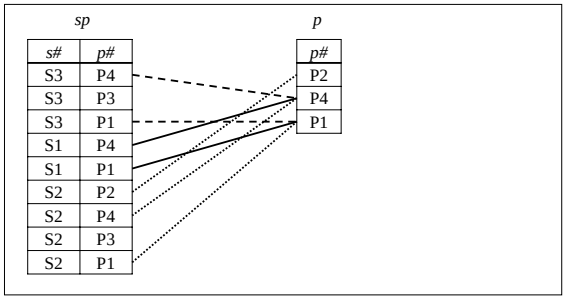
This data characteristic is a special case of class 10, where B and the divisor are sorted and not only grouped.



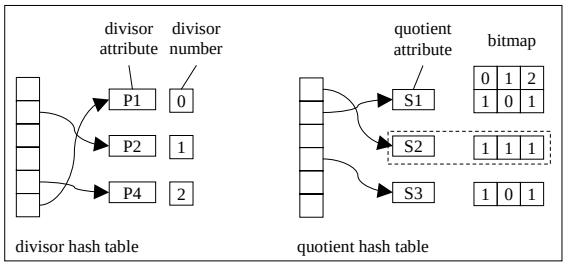
(a) Nested-loop division (NLD)



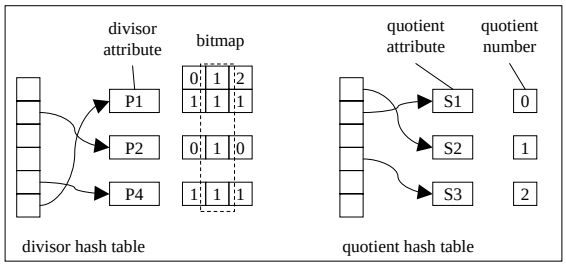
(b) Merge-sort division (MSD)



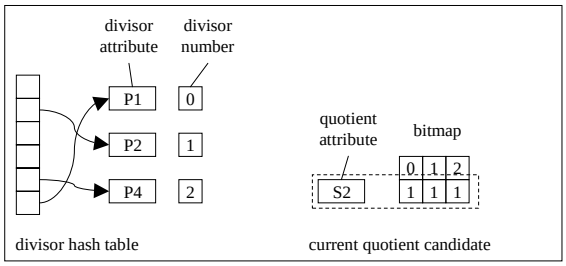
(c) Merge-group division (MGD)



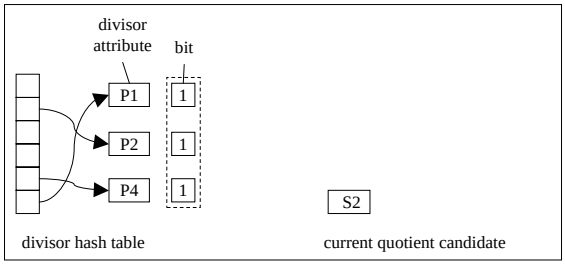
(d) Hash-division (HD)



(e) Transposed hash-division (HDT)



(f) Hash-division for quotient groups (HDQ)



(g) Transposed hash-division for divisor groups (HDTQ)

Figure 3.4: Overview of data structures and processing used in scalar algorithms

The algorithm resembles merge-join when only a single quotient group is processed and it is similar to nested-loop join for processing all groups. Let us briefly sketch the processing of rows within a single group, assuming an ascending sort order. We begin with the first row of dividend and divisor. Let b be the of the columns B. If the divisor value b of the current dividend row and the divisor row match, we proceed with the next row in both tables. If b is greater than the

current divisor row, we scan forward to the next quotient group. If b is less than the divisor row, we proceed with the next row of the group and the current divisor row. If there are no more rows to process in the quotient group but at least one more row in the divisor, we skip the quotient group. If there are no more rows to process in the divisor, we have found a quotient and add it to the output table.

Our merge-sort division is similar to the approach called *naïve division*, presented in [Gra89, GC95] and originating from [SC75]. In both approaches, we can implement the scan of each input such that it ignores duplicates. In contrast to merge-sort division, naïve division explicitly sorts the data before the merge step. Even worse, naïve division does not merely group the dividend on A but sorts it, which is more than necessary. Note that we view sorting or grouping as preprocessing activities that are separate from the core division algorithm. We sketch the pseudo code of merge-sort division without duplicate removal logic in Appendix B on page 175.

The worst case time complexity of this algorithm is $O(|r_1| + |A||r_2|) = O(|r_1| + |r_1||r_2|) = O(|r_1||r_2|)$ because the dividend is scanned exactly once and from the divisor table, we fetch as many rows as the number of quotient candidates times the number of divisor rows. The typical case time complexity is $O(|r_1||r_2|)$. The worst and typical case memory complexity is $O(1)$, since only a constant number of small data structures (two rows) have to be kept in memory.

Figure 3.4(b) illustrates the matches between rows of dividend and divisor. Observe that the data is not sorted but only grouped on *s#* in an arbitrary order.

3.3.2.3 Merge-Group Division

We can generalize merge-sort division to an algorithm that we call *merge-group division (MGD)*. In contrast to MSD, we assume that

- both inputs are only grouped and not necessarily sorted on the divisor columns, but that
- the order of groups in each quotient group is the same as the order of groups in the divisor.

Note that each group within a quotient group and within the divisor consists of a single row. This ordering can occur (or can be achieved) if, e.g., the same hash function is used for grouping the divisor and each quotient group.

In the MSD algorithm, we can safely skip a quotient candidate if the current value of A is greater (less) than that of the current divisor row, assuming an ascending (a descending) sort order. Since we do not require a sort order on these columns in MGD, we cannot skip a group on unequal values, as we do in MSD. For example, suppose that the divisor r_2 has a single integer column and consists of the following rows in the given order: $r_1 = (3,1,5)$ and the B values of the current quotient group G consists of the rows $G = (2,5,4,6)$. We can be sure that G is *not* added to the quotient only after

- we have scanned the entire group G, where we find that the first element of r_1 (3) is not contained in G, or
- we have scanned r_1 up to last element (5) and we have scanned G up to the second element (5) to find that G does not contain the other elements of r_1 (3 and 1) before element 5 appears.

The MGD approach makes use of a look-ahead of n divisor rows for some predefined value $n \geq 1$. As in the MSD approach, we compare the current quotient group row with the current divisor row. In case of inequality, we look ahead up to the nth divisor row to see if there is any other row matching the current group row. If we find such a match, we can skip the current quotient candidate. In our example, a look-ahead of 2 means that we check up to the second element (1) of the divisor. The look-ahead of 2 does not help for any value of G in our example. A look-ahead of 3 means a check with up to the third divisor element (5). When we check the second row (5) of the quotient group, we find a match with the third divisor element (5). Here, we can skip the group because a quotient would have to contain the values 3 and 1 before the occurrence of 5 to qualify due to the assumption that the group orders are the same. In other words, the ordering assumption guarantees that the values 3 and 1 cannot occur after the element 5. Since they have neither occurred in G before element 5, we know that this quotient candidate does not contain all divisor elements, in particular not the elements 3 and 1.

The MSD algorithm is a special case of MGD where the look-ahead is set to one because it does not look further than the current row for each quotient group row since sorting was applied.

In summary, the MGD approach can make use of as much look-ahead as the minimum of the available memory and the current divisor size. Note that the divisor fits into memory in all reasonable cases. Figure 3.4(c) sketches the matches between dividend and divisor rows. Observe that the order of (single-row) groups within each quotient group in the dividend is the same as that of the divisor.

The time complexity of this algorithm is $O(|r_1| + |A||r_2|)$ because the dividend is scanned exactly once and the divisor is scanned entirely for each quotient and at least partially for every quotient candidate. Thus, the worst case time complexity is $O(|r_1| + |r_1||r_2|) = O(|r_1||r_2|)$. The typical case time complexity is also $O(|r_1||r_2|)$. The worst case memory complexity is $O(|r_2|)$ if we keep the entire divisor as a look-ahead in memory. The typical case memory complexity then becomes $O(1)$ since $|r_2| \ll |r_1|$.

3.3.2.4 Classic Hash-Division

In this section, we present the classic *hash-division (HD)* algorithm [GC95]. We call this algorithm "classic" to distinguish it from our variations of this approach in the following sections. Classic hash-division requires no input data properties (class 0).

The two central data structures of HD are the divisor and quotient hash tables, sketched in Figure 3.4(d). The divisor hash table stores divisor rows. Each such row has an integer value, called *divisor number*, stored together with it. The quotient hash table stores quotient candidates and has a bitmap stored together with each candidate, with one bit for each divisor. The pseudo code of hash-division is sketched in Appendix B on page 173.

In a first phase, hash-division builds the divisor hash table while scanning the divisor. The hash function takes the divisor columns as an argument and assigns a hash bucket to each divisor row. A divisor row is stored into the hash bucket only if it is not already contained in the bucket, thus eliminating duplicates in the divisor. When a divisor row is stored, we assign a unique divisor number to it by copying the value of a global counter. This counter is incremented for each stored divisor row and is initialized with zero. The divisor number is used as an index for

the bitmaps of the quotient hash table.

The second phase of the algorithm constructs the quotient hash table while scanning the dividend. For each dividend row, we first check if its B value is contained in the divisor hash table, using the same hash function as before. If yes, we look up the associated divisor number, otherwise we skip the dividend row. In addition to the look-up, we check if the quotient is already present in the quotient hash table. If yes, we update the bitmap associated with the matching quotient row by setting the bit to one whose position is equal to the divisor number we looked up. Otherwise, we insert a new quotient row into the quotient hash table together with a bitmap where all bits are initialized with zeroes and the appropriate bit is set to one, as described before. Since we insert only quotient candidates that are not already contained in the hash table, we avoid duplicate dividend rows.

The final phase of hash-division scans the quotient hash table's buckets and adds all quotient candidates to the output whose bitmaps contain only ones. In Figure 3.4(d), the contents of the hash tables are shown for the time when all dividend and divisor rows of Figure 3.1 have been processed. We see that since the bitmap of S2 contains no zeroes, S2 is the only quotient, indicated by a broken lined box.

Hash-division scans both dividend and divisor exactly once. Because hash tables are employed that have a nearly constant access time, this approach has a worst and typical case time complexity of $O(|r_1| + |r_2|)$ and $O(|r_1|)$, respectively. The memory complexity consists of $O(|r_2|)$ to store the divisor hash table plus $O(|A||r_1|)$ for the quotient hash table. The size of a bitmap is proportional to $|r_1|$. Since the worst case scenario implies that $|A| = |r_1|$, the total worst and typical case memory complexity is $O(|r_1||r_2|)$.

3.3.2.5 Transposed Hash-Division

This algorithm is a slight variation of classic hash-division. The idea is to switch the roles of the divisor and quotient hash tables. The *transposed hash-division (HDT)* algorithm keeps a bitmap together with each row in the divisor hash table instead of the quotient hash table, as in HD. Furthermore, HDT keeps an integer value with each row in the quotient hash table instead of the divisor hash table, as in the HD algorithm.

Same as the classic hash-division algorithm, HDT first builds the divisor hash table. However, we store a bitmap with each row of the *divisor*. A value of one at a certain bit position of a bitmap indicates which quotient candidate has the same values of B as the given divisor row.

In a second phase, also same as HD, the HDT algorithm scans the dividend table and builds a quotient hash table. For each dividend row, the B values are inserted into the divisor hash table as follows. If there is a matching quotient row stored in the quotient hash table, we look up its quotient number. Otherwise, we insert a new quotient row together with a new quotient number. Then, we update the divisor row's bitmap by setting the bit to one whose position is given by the quotient number.

The final phase makes use of a new, separate bitmap, whose size is the same as the bitmaps in the divisor hash table. All bits of the bitmap are initialized with zero. While scanning the divisor hash table, we apply a bit-wise AND operation between each bitmap of the hash table and the new bitmap. The resulting bit pattern of the new bitmap is used to identify the quotients. The

quotient numbers (bit positions) with a value of one are then used to look up the quotients using a *quotient vector* data structure that allows a fast mapping of a quotient number to a quotient candidate. The HDT pseudo code is shown in Appendix B on page 176.

Figures 3.4(d) and (e) contrast the different structure of hash tables in HD and HDT. The hash table contents is shown for the time when all *sp* rows of Figure 3.1 have been processed. While a quotient in the HD algorithm can be added to the output when the associated bitmap contains no zeroes, the HDT algorithm requires a match of the bit at the same position of all bitmaps in the divisor table and it requires in addition a look-up in the quotient hash table to find the associated quotient row.

The time and memory complexities of HDT are the same as those of classic hash-division.

3.3.2.6 Hash-Division for Quotient Groups

Both classic and transposed hash-division can be improved if the dividend is grouped on either B or A. However, our optimizations based on divisor groups lead to aggregate, not scalar algorithms. Hence, this section on scalar algorithms presents some optimizations for quotient groups. The optimizations of hash-division for divisor groups are presented in Section 3.3.3.3.

Let us first focus on classic hash-division. If the dividend is grouped on A, we do not need a quotient hash table. It suffices to keep a single bitmap to check if the current quotient candidate is actually a quotient. When all dividend rows of a quotient group have been processed and all bits of the bitmap are equal to one, the quotient row is added to the output. Otherwise, we reset all bits to zero, skip the current quotient row, and continue processing the next quotient candidate. Because of the group-by-group processing of the improved algorithm, we call this approach *hash-division for quotient groups (HDQ)*.

The HDQ algorithm is non-blocking because we return a quotient row to the output as soon as a group of (typically few) dividend rows has been processed. In contrast, the HD algorithm has a final output phase: the quotient rows are added to the result table after the entire dividend has been processed because hash-division does not assume a grouping on A. For example, the "first" and the "last" row of the dividend could belong to the same quotient candidate, hence the HD algorithm has to keep the state of the candidate quotient row as long as at least one bit of the candidate's bitmap is equal to zero. Note that it is possible to enhance HD such that it is not a "fully" blocking algorithm. If bitmaps are checked during the processing of the input, HD could detect some quotients that can be returned to the output before the entire dividend has been scanned. Of course, we would then have to make sure that no duplicate quotients are created, either by preprocessing or by referential integrity enforcements or by keeping the quotient value in the hash table until the end of the processing. We do not elaborate further on this variation of HD.

HDQ has the same worst and typical case time complexity as HD since we have to scan the dividend and divisor table exactly once. However, the worst case memory consumption, due to the small bitmap, is $O(|r_2|)$ and thus the typical memory complexity is $O(1)$.

3.3.2.7 Transposed Hash-Division for Quotient Groups

We have seen that the HDQ algorithm is a variation of the HD algorithm: if the dividend is grouped on *A*, we can do without a quotient hash table. Exactly the same idea can be applied to HDT yielding an algorithm that we call *transposed hash-division for quotient groups (HDTQ)*.

For grouped quotient columns, we can do without the quotient hash table and we do not keep long bitmaps in the divisor hash table but only a single bit per divisor. Before any group is processed, the bit of each divisor column is set to zero. For each group, we process the rows like in the HDT algorithm. After a group is processed, we add a quotient to the output if the bit of every divisor row is equal to one. Then, we reset all bits to zero and resume the dividend scan with the next group.

We do not show the pseudo code for the HDQ and HDTQ algorithms for brevity. However, we sketch their data structures in the Figures 3.4(f) and (g) for the time when the group of dividend rows containing the quotient candidate S2 have been processed.

The complexities for time and memory are the same those as for HDQ.

3.3.3 Aggregate Algorithms

Definition 7 in Section 2.3.1 is an indirect way to specify the division operator: If the number of divisor tuples is equal to the number of tuples belonging to a group defined by the columns *A*, add *A* to the quotient.

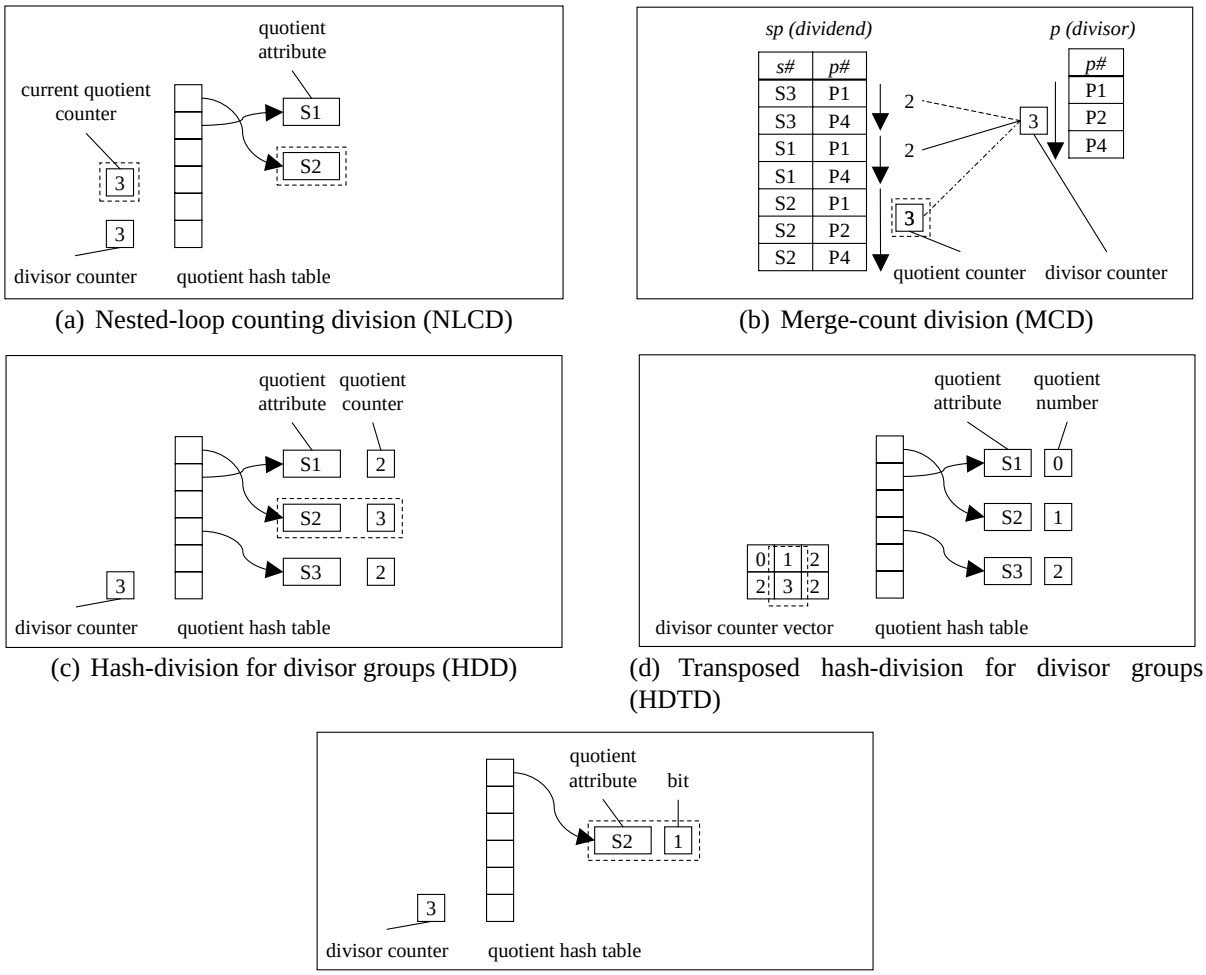
The class of algorithms discussed in this section have in common that in a first phase, the divisor table is scanned once to count the number of divisor rows. Each algorithm then uses different data structures to keep track of the number of rows in a quotient candidate. Some algorithms assume that the dividend is grouped on *A* or *B*.

Remember that aggregate algorithms require the referential integrity precondition $\pi_B(r_1) \subseteq \pi_B(r_2)$, as mentioned in Section 2.5.1.

3.3.3.1 Nested-Loop Counting Division

Similar to scalar nested-loop division, *nested-loop counting division (NLCD)* is the most naïve way in the class of aggregate algorithms. This algorithm scans the dividend multiple times. During each scan, NLCD counts the number of rows belonging to the same quotient candidate.

We have to keep track of which quotient candidates we have already checked using a quotient hash table as shown in Figure 3.5(a). A global counter is used to keep track of the number of dividend rows belonging to the same quotient candidate. We fully scan the dividend in an outer loop: we pick the first dividend row, insert its *A* value into the quotient hash table, and set the counter to one. If the counter's value is equal to the divisor count, we add the quotient to the output and continue with the next row of the outer loop. Otherwise, we scan the dividend in an inner loop for rows with the same *A* value as the current quotient candidate. For each such row, the counter is checked and in case of equality, the quotient is added to the output. When the end of the dividend is reached in the inner loop, we continue with the next row of the outer loop and check the hash table if this new row is a new quotient candidate.



(a) Nested-loop counting division (NLCD)

(b) Merge-count division (MCD)

(c) Hash-division for divisor groups (HDD)

(d) Transposed hash-division for divisor groups (HDTD)

(e) Stream-join division (SJD)

Figure 3.5: Overview of data structures and processing used in aggregate algorithms

The time and memory complexities are the same as for nested-loop division.

For all illustrations in Figure 3.5 broken lined boxes indicate that a quotient is found. Only S2's group has as many dividend rows as the divisor.

3.3.3.2 Merge-Count Division

Assuming that the dividend is grouped on *A*, *merge-count division (MCD)* scans the dividend exactly once. After a quotient candidate has been processed and the number of rows is equal to those of the divisor, the quotient is added to the output. Note that the size of a quotient group cannot exceed the number of divisor groups because we have to guarantee referential integrity.

The aggregate algorithm merge-count division is similar to the scalar algorithms MSD and MGD, described in Sections 3.3.2.2 and 3.3.2.3. Instead of comparing the elements of quotient groups with the divisor, MCD uses a representative (the row count) of each quotient group to compare it with the divisor's aggregate. Figure 3.5(b) illustrates the single scan required to

compare the size of each quotient group with the divisor size.

MCD has a worst case time complexity of $O(|r_1|+|r_2|)$ and a typical case time complexity of $O(|r_1|)$. Since no significant data structures have to be kept in memory except for the current dividend row and the counters, the worst case and typical case memory complexity is $O(1)$.

3.3.3.3 Hash-Division for Divisor Groups

In Section 3.3.2.6, we have analyzed optimizations of hash-division that require a dividend that is grouped on *A*. We now show some optimizations of hash-division for a dividend that is grouped on *B*. Unlike the hash-division-like algorithms based on quotient groups, the following two algorithms are blocking.

The first algorithm does not need a divisor hash table because after a divisor group of the dividend has been consumed, the divisor value will never reappear. We use a counter instead of a bitmap for each row in the quotient hash table. We call this adaptation of the HD algorithm *hash-division for divisor groups (HDD)*. The algorithm maintains a counter to count the number of divisor groups seen so far in the dividend. For each dividend row of a divisor group, we increment the counter of the quotient candidate. If the quotient candidate is not yet contained in the quotient hash table, we insert it together with a counter set to one. When the entire dividend has been processed, we return those quotient candidates in the quotient hash table whose counter is equal to the global counter.

3.3.3.4 Transposed Hash-Division for Divisor Groups

The last algorithmic adaptation that we present is called *transposed hash-division for divisor groups (HDTD)*, based on the HDT algorithm. We can do without a divisor hash table, but we keep an array of counters during the scan of the dividend. The processing is basically the same as the previous algorithm (HDD): we return only those quotient candidates of the quotient hash table whose counter is equal to the value of the global counter. Because all divisor groups have to be processed before we know all quotients, this algorithm is also blocking.

We do not show the pseudo code for the HDD and HDTD algorithms for brevity. However, we sketch the data structures used in the Figures 3.5(c) and (d) for the time when the entire dividend has been processed. Note that the dividend contains only three divisor groups (no P3 rows), because we require that referential integrity between *sp* and *p* is preserved, e.g., by applying a semi-join of the two tables before division. S2 is the only supplier who is contained in all three divisor groups.

The complexities of HDD and HDTD are the same. Their worst and typical case time complexity is $O(|r_1|+|r_2|)$ and $O(|r_1|)$, respectively. The worst and typical case memory complexity is $O(|r_1|)$.

3.3.3.5 Stream-Join Division

The new algorithm *stream-join division (SJD)* [NRM00] is an improvement of hash-division for divisor groups (HDD). As all other algorithms assuming a dividend that is grouped on *B* as the

only or the major set of group columns, SJD is a blocking algorithm. SJD is hybrid because it counts the number divisor rows, like all other aggregate algorithms, and it maintains several bits to memorize matches between dividend and divisor, like all other scalar algorithms. However, in this chapter, we consider SJD an aggregate algorithm due to its similarity to HDD.

The major differences between SJD and HDD are:

- SJD stores a bit instead of a counter together with each quotient candidate in the quotient hash table.
- SJD is able to remove quotient candidates from the quotient hash table before the end of the processing.

The SJD algorithm works as follows. As in HDD, we maintain a counter to count the number of divisor groups seen so far in the dividend. First, we insert all quotient candidates, i.e., *A* values, of the first group in the dividend together with a bit initialized with zero into the quotient hash table. We thereby eliminate possible duplicates in the dividend. Then, we process each following group as follows. For each dividend row of the current group, we look up the quotient candidate in the quotient hash table. In case of a match, the corresponding bit is set to one. Otherwise, i.e., when the *A* value of a given dividend row is not present in the quotient hash table, we skip this row. After a group has been processed, we remove all quotient candidates with a bit equal to zero. Then, we reset the bit of each remaining quotient candidate to zero. Finally, when all groups have been processed, we compare the current group counter with the number of rows in the divisor. In case of equality, all quotient candidates in the quotient hash table with a bit equal to one are added to the output.

Figure 3.5(e) illustrates the use of the quotient hash table in SJD. We assume that the dividend is equal to the *sp* table of class 2 in Figure 3.2(b) with the exception that the P3 group {(S2, P3), (S3, P3)} is missing, due to referential integrity. We show the contents of the hash table for the time when the entire *sp* table has been processed. We see that S3 and S1 are not contained in the hash table because both have already been eliminated after the second group (P2). Only S2's bit is set to one and it is a quotient row because the number of groups (three, without P3) is equal to the number of divisor rows.

The advantage of SJD lies in the fact that the amount of memory can decrease but will never increase after the quotient candidates have been stored in the quotient hash table. However, the time and memory complexity is the same as for HDD. Observe that the maximum amount of memory required is proportional to the number of rows of the *first* group in the dividend. It may happen by chance that the first group is the smallest of the entire dividend. In this case, we obtain a very memory-efficient processing.

This algorithm is called stream-join division because it joins all divisor groups of the dividend (called *streams* in [NRM00]) with each other on the columns *A*.

3.4 Evaluation of Algorithms

In this section, we briefly compare the division algorithms discussed in Section 3.3 with each other and show which algorithm is optimal with respect to time and memory complexities for

Division algorithm	Algorithm class	Data class	Dividend r_1		Divisor r_2	Complexity in O-notation			
						Time		Memory	
			A	B	B	worst	typical	worst	typical
NLCD	aggregate	0	$-$	$-$	$-$	$\|r_1\|^2+\|r_2\|$	$\|r_1\|^2$	1	1
NLD	scalar					$\|r_1\|^2+\|r_2\|$	$\|r_1\|^2$	$\|r_1\|+\|r_2\|$	$\|r_1\|$
HD	scalar					$\|r_1\|+\|r_2\|$	$\|r_1\|$	$\|r_1\|\|r_2\|$	$\|r_1\|\|r_2\|$
HDT	scalar					$\|r_1\|+\|r_2\|$	$\|r_1\|$	$\|r_1\|\|r_2\|$	$\|r_1\|\|r_2\|$
HDD	aggregate	2	$-$	$+$	$-$	$\|r_1\|+\|r_2\|$	$\|r_1\|$	$\|r_1\|$	$\|r_1\|$
HDTD	aggregate					$\|r_1\|+\|r_2\|$	$\|r_1\|$	$\|r_1\|$	$\|r_1\|$
SJD	aggregate					$\|r_1\|+\|r_2\|$	$\|r_1\|$	$\|r_1\|$	$\|r_1\|$
MCD	aggregate	5	$+$	$-$	$-$	$\|r_1\|+\|r_2\|$	$\|r_1\|$	1	1
HDQ	scalar					$\|r_1\|+\|r_2\|$	$\|r_1\|$	$\|r_2\|$	1
HDTQ	scalar					$\|r_1\|+\|r_2\|$	$\|r_1\|$	$\|r_2\|$	1
MGD	scalar	10	$+_1$	$+_2^*$	$+^*$	$\|r_1\|\|r_2\|$	$\|r_1\|\|r_2\|$	$\|r_2\|$	1
MSD	scalar			${}^s+_2^*$	${}^s+^*$	$\|r_1\|\|r_2\|$	$\|r_1\|\|r_2\|$	1	1

Table 3.3: Overview of division algorithms

each class of input data discussed in Section 3.2.

Table 3.3 characterizes the algorithms presented so far and shows for each algorithm the class of required input data, its algorithm class, and its time and memory complexities. Input data are either not grouped ($-$), grouped ($+$), or sorted (r_1). Class 10 is first grouped on A, indicated by $+_1$. For each quotient group, it is grouped ($+_2$) or sorted (${}^s+_2$, denoted by a superscript "s") on B in the same order (denoted by a superscript asterisk $+^*$) as the divisor. The algorithm names corresponding to the abbreviations in the first column are given in Table 3.2. We assigned the algorithms to those data classes that have the least restrictions with respect to grouping. Remember that an algorithm of class C can also process data of classes that are reachable from C in the dependency graph in Figure 3.3. The overview of division algorithms in Table 3.3 shows that, despite the detailed classification in Table 3.1 (comprising 13 classes and enumerating all possible kinds of input data), there are *four* major classes of input data that are covered by dedicated division algorithms:

- class 0, which makes no assumption of grouping,
- class 2, which covers dividends that are grouped only or first on B,
- class 5, which covers dividends that are grouped only or first on A, and finally
- class 10, which specializes class 5 (and class 0, of course) by requiring that for each quotient group, the rows of B and the divisor appear in the same order. Hence, the dividend is grouped on A as major and B as minor.

Note that algorithms for class 2, namely HDD, HDTD, and SJD, have not been identified in the literature so far. They represent a new straightforward approach to deal with a dividend that

is grouped on *B*. Together with the other three major classes, a query optimizer can exploit the information on the input data properties to make an optimal choice of a specific division operator.

Suppose, we are given input data of a class that is different from the four major classes. Which algorithms are applicable to process our data? According to the graph in Figure 3.3, all algorithms belonging to major classes, which are direct or indirect parent nodes of the given class, can be used. For example, any algorithm of major classes 0 and 5 can process data of the non-major classes 6, 7, and 9.

Several algorithms belong to each class of input data in Table 3.3. In class 0, both HD and HDT have a linear time complexity (more precisely, *nearly* linear due to hash collisions). However, they have a higher memory complexity than the other algorithms of this class, NLCD and NLD.

We have designed three aggregate algorithms for class 2. They all have the same linear time and memory complexities.

Class 5 has two scalar and one aggregate algorithm assigned to it, which all have the same time complexity. The constant worst case memory complexity of MCD is the lowest of the three.

The two scalar algorithms MGQ and MSD of class 10, which consists of two subgroups (sorted and grouped divisor values) have the same time complexity. The worst case memory complexity of MSD is lower than that of MGD because MSD can exploit the sort order. It may seem odd that the worst case time complexity is higher than for class 5. This is due to our definition of the worst case: If the number of (distinct) values in $\pi_A(r_1)$ is as high as $|r_1|$, i.e., if each dividend tuple forms a quotient candidate then the time complexity of $O(|r_1| + |A||r_2|)$ becomes equal to $O(|r_1||r_2|)$, as we wrote in Sections 3.3.2.2 and 3.3.2.2.

It is important to observe that one should not directly compare complexities of scalar and aggregate algorithms in Table 3.3 to determine the most efficient algorithm overall. This is because *aggregate* algorithms require duplicate-free input tables, which can incur a very costly preprocessing step. There is one exception of aggregate algorithms: SJD ignores duplicate dividend rows because of the hash table used to store quotient candidates. It does not matter if a quotient occurs more than once inside a divisor group because the bit corresponding to a quotient candidate can be set to one any number of times without changing its value (1). However, SJD does not ignore duplicates in the divisor because it counts the number of divisor rows.

In general, *scalar* division algorithms ignore duplicates in the dividend and the divisor. Note that the scan operations of MGD and MSD can be implemented in such a way that they ignore duplicates in both inputs [GC95]. However, to simplify our presentation, the pseudo code of MSD in Appendix B on page 175 does not ignore duplicates.

Let us briefly illustrate some example issues that we have to take into account when comparing division algorithms. The first issue is time versus memory complexity. In class 0, for example, four algorithms have been identified. NLCD and NLD have a quadratic time complexity compared to the linear complexities of HD and HDT. Despite the different processing performance of these algorithms, a query optimizer may prefer to pick a division operator based on the NLCD algorithm instead of HD or HDT if the estimated amount of input data is small and the optimizer wants to avoid the overhead of building hash tables. We do not go into the details of query optimization here because, in general, the choice of picking a specific operator from a set of logically equivalent operators (like join and division) also depends on factors other than

time and memory complexity, as we have mentioned in Section 3.2.2. Nevertheless, time and memory consumption are the dominant factors in reality.

The second issue is about the efficiency of a query processor for certain operations. We presented two different approaches for hash-division: the classic approach (HD), where bitmaps are stored together with quotient candidates in the quotient hash table, and a new approach (HDT) where bitmaps are stored with each divisor row in the divisor hash table (see Figures 3.4(d) and (e) for illustrations). These dual approaches may seem interchangeable at first sight with respect to efficiency. However, in some situations, a query optimizer may prefer one rather than the other, depending on how efficiently the system processes bitmaps. Suppose the system can process a few extremely long bitmaps more efficiently than many short bitmaps. If there are many quotient candidates in the input data (which is typical) but there is a relatively short divisor, then the bitmaps stored in HD are relatively short but there are many of them. In contrast, HDT would build very long bitmaps (which may be the deciding factor) but only a few of them would be stored in the divisor hash table. Analogously, the optimizer may prefer HD to HDT if the input consists of few but very large quotient candidates. Similar situations apply to the other pairs of transposed and non-transposed algorithms, i.e., for the HDD/HDTD and HDQ/HDTQ pairs.

3.5 Related Work

Quantifiers in queries can be expressed by relational algebra. Due to the lack of efficient division algorithms in the past, early work has recommended avoiding the relational division operator to express universal quantification in queries [Bry89]. Instead, universal quantification is expressed with the help of the well-known *anti-semi-join* operator, or *complement-join*, as it is called in that paper.

Other early work suggests approaches other than division to process (universal) quantification [Day83, Day87]. Universal quantification is expressed by new algebra operators and is optimized based on query graphs in a non-relational data model [Day87]. Due to the lack of a performance analysis, we cannot comment on the efficiency of this approach.

The research literature provides only few surveys of division algorithms [CKMP97, Gra93, GC95]. Some of the algorithms reviewed in this paper have been compared both analytically and experimentally [GC95]. The conclusion is that hash-division outperforms all other approaches. Complementing this work, we have shown that an optimizer has to take the input data characteristics and the set of given algorithms into account to pick the best division algorithm. The classification of four division algorithms in [GC95] is based on a two-by-two matrix. One axis of the matrix distinguishes between algorithms based on sorting or based on hashing. The other axis separates "direct" algorithms, which allow processing the (larger) dividend table only once, from "indirect" algorithms, which require duplicate removal (by employing semi-join) and aggregation. For example, the merge-sort division algorithm of Section 3.3.2.2 falls into the category "direct algorithm based on sorting," while the hash-division for divisor groups algorithm of Section 3.3.3.3 belongs to the combination "indirect algorithm based on hashing." Our classification details these four approaches and focuses on the fact that data properties should be exploited as much as possible by employing "slim" algorithms that are separated from preprocessing algo-

rithms, like grouping and sorting.

Based on a classification of queries that contain universal quantification, several query evaluation techniques have been analyzed [CKMP97]. The input data of this algorithm analysis is stored in an object-oriented or object-relational database, where set-valued columns are available. Hence, the algorithms they examine can presuppose that the input data is grouped on certain columns. For example, the table *sp* in Figure 3.1 could be represented by a set-valued column *p#set* of a supplier table *s*. The authors conclude that universal quantification based on anti-semi-join is superior to all other approaches, similar to the conclusion of [Bry89]. Note, however, that this paper has a broader definition of queries involving universal quantification than the classic definition that involves the division operator. However, the anti-semi-join approach requires a considerable overhead for preprocessing the dividend. An equivalent definition of the division operator using anti-semi-join ($\overline{\ltimes}$) as well as semi-join ($\ltimes$) and left outer join ($\bowtie_{lo}$), is: $r_1 \div r_2 = ((r_1 \ltimes r_2) \bowtie_{lo} r_2) \overline{\ltimes} r_2$.

In this chapter, we focused on the universal (for-all) quantifier. *Generalized quantifiers* have been proposed to specify quantifiers like "at least ten" or "exactly as many" in SQL [HP95]. Such quantifiers can be processed by algorithms that employ multi-dimensional matrix data structures [RBG96]. In that paper, however, the implementation of an operator called *all* is presented that is similar but different from relational division. Unlike division, the result of the *all* operator contains some attributes of the divisor. Hence, we have to employ a projection on the quotient attributes of the *all* operator's result to achieve a valid quotient.

Transformation rules for optimizing queries containing multiple (existential and universal) quantifications are presented in [JK83]. Our contribution complements this work by offering strategies to choose a single (division) operator, which may be one element of a larger query processing problem.

3.6 Summary

Based on a classification of input data properties, we were able to differentiate the major currently known algorithms for relational division. In addition, we could provide new algorithms for previously not supported data properties. Thus, for the first time, an optimizer has a full range of algorithms, separated by their input data properties and efficiency measures, to choose from.

We are aware of the fact that database system vendors are reluctant to implement several alternative algorithms for the same query operator, in our case the division operation. One reason is that the optimizer's rule set has to be extended, which can lead to a larger search space for queries containing division. Another reason is that the optimizer must be able to detect a division in a query. This is a non-trivial task because a division cannot be expressed in the current SQL standard [ISO02]. We have suggested a hypothetical SQL syntax for division (the "small" divide) in Section 2.5.2.

The following chapter is dedicated to algorithms for the set containment join and set containment division operator. The latter has an algebraic definition that is based on the division operator and that can thus exploit the algorithms presented in this chapter.

"[...] even if we didn't come up with new ideas in universities, even if we just take what's out there [from industry] but analyze it, study it, organize the ideas so that people can learn and improve things, I think that that's a great contribution."

H. Garcia-Molina [Win02]

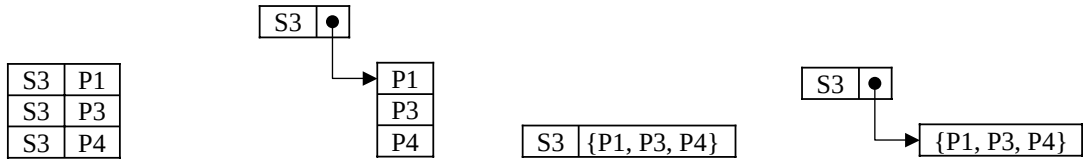
4

Algorithms for Set Containment Division and Set Containment Join

In this chapter, we discuss how to realize physical operators for set containment division and set containment join by reviewing known approaches, discussing parallelization strategies, and presenting a new data structure to improve the performance for set containment division when at least one of dividend or divisor fits into main memory. Most of these operators have been implemented in a query execution engine. They are subject to performance experiments and to a discussion of implementation details in Chapter 6.

4.1 Introduction

Division is an operator of the relational algebra, which is based on the relational model. In the basic relational model, all relations are in first normal form (1NF), i.e., all attribute domains are atomic. One possible extension of the relational model provides relations with multivalued attributes, where the attribute domain is a collection type like bag or set, defined on top of a primitive domain like floating-point number or text string. A more rigorous extension of the relational model is the *nested relational model* [Mak77, JS82], where attributes can be relations them-



(a) Unnested internal (b) Unnested external (c) Nested internal (d) Nested external

Figure 4.1: Storage representations of set-valued attributes

s#	$p\#set_1$
S1	{P1, P4}
S2	{P1, P2, P3, P4}
S3	{P1, P3, P4}

(a) *sp*

$p\#set_2$	*color*
{P1, P2, P4}	blue
{P1, P3}	red

(b) *p*

s#	$p\#set_1$	$p\#set_2$	*color*
S2	{P1, P2, P3, P4}	{P1, P2, P4}	blue
S2	{P1, P2, P3, P4}	{P1, P3}	red
S3	{P1, P3, P4}	{P1, P3}	red

(c) $sp \bowtie_{p\#set_1 \supseteq p\#set_2} p$

Figure 4.2: An example containment join on the set-valued attribute *p#set*

selves. In particular, Jaeschke et al. [JS82] suggest the *nest/unnest* operators. Further proposals have been made for transforming data between non-1NF and 1NF, e.g., Fischer et al. [FT82] (nest/unnest) and Özsoyoğlu et al. [ÖÖM87] (pack/unpack).

Two orthogonal classifications have been suggested for the storage representation of sets: nesting and location [Ram02]. The *nested* representation stores the values as a variable length attribute, the unnested representation stores them as multiple tuples. In a classification based on the storage *location*, one can distinguish between an internal representation, where the set elements are stored together with the accompanying attribute values, and an external representation, where the set elements are stored in a separate auxiliary table connected by foreign key references, as depicted in Figure 4.1, inspired by [Ram02]. In this figure, we show a subset of the "supplies" table *sp*, where the supplier number *s#* is a column of an atomic data type and the parts being supplied are of a set-valued data type. Here, we represent the fact that supplier S3 supplies parts P1, P3, and P4. Only the unnested internal representation conforms to 1NF.

Figure 4.2 illustrates an example computation of the set containment join based on a supplier-parts database similar to that in Figure 2.4 on page 13, but with different data. We find that supplier S2 offers all blue and all red parts. All red parts are also supplied by supplier S3.

Suppose, the tables *sp* and *p* are defined as before and that the layout of the set-valued columns $p\#set_1$ and $p\#set_2$ is *unnested internal* for both tables, as sketched in Figure 4.3. How is the set containment join result represented in an unnested internal layout? Since all join attributes have to be preserved in a theta-join result, one possible definition for representing the matches could be to pair each row from the left side with each row of the right side, i.e., one could com-

s#	*p#*
S1	P1
S1	P4
S2	P1
S2	P2
S2	P3
S2	P4
S3	P1
S3	P3
S3	P4

(a) *sp*

p#	*color*
P1	blue
P2	blue
P4	blue
P1	red
P3	red

(b) *p*

s#	*color*
S2	blue
S2	red
S3	red

(c) $sp \div^* p$

Figure 4.3: An example set containment division

pute the Cartesian product between each pair of groups that fulfills the set containment predicate, as illustrated in Table 4.1. However, the result table cardinality can become enormous. The set containment division operator is more space-efficient because it returns only the columns of the non-join attributes. By joining the result with *p* and *sp*, we could produce the same result as the set containment *join* in an unnested internal representation.

The remainder of this chapter is organized as follows. In Section 4.2, we give an overview of set containment join algorithms before we discuss basic algorithms for set containment division in Section 4.3. Section 4.4 discusses horizontal partitioning options that enable a parallelization of set containment division. In Section 4.5, we introduce a subset index data structure to improve the performance of set containment division when at least one table fits into main memory. Section 4.6 discusses the role of indexes that manage data on external memory. We summarize this chapter in Section 4.7.

4.2 Set Containment Join Algorithms

The set containment problem has been studied in great detail in the past [HM97, MGM03, MGM02a, MGM02b, Ram02, RPNK00]. In particular, several efficient set containment test algorithms have been developed and storage structures to represent sets in relational, object-relational, and object-oriented databases have been discussed. Interestingly, the division operator is closely related to set containment join but the literature on set containment join has not mentioned this fact.

There are two classes of algorithms realizing the set containment join, one based on signatures [HM97] and the other based on partitioning [RPNK00]. Enhanced algorithms combining both techniques have been developed which significantly outperform all previous approaches [MGM03, MGM02a, MGM02b]. In this section, we summarize these approaches.

All set containment join algorithms assume that the join attributes are set-valued. Melnik et al. [MGM02b] compare the new approaches to SQL-based approaches based on *counting* the number of elements in the join result between two sets and comparing it to the set cardinality

s#	$p\#_1$	$p\#_2$	*color*
S2	P1	P1	blue
S2	P1	P2	blue
S2	P1	P4	blue
S2	P2	P1	blue
S2	P2	P2	blue
S2	P2	P4	blue
S2	P3	P1	blue
S2	P3	P2	blue
S2	P3	P4	blue
S2	P4	P1	blue
S2	P4	P2	blue
S2	P4	P4	blue
S2	P1	P1	red
S2	P1	P3	red
S2	P2	P1	red
S2	P2	P3	red
S2	P3	P1	red
S2	P3	P3	red
S2	P4	P1	red
S2	P4	P3	red
S3	P1	P1	red
S3	P1	P3	red
S3	P3	P1	red
S3	P3	P3	red
S3	P4	P1	red
S3	P4	P3	red

Table 4.1: Hypothetical result of set containment join using an unnested internal storage representation

of the candidate subset, as discussed in Section 2.5.1. Unfortunately, no comparison is given with SQL-based approaches using NOT EXISTS, also mentioned in Section 2.5.1.

A recent study compared set containment joins based on a nested internal and an unnested internal set representation [HM02], also based on the counting approach, only. In the nested approach, a user-defined containment test predicate is employed that takes two set-valued attributes as parameters. According to current database technology for evaluating user-defined predicates, the commercial system in use applies the test predicate on the result table of a Cartesian product of both input tables. By rewriting the query into one using an unnested layout, a table function is employed that unnests the set-valued attribute into a table. The optimizer of the system used in their experiments decided to first build an intermediate result table that comprises the set identifier and the element value as attributes, sorted on the element values. Then, the query execution plan suggests to merge-join the two sorted input streams on the element value attributes. After

that, the sorted data is grouped on the set identifiers and set cardinalities. Finally, a filter condition appends only those set identifier pairs to the result where the cardinality of the contained set is equal to the number of matches for this pair of sets. The experiments of this study have shown that the effort of unnesting the sets and preprocessing the data by sorting it on the attributes to be matched can greatly improve the straightforward nested-loop approach. Unfortunately, the results have not been compared to more sophisticated approaches as the ones proposed, for example, in [MGM03].

Helmer gives an excellent overview of algorithms for set containment joins [Hel00]. His focus is on index structures for set-valued attributes, in particular sequential signature files, extensible signature hashing, recursive linear signature hashing, and inverted files. His work does not discuss the relationship of the set containment join to relational division.

Ramasamy also discusses set containment join algorithms [Ram02]. In this work, the algorithm partitioning set join is discussed in great detail. We will present this algorithm in Section 4.2.4. He does not discuss algorithms for the unnested internal table layout because it "replicates the rest of attributes for each set element, thereby consuming a large amount of storage. Second, this replication leads to update anomalies." The first argument is valid but in reality, almost all databases store sets in an unnested internal format because they comply to the first normal form (1NF). We believe that it is necessary and interesting to investigate also algorithms that do not require to nest and unnest data on the fly but that can perform containment tests on data in 1NF. The second argument is not clear. Can the supplier-parts database in Figure 2.4 lead to update anomalies? No, but it may lead to anomalies if we had stored the part color information not in *p* but in the *sp* relation. It depends on the database design if an update can lead to an anomaly.

In a recent study, Mamoulis [Mam03] reviewed several approaches for joins on set-valued attributes, including set containment and set overlap. He suggested a new algorithm called *block nested-loop using an inverted file* that outperformed the partitioning set join. However, this work did not cover the more sophisticated algorithms by Melnik [MGM03], which are claimed to outperform partitioning set join as well.

4.2.1 Signature-Based Set Containment Join Algorithms

The idea of this approach is to reduce the expensive set comparison costs by using bitmap operations. Given a bitmap of length l and a hash function h that maps a set element value to an index of the bitmap between 1 and l, a signature $sig(S)$ of a set S is computed as follows: For each set element value v, set the bit at position $h(v)$ to 1. We test for two sets S_1 and S_2 whether $S_1 \subseteq S_2$ by first evaluating the expression $sig(S_1) \wedge \neg sig(S_2) \neq 0$, where $\wedge$ and $\neg$ are bitwise AND and NOT operators. If this expression is true, we can safely skip this value pair because there is at least one element $v \in S_1$ that is not contained in S_2, leading to $sig(h(v)) = 1$ for S_1 but $sig(h(v)) = 0$ for S_2.[1] Otherwise, we remove false positives by checking if really $S_1 \subseteq S_2$. The advantage of this approach is that all negatives are efficiently found.

Figure 4.4 shows an example signature-based set containment join based on Figure 4.2 with

[1] Here, we have applied the *sig* function to a single index position of the bitmap instead of an entire set of values.

s#	*p#set*	*sig* (*p#set*)	¬*sig* (*p#set*)
S1	{P1, P4}	010	101
S2	{P1, P2, P3, P4}	111	000
S3	{P1, P3, P4}	011	100
S4	{P3, P4}	011	100

(a) *sp*

p#set	*color*	*sig* (*p#set*)
{P1, P2, P4}	red	110
{P1, P3}	blue	011

(b) *p*

s#	*sp.p#set*	*p.p#set*	*color*	¬*sig* (*sp.p#set*) ∧*sig* (*p.p#set*)	Need to verify?	False positive?	In result?	Row
S1	{P1, P4}	{P1, P2, P4}	red	100	N	–	N	1
S1	{P1, P4}	{P1, P3}	blue	001	N	–	N	2
S2	{P1, P2, P3, P4}	{P1, P2, P4}	red	000	Y	N	Y	3
S2	{P1, P2, P3, P4}	{P1, P3}	blue	000	Y	N	Y	4
S3	{P1, P3, P4}	{P1, P2, P4}	red	100	N	–	N	5
S3	{P1, P3, P4}	{P1, P3}	blue	000	Y	N	Y	6
S4	{P3, P4}	{P1, P2, P4}	red	100	N	–	N	7
S4	{P3, P4}	{P1, P3}	blue	000	Y	Y	N	8

(c) $sp \bowtie_{p\#set \supseteq p\#set} p$

Figure 4.4: An example set containment join using signatures

one additional supplier in the *sp* table, namely S4 supplying parts P3 and P4. Suppose that the length of a bitmap l is 3 and that the hash functions are defined as the integer value of the last character of *p#* modulo l. Hence, $h(\text{P1}) \bmod 3 = 1$, $h(\text{P2}) \bmod 3 = 2$, $h(\text{P3}) \bmod 3 = 0$, and $h(\text{P4}) \bmod 3 = 1$. When we compute the bitmap expression $\neg sig(sp.p\#set) \wedge sig(p.p\#set)$, we find that the rows number 1, 2, 5, and 7 cannot qualify for the result because the resulting bitmap is unequal to zero. Only the remaining rows have to be checked, where we find that three of the four rows belong to the result table, namely rows 3, 4, and 6, and that row 8 turns out to be a false positive. Instead of comparing the sets in all eight combinations, the use of signatures reduces the comparison overhead to $\frac{4}{8} = 50\%$.

4.2.2 The S-Tree

The *S-tree*, introduced by Deppisch [Dep86], is a height-balanced multi-way tree that is similar to a B-tree [BM72]. In contrast to a B-tree, the inner nodes of an S-tree store the *signatures* (bitmaps) and the leaf nodes store pairs of an object or object identifier together with its signature. Each edge from an inner node to a child node is associated with a signature that is the result of a logical OR-operation over all signatures stored in the child node.

Figure 4.6 illustrates an S-tree for table *sp* in Figure 4.5(a). This S-tree is of type $(K,k,h) = (3,1,3)$, where K denotes the maximum degree of a node and k is the minimum degree of a node except for the root, which is allowed to have at least two children. Parameter h is the height of the tree. Although the illustrated S-tree only consists of nodes with at least two children, during

s#	*p#set*	*sig*(*p#set*)
S1	{P1, P2, P4, P5, P8}	11011
S2	{P2, P3, P4, P5}	01111
S3	{P2, P3, P4}	01110
S4	{P1, P2, P6}	11000
S5	{P1, P4, P7}	10010
S6	{P1, P3}	10100
S7	{P3, P4}	00110
S8	{P2, P3}	01100
S9	{P2, P6}	01000
S10	{P4, P5}	00011
S11	{P8}	10000
S12	{P3}	00100

(a) *sp*

p#set	*color*	*sig*(*p#set*)
{P8}	blue	10000
{P6}	green	00000
{P1, P2, P5, P6}	red	11001
{P5, P8}	yellow	10001

(b) *p*

Figure 4.5: An example supplier-parts database

the construction of the S-tree, it happened that a node had only a single entry. The signatures defined for the sets in table *sp* have been computed using several hash functions but this fact is not important for this example.[2]

The retrieval of supersets by traversing the S-tree is straightforward. Suppose, we wanted to find the sets containing the "blue" parts (part P8). Using our hash function, which we do not discuss at this moment, the set {P8} has the signature 10000. We begin at the root and test all signatures s in the node if $10000 \wedge \neg s = 00000$. We find that this is true for both signatures in the root. We recursively test the associated children in the same way. Hence, the paths that lead from the root to the candidate sets are:

- $11100 \rightarrow 10100 \rightarrow 10000 \rightarrow$ S11,
- $11100 \rightarrow 10100 \rightarrow 10100 \rightarrow$ S6,
- $11111 \rightarrow 10011 \rightarrow 10010 \rightarrow$ S5,
- $11111 \rightarrow 11011 \rightarrow 11000 \rightarrow$ S4, and
- $11111 \rightarrow 11011 \rightarrow 11011 \rightarrow$ S1.

All of the five qualifying sets have to be checked, where we find that the resulting quotient is $\{S1, S11\}$, i.e., there are three false positives.

The insertion and deletion of a set is more challenging and several variations have been studied. In particular, the split criterion offers room for optimization: How do we distribute the signatures into two nodes when there is a node overflow? A similar problem is how to

[2]The signatures are the same as the concatenation of bit values used in Figure 4.7(a) for the adaptive pick-and-sweep join algorithm that we will discuss in Section 4.2.5.

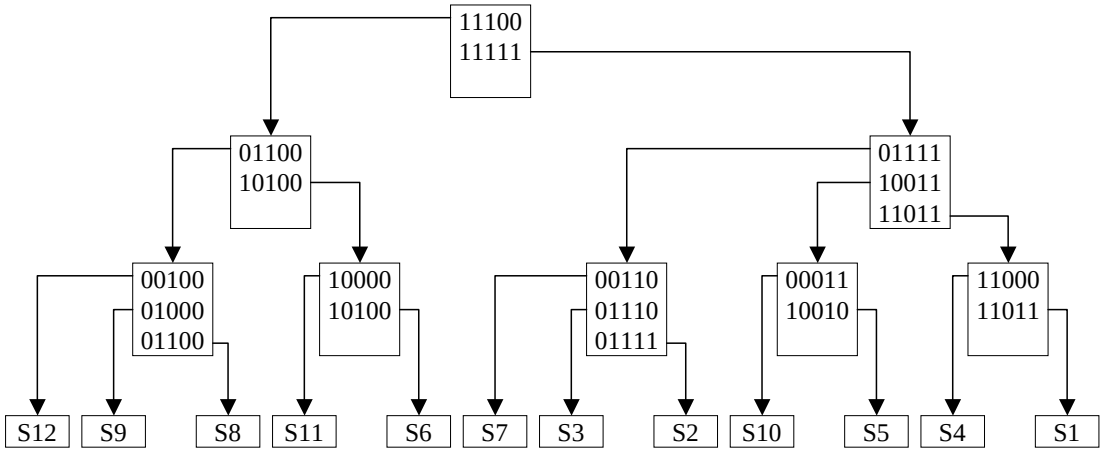


Figure 4.6: An example S-tree for table *sp* in Figure 4.5(a)

collapse empty or almost empty nodes into fewer nodes? In the original work on S-trees [Dep86], Deppisch suggests the measure *signature weight* γ, which is the number of ones in the signature. When a node needs to be inserted, one traverses the tree recursively, starting at the root, and compares the *node signatures* contained in the node with the given signature in order to find the path to the leaf where the insertion will take place. One approach to measure the similarity between two signatures s and s' is to use the *Hamming distance metric* $\delta(s,s') = \gamma(s \vee s') - \gamma(s \wedge s')$, i.e., the number of bits in the signatures with different values. Since a node should be split such that the signature weight of the two new nodes are low, a different measure is suggested by Deppisch, called *weight increase distance* $\varepsilon = \gamma(\neg s \wedge s')$. Instead of computing the expression $\gamma(s \vee s')$ in the Hamming distance metric, one should use the expression $\gamma(s) + \varepsilon(s,s')$. We do not discuss further criteria in this thesis but refer to the extensive literature on this topic, e.g., [NM02, TBM02].

4.2.3 Partition-Based Set Containment Join Algorithms

Using signatures can improve the cost of set comparisons. Nevertheless, the *number* of signature comparisons between a dividend table r_1 and a divisor table r_2 remains $|r_1||r_2|$. Partitioning can help to reduce this number. The idea of partitioning is to decompose the input tables r_1 and r_2 into k subsets $r_1^1,\ldots,r_1^k$ and $r_2^1,\ldots,r_2^k$ such that $r_1 \bowtie r_2 = \bigcup_{i=1}^{k} r_1^i \bowtie r_2^i$. In our case, the join operator is a set containment join $\bowtie_{\supseteq}$.

In the following, we use the terminology introduced in [MGM03]. A *partitioning function* π assigns set elements in a table r_1 and r_2 to one or more partitions $r_1^1,\ldots,r_1^k$ and $r_2^1\ldots,r_2^k$, respectively. Two special values characterize the quality of a partition-based algorithm. The *comparison factor* c is the quotient of the number of comparisons using partitioning and the number of those without partitioning. In an implementation of such an algorithm in an RDBMS, the signatures will be stored in files together with the row identifiers, not the columns of the row. The other value, called *replication factor* r, evaluates the fact that some of the signatures occur

in more than one partition, i.e., that some data is replicated.

EXAMPLE 4: Consider the dividend table *sp* and divisor table *p* in Figure 4.5. Suppose π is defined such that

- $\pi(\text{P6}) = \pi(\text{P7}) = \pi(\text{P8}) = \{1\}$,
- $\pi(\text{P1}) = \pi(\text{P2}) = \pi(\text{P3}) = \pi(\text{P4}) = \{2\}$,
- $\pi(\text{P5}) = \{1,3\}$.

Hence, r_1 is partitioned into

- $r_1^1 = \{\text{S1}, \text{S2}, \text{S4}, \text{S5}, \text{S9}, \text{S10}, \text{S11}\}$,
- $r_1^2 = \{\text{S1}, \text{S2}, \text{S3}, \text{S4}, \text{S5}, \text{S6}, \text{S7}, \text{S8}, \text{S9}, \text{S10}, \text{S12}\}$, and
- $r_1^3 = \{\text{S1}, \text{S2}, \text{S10}\}$.

The partitions of r_2 are

- $r_2^1 = \{\text{blue}, \text{green}, \text{red}, \text{yellow}\}$,
- $r_2^2 = \{\text{red}\}$, and
- $r_2^3 = \{\text{red}, \text{yellow}\}$.

The number of comparisons for computing $\bigcup_{i=1}^{3} r_1^i \bowtie_{\supseteq} r_2^i$ is $\sum_{i=1}^{3} |r_1^i||r_2^i| = 7 \cdot 4 + 11 \cdot 1 + 3 \cdot 2 = 45$, while the number of comparisons for the original expression $r_1 \bowtie_{\supseteq} r_2$ is $12 \cdot 4 = 48$. The saving is $\frac{3}{48} = \frac{1}{16}$. Hence, the comparison factor is $c = \frac{45}{48} = \frac{15}{16} = 0.9375$.

In our example, the number of (signature, row identifier)-pairs that are stored in a file is $\sum_{i=1}^{3} |r_1^i| + |r_2^i| = (7+2) + (11+1) + (3+2) = 28$. In the original tables, the number of signatures that belong to r_1 and r_2 is $12 + 4 = 16$. This results in a replication factor r of $\frac{28}{16} = \frac{7}{4} = 1.75$. □

The aim of any partition-based algorithm is to reduce the comparison factor c and the replication factor r as much as possible, i.e., the optimal values are $c = 0$ and $r = 1$. Of course, there is a trade-off between these two factors and the challenge is to define the function π appropriately. The following two algorithms, partitioning set join and adaptive pick-and-sweep join, aim at optimizing these factors.

4.2.4 Partitioning Set Join

The *partitioning set join (PSJ)*, suggested in [RPNK00], takes as input a user-defined number of partitions k. For each set in the dividend r_1, we assign it to *all* those partitions r_1^i where there is an element e in the set such that $(v(e) \mod k) + 1 = i$. Each set in the divisor r_2 is assigned to a single partition. We *randomly select* an element e from a set. Let the integer value of this element be $v(e)$. We compute $(v(e) \mod k) + 1 = i$ and assign the set to partition r_2^i.

Symbol	Description in terms of set containment join (set containment division)
r_1	Dividend table, i.e., table of supersets w.r.t. r_2
r_2	Divisor table, i.e., table of subsets w.r.t. r_1
$\|r_1\|$	Table cardinality of r_1
$\|r_2\|$	Table cardinality of r_2
θ_1	Average set cardinality of sets in r_1 (average number of rows in a dividend group)
θ_2	Average set cardinality of sets in r_2 (average number of rows in a divisor group)
ρ	Ratio of table cardinalities, $\rho = \frac{\|r_1\|}{\|r_2\|}$
λ	Ratio of set cardinalities, $\lambda = \frac{\theta_1}{\theta_2}$
k	Number of partitions
l	Number of hash functions used in APSJ, $l = k-1$

Table 4.2: Parameters used for analyzing the algorithms

According to [MGM02b], the comparison factor of PSJ is

$$c_{\text{PSJ}} = 1 - \left(1 - \frac{1}{k}\right)^{\theta_1},$$

and the replication factor is

$$r_{\text{PSJ}} = \frac{1}{\rho+1} + \frac{\rho}{\rho+1} k \left(1 - \left(1 - \frac{1}{k}\right)^{\theta_1}\right),$$

where θ_1 is the average number of set elements in r_1 and ρ is the quotient of the table cardinalities of r_1 and r_2. These and other parameters that describe the characteristics of datsets and algorithms are summarized in Table 4.2.

The assumptions made for deriving the two formulas are [MGM02b]:

- Each set in r_2 contains a fixed number of elements θ_2 and each set in r_1 contains θ_1 elements, where $0 < \theta_2 \leq \theta_1$.
- The set elements of r_1 and r_2 are drawn from an integer domain D using a uniform probability distribution. Furthermore, $|D| \gg k$, $|D| \gg \theta_1$, and $|D| \gg \theta_2$.
- Joining each pair of partitions r_1^i and r_2^i requires $|r_1^i||r_2^i|$ signature comparisons, no clever join strategy is employed but merely a simple strategy like nested-loop join.

EXAMPLE 5: As in the example of the previous section, we consider the supplier-parts database in Figure 4.5. Suppose that the number of partitions is $k = 3$ and that v, the function that maps an element to an integer value, is defined as the part number without the prefix "P," e.g., $v(\text{P8}) = 8$. Since, e.g., $(v(\text{P8}) \mod 3) + 1 = 2 + 1 = 3$, the dividend sets with *s#* values S1 and S11, that both contain P8, are assigned to partition r_1^3. The partitioning function can be specified as follows:

- $\pi(\text{P3}) = \pi(\text{P6}) = \{1\}$,
- $\pi(\text{P1}) = \pi(\text{P4}) = \pi(\text{P7}) = \{2\}$, and
- $\pi(\text{P2}) = \pi(\text{P5}) = \pi(\text{P8}) = \{3\}$.

The dividend table r_1 consists of the partitions

- $r_1^1 = \{\text{S2}, \text{S3}, \text{S4}, \text{S6}, \text{S7}, \text{S8}, \text{S9}, \text{S12}\}$,
- $r_1^2 = \{\text{S1}, \text{S2}, \text{S3}, \text{S4}, \text{S5}, \text{S6}, \text{S7}, \text{S10}\}$, and
- $r_1^3 = \{\text{S1}, \text{S2}, \text{S3}, \text{S4}, \text{S8}, \text{S9}, \text{S10}, \text{S11}\}$.

For the sets in the divisor r_2, suppose that we randomly picked element P2 from the "red" set. Hence, r_2 is partitioned as follows:

- $r_2^1 = \{\text{green}\}$, and
- $r_2^2 = \{\}$, and
- $r_2^3 = \{\text{blue}, \text{red}, \text{yellow}\}$.

The number of comparisons for computing $\bigcup_{i=1}^{3} r_1^i \overline{\ltimes}_{\supseteq} r_2^i$ is $\sum_{i=1}^{3} |r_1^i||r_2^i| = 8 \cdot 1 + 8 \cdot 0 + 8 \cdot 3 = 32$, leading to a comparison factor of $c = \frac{32}{48} = \frac{2}{3} = 0.\overline{6}$. The number of signatures to be stored in a file is $\sum_{i=1}^{3} |r_1^i| + |r_2^i| = (8+1) + (8+0) + (8+3) = 28$. Hence, the replication factor is $r = \frac{28}{16} = \frac{7}{4} = 1.75$. The comparison factor value is closer to the optimum than in Example 4, whose partitioning function was constructed arbitrarily. However, the partitioning factor value is the same.

Suppose, the data of the dividend and divisor had the properties stated in the assumptions for the theoretical values for c and r, given above. In our example, we have $|r_1| = 12$, $|r_2| = 4$, $|D| = 8$, $k = 3$, $\theta_1 = \frac{30}{12} = \frac{5}{2}$, $\theta_2 = \frac{7}{4}$, and $\rho = \frac{12}{4} = 3$. Hence, we would have a theoretical comparison factor of

$$c_{\text{PSJ}} = 1 - \left(1 - \frac{1}{k}\right)^{\theta_1} = 1 - \left(1 - \frac{1}{3}\right)^{\frac{5}{2}} \approx 0.637,$$

and a replication factor of

$$r_{\text{PSJ}} = \frac{1}{\rho + 1} + \frac{\rho}{\rho + 1} k c_{\text{PSJ}} = \frac{1}{4} + \frac{9}{4} c_{\text{PSJ}} \approx 1.684.$$

In this example, the values that we computed manually differ from the theoretical values by $\frac{c - c_{\text{PSJ}}}{c_{\text{PSJ}}} = 5\%$ for the comparison factor and by $\frac{r - r_{\text{PSJ}}}{r_{\text{PSJ}}} = 4\%$ for the replication factor.

□

s#	h_1	h_2	h_3	h_4	h_5
S1	1	1	0	1	1
S2	0	1	1	1	1
S3	0	1	1	1	0
S4	1	1	0	0	0
S5	1	0	0	1	0
S6	1	0	1	0	0
S7	0	0	1	1	0
S8	0	1	1	0	0
S9	0	1	0	0	0
S10	0	0	0	1	1
S11	1	0	0	0	0
S12	0	0	1	0	0

(a) *sp*

color	h_1	h_2	h_3	h_4	h_5
blue	1	0	0	0	0
green	0	0	0	0	0
red	1^*	1	0	0	1
yellow	1	0	0	0	1^*

(b) *p*

Figure 4.7: Example hash functions for adaptive pick-and-sweep join

4.2.5 Adaptive Pick-and-Sweep Join

The idea of partitioning set join has been generalized by a more sophisticated definition for the partitioning function. The new approach, called *adaptive pick-and-sweep join (APSJ)*, was suggested by Melnik et al. [MGM03]. The authors claim that APSJ outperforms PSJ "most of the time" for large set cardinalities (greater than ten). They show how to "adaptively" construct partitioning functions that are optimal w.r.t. the input data characteristics, namely the table cardinality and the average set cardinality.

In the following, we use the terminology of Melnik et al. [MGM02b]. The APSJ algorithm partitions r_1 and r_2 into k partitions as follows. Suppose, there exist $k-1$ *boolean hash functions* $h_1, \ldots, h_{k-1}$, where each function takes a set V of integer values as input and returns 0 (*false*) or 1 (*true*). An example hash function is $h_i(V) = 1 \Leftrightarrow \exists e \in V : e \mod 7 = i$. The tables in Figure 4.7 show which hash functions fire (return the value 1) for a given element value for the tables in Figure 4.5. In this figure, we assume that we ignore the prefix "P" of the (text string) *p#* value to create the integer value for a set element, as described in Example 5 on page 62.

For each set $V \in r_1$, V is added to *all* partitions, whose hash function fire. In addition, V is assigned to the default partition r_1^0. Hence, e.g., the set S8 is assigned to r_1^0, r_1^2, and r_1^3.

For each set $V \in r_2$, the indexes of hash functions that fire are given by $I = \{i \mid h_i(V) = 1\}$. One *randomly* picks one of these values $j \in I$ and assigns V to r_2^j. If there is no index, i.e., $I = \emptyset$, V is assigned to the *default partition* r_2^0. In the table p in Figure 4.7(b), representing the dividend r_2, we have indicated by an asterisk the randomly chosen index for those sets that have more than one index. For example the "red" set is assigned to partition r_2^1.

The partitioning function can be described as

- $\pi(\text{P1}) = \pi(\text{P8}) = \{1\}$,
- $\pi(\text{P2}) = \{2\}$,

- $\pi(\text{P3}) = \{3\}$,
- $\pi(\text{P4}) = \{4\}$, and
- $\pi(\text{P5}) = \{5\}$.

Please notice that the function is not defined for the values P6 and P7, i.e., in this example, π is a *partial* function.

Together with the default partition, APSJ uses k partitions for each input table. Because of the default partitions, APSJ can handle empty sets in the input tables, unlike PSJ.[3]

It is possible to derive optimal values for c and r, depending on the dataset characteristics—the same as those for PSJ—and in addition on the firing probabilities p of the hash functions. If hash functions fire for *many* values in the integer domain D, then each partition r_1^i will contain many signatures of r_1, leading to expensive joins. On the other hand, if the hash functions fire for only *few* elements, then the default partition r_2^0 will contain many elements of r_2, which leads to an expensive join $r_1^0 \bowtie_{\supseteq} r_2^0$ that may be almost as expensive as the original join problem $r_1 \bowtie_{\supseteq} r_2$.

For the following statements we consider two parameters that characterize the overall input data characteristics: $\rho = \frac{|r_1|}{|r_2|}$, the ratio of table cardinalities, and $\lambda = \frac{\theta_1}{\theta_2}$, the ratio of set cardinalities, indicated in the parameter overview of Table 4.2. Regarding the theoretical value for the comparison factor, Melnik and Garcia-Molina make the following theorem in Section 3.1 of [MGM03]:

THEOREM 4: *The optimal comparison factor of APSJ, given by*

$$c_{\text{APSJ}}^{\text{opt}} = 1 - \frac{k-1}{\lambda+k-1}\left(\frac{\lambda}{\lambda+k-1}\right)^{\frac{\lambda}{k-1}}$$

is achieved when the firing probability is

$$p = \left(\frac{\theta_1}{\theta_2 l + \theta_1}\right)^{\frac{1}{\theta_2 l}}.$$

No proof of this theorem was given by them. We show in Appendix A on page 159 how to arrive at this formula. The proof of the following theorem is also given in Appendix A.

THEOREM 5: *The optimal replication factor of APSJ is*

$$r_{\text{APSJ}}^{opt} = \frac{1}{\rho+1} + \frac{\rho}{\rho+1}\left(k-(k-1)\left(\frac{\lambda}{\lambda+k-1}\right)^{\frac{\lambda}{k-1}}\right).$$

Note that the comparison and replication factors of PSJ depend on the absolute value θ_1, while those for APSJ depend on the relative sizes λ and ρ.

[3] In this thesis, we do not deal with the case of empty sets in the description of the algorithms since it would complicate the presentation. See the brief discussion on a special case of this issue (when one of the *divisor* sets is empty) in Section 2.3.4.

EXAMPLE 6: For the hash functions defined in Figure 4.7, we can derive the following partitions of the dividend:

- $r_1^0 = \{\text{S1}, \ldots, \text{S12}\}$,
- $r_1^1 = \{\text{S1}, \text{S4}, \text{S5}, \text{S6}, \text{S11}\}$,
- $r_1^2 = \{\text{S1}, \text{S2}, \text{S3}, \text{S4}, \text{S8}, \text{S9}\}$,
- $r_1^3 = \{\text{S2}, \text{S3}, \text{S6}, \text{S7}, \text{S8}, \text{S12}\}$,
- $r_1^4 = \{\text{S1}, \text{S2}, \text{S3}, \text{S5}, \text{S7}, \text{S10}\}$, and
- $r_1^5 = \{\text{S1}, \text{S2}, \text{S10}\}$.

The partitions of the divisor are

- $r_2^0 = \{\text{green}\}$,
- $r_2^1 = \{\text{blue}, \text{red}\}$,
- $r_2^2 = \{\}$,
- $r_2^3 = \{\}$,
- $r_2^4 = \{\}$, and
- $r_2^5 = \{\text{yellow}\}$.

The factors c and r can now be derived as follows. The total number of comparisons is $\sum_{i=0}^{5} |r_1^i||r_2^i| = 12 \cdot 1 + 5 \cdot 2 + 6 \cdot 0 + 6 \cdot 0 + 6 \cdot 0 + 3 \cdot 1 = 25$, hence, $c = \frac{25}{48} = 0.5208\overline{3}$. The total number of signatures to be written into (and to be read from) a file is $\sum_{i=0}^{5} |r_1^i| + |r_2^i| = (12+1) + (5+2) + (6+0) + (6+0) + (6+0) + (3+1) = 42$. The replication factor is thus $r = \frac{42}{16} = \frac{21}{8} = 2.625$.

Note that in order to be more illustrative, we employ in this example arbitrarily defined hash functions and a different number of partitions compared to Example 5 for PSJ. Hence, it is not correct to compare the comparison and replication factors directly. Here, the comparison factor value is better than that in the PSJ example ($\frac{25}{48}$ vs. $\frac{2}{3} = \frac{32}{48}$). However, the replication factor value is worse than that for PSJ ($\frac{21}{8}$ vs. $\frac{7}{4} = \frac{14}{8}$).

Suppose, the data of the dividend and divisor had the properties stated in the assumptions for the theoretical values for c and r, given above. In our example, we have $k = 6$, $\rho = \frac{12}{4} = 3$, and $\lambda = \frac{5/2}{2} = \frac{5}{4}$. Hence, we would have a comparison factor of

$$c_{\text{APSJ}}^{\text{opt}} = 1 - \frac{6-1}{\frac{5}{4}+6-1}\left(\frac{\frac{5}{4}}{\frac{5}{4}+6-1}\right)^{\frac{\frac{5}{4}}{6-1}} = 1 - \frac{4}{5}\left(\frac{1}{5}\right)^{\frac{1}{4}} \approx 0.327$$

and a replication factor of

$$r_{\text{APSJ}}^{\text{opt}} = \frac{1}{3+1} + \frac{3}{3+1}\left(6-(6-1)\left(\frac{\frac{5}{4}}{\frac{5}{4}+6-1}\right)^{\frac{\frac{5}{4}}{6-1}}\right) = \frac{1}{4} + \frac{3}{4}\left(6-5\left(\frac{1}{5}\right)^{\frac{1}{4}}\right)$$

$$\approx 2.656.$$

In this example, the values that we computed manually differ from the theoretical values by $\frac{c-c_{\text{APSJ}}^{\text{opt}}}{c_{\text{APSJ}}^{\text{opt}}} = 59\%$ for the comparison factor and by $\frac{r-r_{\text{APSJ}}^{\text{opt}}}{r_{\text{APSJ}}^{\text{opt}}} = -1\%$ for the replication factor. The reason for the difference of the replication factor values is that the sets are not uniformly distributed accross the partitions due to the relatively small number of partitions and the relatively small datasets. A slight skew in the partitioning distribution caused a big difference. For example, it happended that for the *green* parts no hash function fires. Hence, they are assigned to the default partition r_2^0, leading to a summand $12 \cdot 1$ in the numerator of the replication factor quotient.

□

4.3 Set Containment Division Algorithms

We present several physical set containment division operators and discuss their performance characteristics. When we discuss the computational complexities of an algorithm, we use the parameters given in Table 4.2 on page 62.

The performance of an algorithm is influenced by the characteristics of the input data. Let r denote either the dividend or the divisor table. The size of r, denoted by $|r|$ is defined by two factors: the number of groups (sets) n and the average size of a group (set) θ, i.e., $|r| = n\theta$.

4.3.1 An Algorithm Template

Remember from Section 2.3.2 on page 9 that the logical set containment division operator has the dividend r_1 and divisor r_2 as input with the relation schemas $R_1(A \cup B)$ and $R_1(C \cup D)$, respectively. The attribute sets A and D play the role of a set identifier and the C and D attribute sets represent element values. One way to realize set containment division is to map the algebraic definition 8 of set containment division on page 11, namely

$$r_1 \div^* r_2 = \bigcup_{t \in \pi_C(r_2)} (r_1 \div \pi_B(\sigma_{C=t}(r_2))) \times (t)$$

directly to an algorithm, whose Java-style pseudo code is given in Algorithm 1. The idea is to divide the entire dividend table r_1 iteratively by a group of rows from the divisor table r_2 which have the same value for the columns C.

```
Cursor oldResult = null;
Cursor newResult = null;
Cursor projection = new Distinct(new Projection("D", new Cursor("r2")));

while (projection.hasNext()) {
  Row    cValue = projection.next();
  Cursor cTable = new SingleRowTable(cValue);
  Cursor quotient = new Division(
    new Cursor("r1"),
    new Projection(
      "B",
      new Selection(
        new Predicate(new RowComparator("C", cValue)),
        new Cursor("r2");
      )
    )
  );
  newResult = new Union(
    oldResult,
    new CartesianProduct(quotient, cTable)
  );
  oldResult = newResult;
  projection.next();
}

return newResult;
```

Algorithm 1: Set containment division pseudo code of the algorithm template

The pseudo code of the algorithm leaves open the question how the operators (division, projection, union, selection) are realized. Hence, it is only an *algorithm template*. Ideally, a query optimizer of an RDBMS picks an implementation that allows an optimal execution, depending on the actual optimization criteria like response time, throughput, or memory consumption. Many of the algorithms realizing the classical division operator have been discussed in Chapter 3.

The algorithm template for set containment division looks very inefficient at first sight. One may argue that if the dividend is larger than the main memory available, we cause considerable I/O for scanning the dividend for each divisor group. This is true if there is no index defined on the B columns of r_1. If we are lucky and there is such an index, the I/O could be significantly lower, especially if the access pattern is "local," i.e., if the dividend rows belonging to the B values we are searching for are stored consecutively on disk (clustered index) *and* we access the rows in the same order as they are stored on disk. In this case, the chances are that the desired rows are still in the buffer, assuming the typical *least recently used (LRU)* buffer replacement strategy is enabled. It is not unusual that an index is defined on $r_1.B$. Consider a typical data warehouse fact table like $\mathit{lineitem}(\mathit{order\#}, \mathit{line\#}, \mathit{part\#}, \mathit{supplier\#}, \mathit{date}, \mathit{quantity}, \ldots)$ from the TPC-H benchmark [Tra02], which is a detailed form of the transaction table $t^{\mathrm{v}}(t\#, i\#)$ used for the frequent itemset discovery problem, described in Chapter 5. If we consider the *lineitem* table as the transaction table t^{v} then the column *order#* is equivalent to the transaction column *t#* and *part#* corresponds to the item column *i#*, or briefly, $\mathit{lineitem}(\mathit{order\#}, \mathit{part\#}) \equiv t^{\mathrm{v}}(t\#, i\#)$. The *lineitem* table has indexes defined on all columns that refer to dimension tables. Here, we will have an index defined on each column *part#*, *supplier#*, and *date*.

Another problem occurs if the divisor is larger than main memory and we do not know about

its row ordering. In this case, it may become expensive to process the *Distinct* operator to find the duplicate-free collection of D values in r_2. Again, if we have an index on D or if we know that the data is grouped on D, then the optimizer may arrive at a much more efficient execution plan than the algorithm template suggests. For example, if the divisor is grouped on D, the optimizer might choose an "identity operator" as an implementation for *Distinct* that simply hands over each row from input to output without reordering it.

In summary, the idea of the algorithm template for set containment division is to use algorithms that can process data that is produced by algorithms called inside the algorithms. For example, one could choose algorithms that make no assumption on the ordering of input data: *Projection* would be implemented by an algorithm that removes duplicates. *Division* could be realized by hash-division. However, if *Projection* uses sorting to remove duplicates (instead of hashing) then one could use the merge-phase of sort-merge division for the division operator.

4.3.2 Merge-Sort Set Containment Division

This variation of set containment division, an "instance" of the template in Algorithm 1, requires the dividend r_1 be first grouped on A and second sorted on B and that the divisor r_2 is first sorted on D and second on C. Given this setting, the time complexity of Algorithm 1 becomes $O(|r_2| + \frac{|r_2|}{\theta_2}|r_1|)$, because we read the rows of the divisor table r_2 exactly once and for each of the $\frac{|r_2|}{\theta_2}$ divisors, we scan the dividend r_1 once.

4.3.3 Hash-Based Set Containment Division

Analogous to merge-sort set containment division, we can replace the generic division operation in the pseudo code template in Algorithm 1 by the hash-division algorithm.[4] The advantage of this approach is that we do not require the rows of a group to be sorted. Remember from Section 3.3.2.4 that hash-division maintains two hash tables, one for storing the rows of the divisor and the other for the quotient candidates. First, the divisor is scanned once to build the divisor hash table and then the dividend is scanned once to populate the quotient hash table and to set bits in the quotient hash tables' bitmaps. This causes a worst case memory complexity of $O(|r_1||r_2'|)$, where r_2' is a classical divisor table, not a table of more than one divisor group.

The time complexity of Algorithm 1, when division is replaced by hash-division, is the same as for merge-sort set containment division.

An optimization of this approach is possible if a large main-memory is available. We can build several divisor hash tables in memory by scanning over more than one divisor group consecutively. Then, we build and update the quotient hash tables of each separate division operation in parallel while scanning the dividend once, i.e., the *best-case* time complexity is $O(|r_1| + |r_2|)$. If only a subset of the $\frac{|r_2|}{\theta_2}$ division operations can be executed in parallel during a single scan

[4] Alternatively, transposed hash-division (Section 3.3.2.5), an algorithm that also does not make any assumption on the sortedness of the inputs, can be employed instead of hash-division. Refer to Table 3.3 on page 48 for a complete list of optional algorithms.

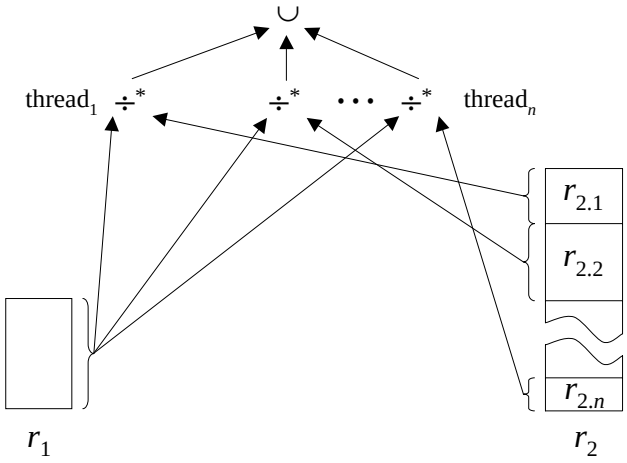


Figure 4.8: Parallel threads executing set containment division

of r_1, we need more than one scan, say k iterations, yielding a computational complexity of $O(k|r_1|+|r_2|)$.

4.4 Parallel Execution of Set Containment Division

The naïve approach specified in Algorithm 1 executes the division operations sequentially. To improve the response time, we could execute some or all division operations in parallel and then merge the quotients asynchronously. The RDBMS can decide on an optimal degree of parallelism to distribute the work to threads of execution.

The parallelization of a query execution plan typically requires that the tables are partitioned into several smaller tables. In this section, we only consider horizontal partitioning, where entire rows of a table are stored in a partition.[5] The idea of partitioning is illustrated in Figure 4.8, where each thread of execution (e.g., a process or thread of an operating system) processes some part of the set containment division. Here, only the divisor r_2 is divided into several disjoint partitions and the entire dividend is the input of each thread. The top union operator merges the disjoint quotient rows into the result table.

There are several options for partitioning the dividend and divisor. Table 4.3 gives an overview of partitioning classes. A plus sign indicates that the respective table is partitioned on the given set of columns, a minus sign indicates that no partitioning was made on the column set. A subscript number indicates on which columns the table was partitioned first and second (similar to the grouping column ordering in Table 3.1 on page 34). An asterisk in columns $r_1.B$ and $r_2.C$

[5]Vertical partitioning, rarely used in practice, stores the table columns in different partitions. For example, consider a table *citizen* that stores the name, address, passport photo, and fingerprint scan images of a person in a table. If some I/O critical queries need access to the personal record data only, it may be beneficial to store these columns separately from the image data. However, queries affecting both types of columns require to join the rows of the partitions, which may be expensive.

Class	Dividend r_1		Divisor r_2		Description
	A	B	B	C	
1	$-$	$-$	$-$	$+$	
2	$-$	$-$	$+$	$-$	
3	$-$	$-$	$+_2$	$+_1$	Divisor first partitioned on C, second on B
6	$-$	$+^*$	$+^*$	$-$	Dividend and divisor partitioned using the same function
7	$-$	$+^*$	$+^*_2$	$+_1$	Divisor first partitioned on C, second on B, and dividend and divisor partitioned using the same function for B
8	$+$	$-$	$-$	$-$	
9	$+$	$-$	$-$	$+$	
14	$+_1$	$+^*_2$	$+^*$	$-$	Dividend first partitioned on A, second on B, and dividend and divisor partitioned using the same function for B
15	$+_1$	$+^*_2$	$+^*_2$	$+_1$	Dividend and divisor first partitioned on A and C, respectively, and second on B using the same partitioning function for B

Table 4.3: Partitioning classes enabling a parallelization of set containment division

shows that the same partitioning function is used for both columns. For example, the tables r_1 and r_2 in Figure 4.9(a) and (b) may be partitioned into two partitions where one partition holds all rows with odd values and the other with even values of column *b* and *c*, respectively.

A partitioning class number is the value that we get when we interpret the sequence of minus and plus signs as a binary number, where minus and plus correspond to digit 0 and 1, respectively. For example $(A,B,C,D) = (-,+,+,-)$ corresponds to class $0110_2 = 6_{10}$.

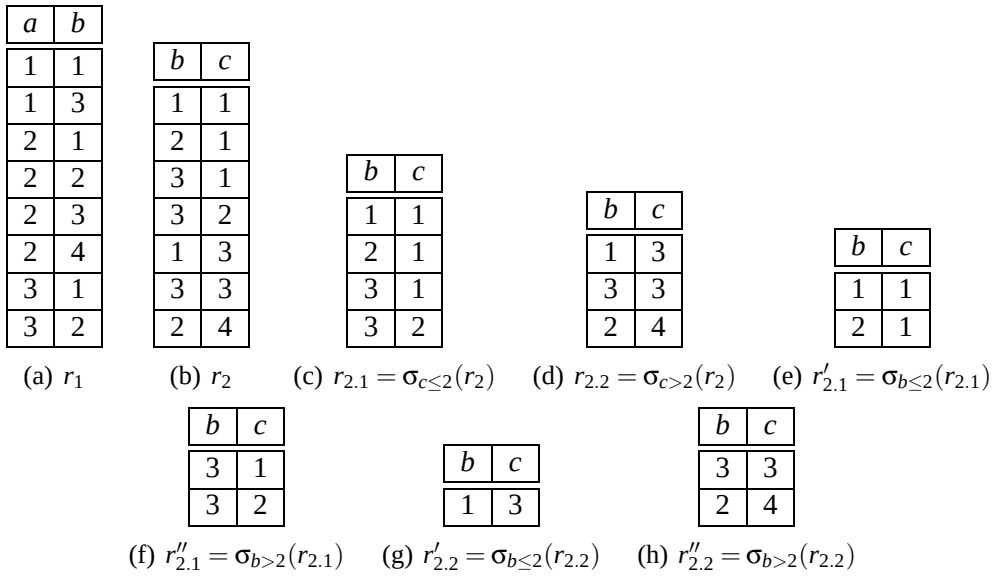
No horizontal partitioning options other than the ones shown in Table 4.3 enable a parallelization strategy for set containment division. Table 4.4 shows the remaining classes that are possible but that do not contribute to a parallelization strategy for set containment division. For example, partitioning class 4 does not help since we partition only on the B columns of the dividend. This means that we compare a *subset* of a quotient candidate of r_1 against *all* values of a divisor group, which is incorrect. The reason is that there is only one law involving a partitioning of the dividend (Law 2 on page 17), which requires a partitioning on the A columns, but not on the B columns alone.

Consider, e.g., partitioning class 1, where r_2 is partitioned on C and no other partitioning is given. Let us investigate how we can exploit this partitioning. One can distinguish between two types of execution strategies, a static one and a dynamic one. The static strategy assigns a fixed partition of r_2 to each thread of execution. All partitions are disjoint and span a set (or range) of B values of r_2. In the dynamic execution strategy, we use a partitioning operator that returns an entire group, i.e., a sequence of rows having the same B value. Each call to the partitioning operator returns a different (the "next") group. This approach guarantees that none of the threads becomes idle and hence allows a good response time.[6]

[6]The dynamic approach is good on the average but not necessarily optimal. Suppose, the degree of parallelism is two and r_2 consists of a sequence of four groups of rows (g_1, g_2, g_3, g_4), where g_1 and g_3 take 2 time units to finish the division and g_2 and g_4 take 3 units. The dynamic assignment leads to the sequence (g_1, g_3) for thread 1 and (g_2,

Class	Dividend r_1		Divisor r_2		Description
	A	B	B	C	
0	$-$	$-$	$-$	$-$	
4	$-$	$+$	$-$	$-$	
5	$-$	$+$	$-$	$+$	
10	$+$	$-$	$+$	$-$	
11	$+$	$-$	$+_2$	$+_1$	Dividend partitioned on A, and divisor first partitioned on D, second on C
12	$+_1$	$+_2$	$-$	$-$	Dividend first partitioned on A, second on B
13	$+_1$	$+_2$	$-$	$+$	Dividend first partitioned on A, second on B, and divisor partitioned on C

Table 4.4: Partitioning classes disabling a parallelization of set containment division



(a) r_1 (b) r_2 (c) $r_{2.1} = \sigma_{c\leq 2}(r_2)$ (d) $r_{2.2} = \sigma_{c>2}(r_2)$ (e) $r'_{2.1} = \sigma_{b\leq 2}(r_{2.1})$

(f) $r''_{2.1} = \sigma_{b>2}(r_{2.1})$ (g) $r'_{2.2} = \sigma_{b\leq 2}(r_{2.2})$ (h) $r''_{2.2} = \sigma_{b>2}(r_{2.2})$

Figure 4.9: Example partitions of partitioning classes 1 and 3

Not only can the set containment division be parallelized by partitioning r_2 according to the C columns, thus defining separate divisor groups and executing each division in parallel to other divisions. Also the division operation itself can be parallelized by partitioning r_2 on the C columns, i.e., on the elements that we test whether they are contained in a group of r_1 defined by A. This option was described by Law 1 on page 16, represented by partitioning class 3 in Table 4.3.

To illustrate the idea of employing both parallelization approaches, let us assume the dividend and divisor tables in Figures 4.9(a) and (b). First, let us consider partitioning class 1, where the

g_4) for thread 2. The total execution time is hence $\max\{2+2, 3+3\} = 6$ units. An optimal assignment would have been (g_1, g_2) and (g_3, g_4), respectively, leading to a total of $\max\{2+3, 2+3\} = 5$ units.

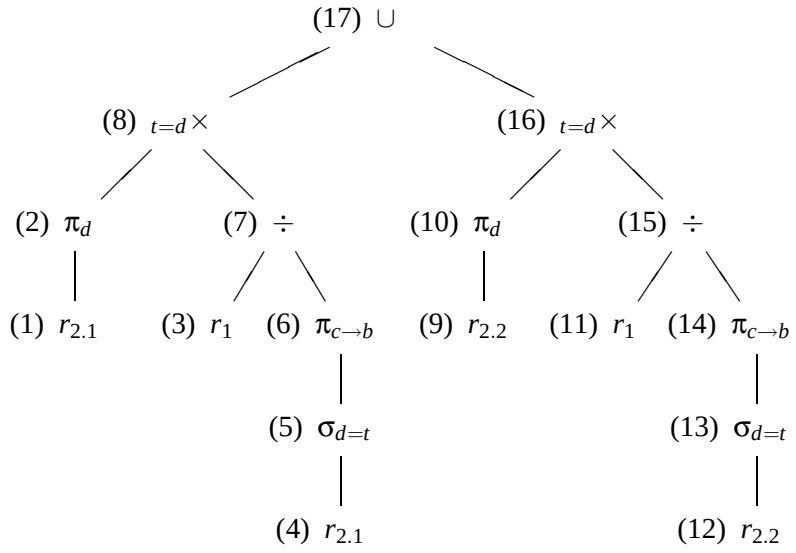


Figure 4.10: Query execution plan of a parallel set containment division based on partitioning class 1

divisor is partitioned on column *c*, i.e., the divisor groups are distributed into separate tables. In our example, we partition the values of *c* into the two intervals $(-\infty, 2]$ and $(2, \infty)$, as illustrated in Figures 4.9(c) and (d). Assuming a degree of parallelism of two, the query execution plan has the structure shown in Figure 4.10. This degree of parallelism is reflected by the binary union (operator 17). The plan contains a correlated version of a Cartesian product (operators 8 and 16), which evaluates for each row of the right input the left sub-plan. It uses a variable *t* that is bound to the *c* value of a partition of r_2.

The second, more fine-grained parallelization approach based on partitioning class 3 employs pipelining during the execution of a each separate division. In addition to the partitioning on column *c*, we partition the divisor also on *b*, i.e., we partition r_2 on the value pairs (c, b). Using an interval notation, we partition r_2 into four partitions according to the interval pairs $((-\infty, 2], (-\infty, 2])$, $((-\infty, 2], (2, \infty))$, $((2, \infty), (-\infty, 2])$, and $((2, \infty), [2, \infty))$, as illustrated in Figures 4.9(e)–(h). Given these partitions, the query execution plan becomes more complex, as shown in Figure 4.11. The difference between the two plans is the application of Law 1: We replace $r_1 \div (r'_{2.1} \cup r''_{2.1})$ by $(r_1 \ltimes (r_1 \div r'_{2.1})) \div r''_{2.1}$ and, analogously, $r_1 \div (r'_{2.2} \cup r''_{2.2})$ by $(r_1 \ltimes (r_1 \div r'_{2.2})) \div r''_{2.2}$.

To make the processing more intuitive, Figure 4.12 shows some of the intermediate results during the execution of the plan in Figure 4.11. For the operators that are contained in the left sub-plan of the correlated Cartesian product (operators 14 and 28), we give the results after each iteration of the correlation variable *t* that picks a single row of the right input. For example, the first row value of *t* in operator 14 is (1). This value is taken as input for the selection predicate of operator 6, which leads to two result rows produced by operator 8, namely rows (2) and (3). The final result of the execution is the table produced by operator 29, given in Figure 4.12(i). It

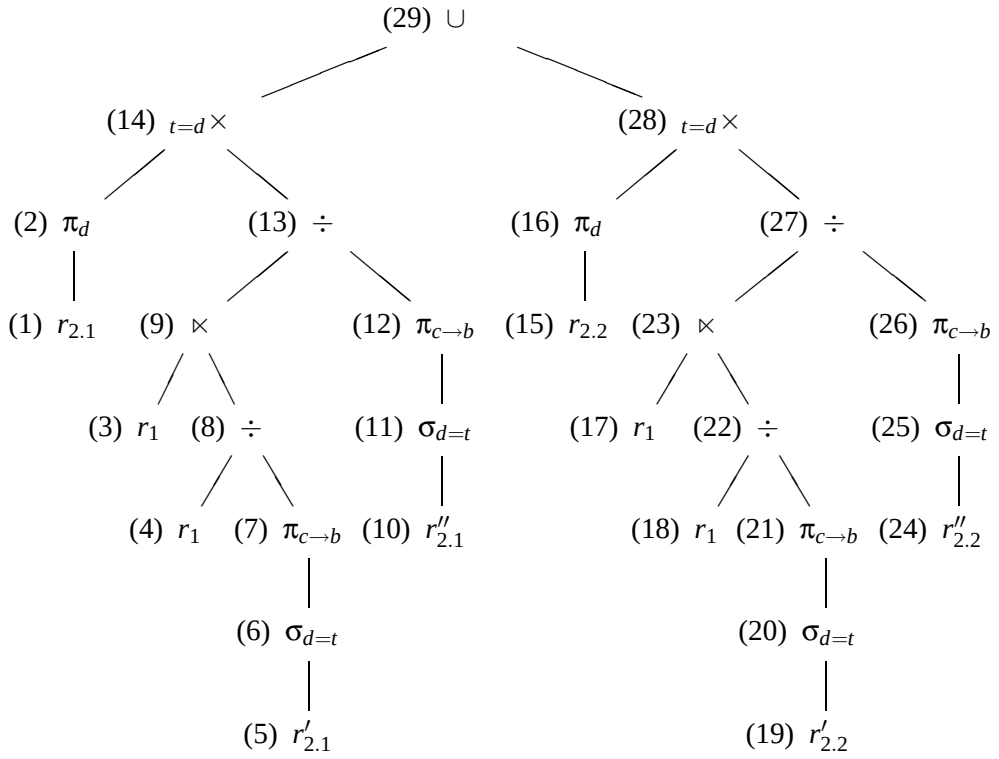


Figure 4.11: Query execution plan of a parallel set containment division based on partitioning class 3

is the same result as that of operator 17 in the plan in Figure 4.10 and, of course, it is equal to $r_1 \div^* r_2$.

4.5 Set Containment Division Using a Subset Index

In this section, we define a model to represent the fact that some sets are subsets of other sets, given a collection of sets. This model is the basis for an in-memory data structure to improve the cost of containment tests for set containment division algorithms.

4.5.1 The Subset Graph

Let r be a relation with the schema $R(A \cup B)$, where A and B are disjoint nonempty sets of attributes. In the following, we assume that A represents the set identifying attributes and B the element identifying attributes like r_1 in the definition of set containment join in Section 2.3.2 on page 9. We first formally describe the fact that there are groups (subsets of tuples) in r, each having a different value of the attributes A. This is achieved by the group operator ${}_G\gamma_F$, listed in the operator overview in Table 2.3 on page 7. Here, $G = \{A\}$ and no aggregate functions are applied, i.e., $F = \emptyset$. If $r \neq \emptyset$ then we can partition r into $n \geq 1$ groups $g_1, \ldots, g_n$, where

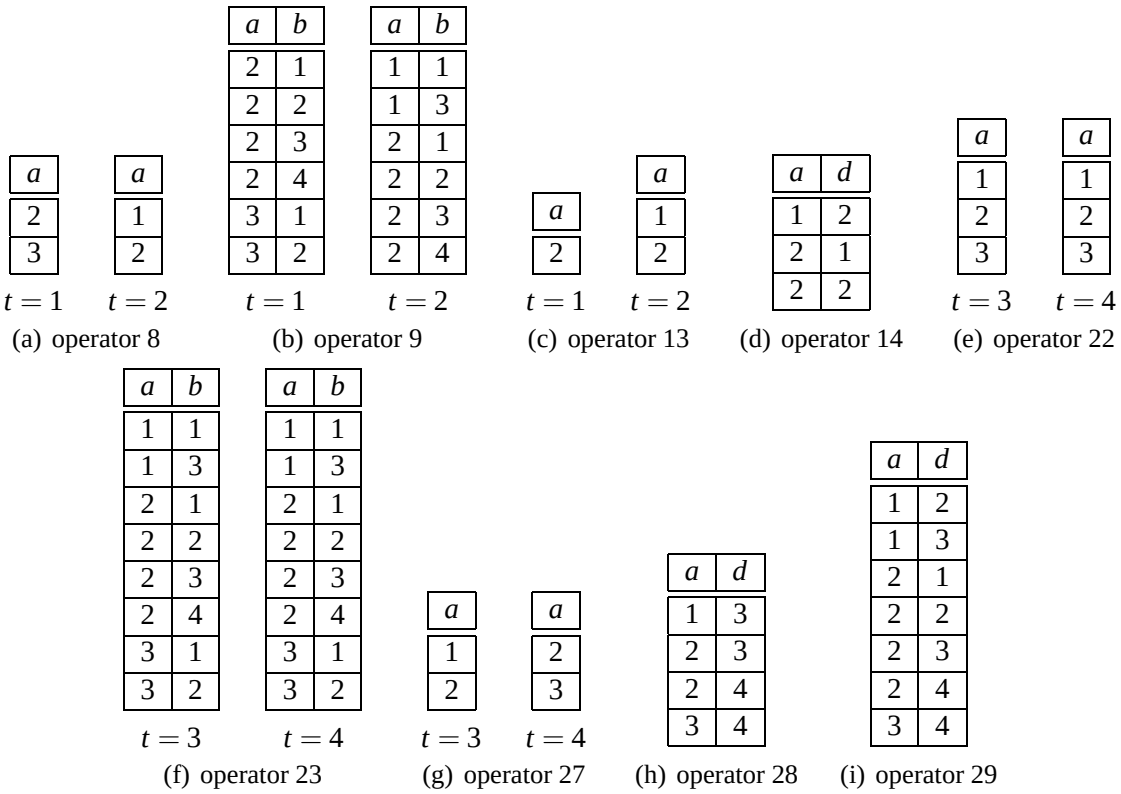


Figure 4.12: Intermediate results of the query execution plan in Figure 4.11 based on partitioning class 3

1. $\forall i \in \{1,\ldots,n\} : g_i \neq \emptyset$,
2. $\bigcup_{i=1}^{n} g_i = r$,
3. $\forall i \in \{1,\ldots,n\} \;\; \forall j \in \{1,\ldots,n\} : (i \neq j) \rightarrow (g_i \cap g_j = \emptyset)$, and
4. $\forall t_1, t_2 \in r \;\; \exists i \in \{1,\ldots,n\} \;\; \exists j \in \{1,\ldots,n\} : (t_1 \in g_i \wedge t_2 \in g_j \wedge t_1.A = t_2.A) \rightarrow (i = j)$.

Conditions 1–3 state that r is partitioned into disjoint nonempty sets and condition 4 requires that each group contains only tuples with the same value of attributes A and no pair of groups has the same value of A.

We now define a directed acyclic graph (DAG) on r, called *subset graph of* r, where each group $g_i \in r$ defined by A is a vertex and there is an edge from vertex v_1 to v_2 if and only if the values of the attribute set B of the group representing v_1 are a proper subset of those of v_2. A subset graph $G_{\subset}(r) = (V,E)$, where V is a set of vertices and E is a set of edges, is defined as follows:

$V = \{g_1, ..., g_n\}$, as defined above, and
$E = \{(g_i, g_j) \in V \times V \mid \;\; \exists i \in \{1,\ldots,n\} \;\; \exists j \in \{1,\ldots,n\} :$
$\pi_B(g_i) \subset \pi_B(g_j) \wedge \nexists k \in \{1,\ldots,n\} : g_k \in V \wedge (g_i, g_k) \in E \wedge (g_k, g_j) \in E\}$.

In other words, there is an edge $g_i \rightarrow g_j$ between two groups of r if and only if $\pi_B(g_i) \subset \pi_B(g_j)$ and if there is no other group g_k such that $\pi_B(g_i) \subset \pi_B(g_k) \subset \pi_B(g_j)$, i.e., transitive containments are *not* represented by an edge, only direct containments.[7] In Figure 4.13, we sketch a set containment division example. The set identifying columns in r_1 and r_2 are the sets $A = \{a\}$ and $D = \{c\}$, respectively. Hence, the remaining column set $B = \{b\}$ identifies the elements of a set. Figure 4.14 shows a subset graph for the divisor table in Figure 4.13(b).

The benefit of the subset graph can be summarized as follows. On the one hand, if we know that the B values of certain group g defined by a value A of the dividend are contained in the set of B values of a group g' of the divisor's subset graph, then we know that all subsets of $g.B$ are also contained in $g'.B$. On the other hand, we can employ a special case of the *Apriori property*, introduced for frequent itemset discovery and further discussed in Section 5.6, saying that if a set $g'.B$ of the divisor is not contained in $g.B$ then no superset of $g'.B$ is contained in $g.B$ neither. The Apriori property states that a k-itemset can only be frequent if all $(k-1)$-subsets are frequent, too. One can consider our idea as applying the Apriori property to sets in general and setting the minimum support threshold to 1.

4.5.2 Compression

The representation of the set elements in the set containment graph is straightforward because all element values are listed. A more compact representation is possible. Instead of keeping all elements in a vertex, we store only the elements that are *not* already stored in any subset vertex. Hence, the vertices that have *no* subset store all elements like in the uncompressed subset graph. The same idea has been suggested for other data structures, like the *frequent pattern tree* [HPY00] for the frequent itemset discovery. However, although our subset graph bears some resemblance to the frequent pattern tree, the subset graph is not always a tree and, therefore, the compression is slightly different. Compressing the representation of sets allows us to build an index for a larger table than with an uncompressed representation while still fitting into main memory.

Consider for example vertices 4 and 9 in Figure 4.14. It suffices to store the common elements in the subset vertex and the remainder in the superset vertex yielding the sets $\{1\}$ and $\{2,6\}$, as illustrated in Figure 4.15. Looking at vertex 12, we find that all its direct supersets do not need to store the subset $\{3\}$, therefore it is eliminated in the compressed subset graph. Similarly, vertex 3 does not need to store any element because all of them are contained in one ore more of its direct subsets. In general, a vertex represents the set of elements that we get when we merge its own elements recursively with all elements of its subset vertices.

We have implemented an uncompressed and a compressed version of the subset index, which is based on the subset graph. This data structure will be discussed in the following section.

[7] The subset graph is equal to a *Hasse diagram* [PS03], where the relation is the proper subset operator ($\subset$). Given a graph, a Hasse diagram eliminates all self loops and all edges that are implied by transitivity. In contrast to our subset graph, the edges in a Hasse diagram are undirected.

a	*b*
1	1
1	2
1	4
1	5
1	8
2	2
2	3
2	4
2	5
3	2
3	3
3	4
5	1
5	4
5	7
4	1
4	2
4	6
10	4
10	5
7	3
7	4
9	2
9	6
8	2
8	3
6	1
6	3
11	8
12	3

(a) r_1

b	*c*	*card*
1	1	5
2	1	5
4	1	5
5	1	5
8	1	5
2	2	4
3	2	4
4	2	4
5	2	4
2	3	3
3	3	3
4	3	3
1	5	3
4	5	3
7	5	3
1	4	3
2	4	3
6	4	3
4	10	2
5	10	2
3	7	2
4	7	2
2	9	2
6	9	2
2	8	2
3	8	2
1	6	2
3	6	2
8	11	1
3	12	1

(b) r_2

a	*c*
1	1
1	10
1	11
2	2
2	3
2	8
2	7
2	12
3	3
3	8
3	7
3	12
5	5
4	4
4	9
10	10
7	7
7	12
9	9
8	8
8	12
6	6
6	12
11	11
12	12

(c) $r_3 = r_1 \div^* \pi_{b,c}(r_2)$

Figure 4.13: Example tables for set containment division

4.5.3 The Subset Index

It is straightforward to realize the subset graph as a lookup data structure that allows finding all subsets and supersets of a given set. The motivation for this index is the situation where either the dividend or the divisor is much larger than main memory and the other table fits into memory. We want to build an in-memory data structure for the smaller table and probe it while scanning the larger, disk-resident table once.

Given a dividend table r_1 and divisor relation r_2 with schemas $R_1(A \cup B)$ and $R_2(B \cup C)$, re-

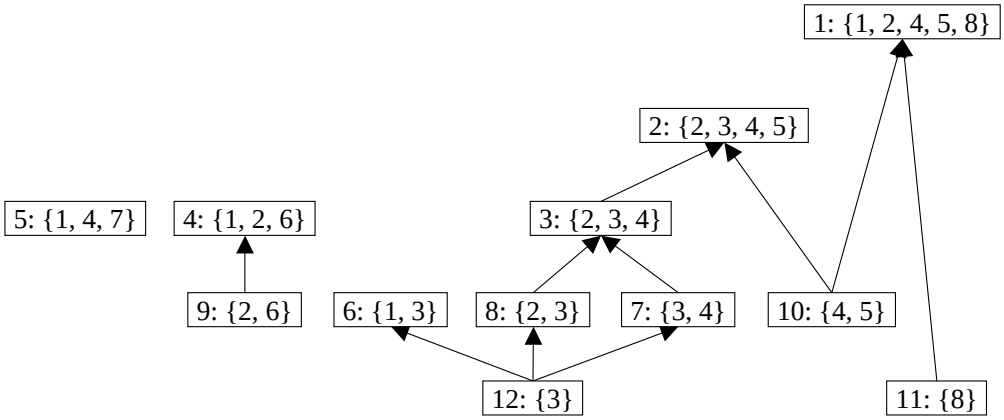


Figure 4.14: Uncompressed subset graph $G_{\subset}(r_2)$ built for divisor table r_2 in Figure 4.13(b)

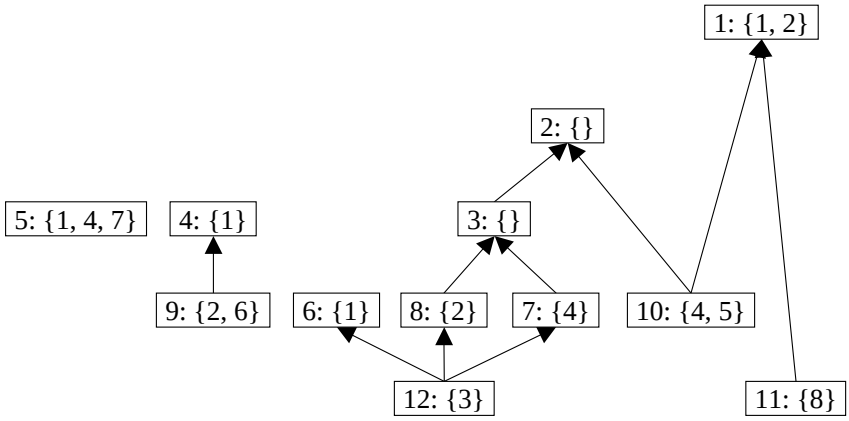


Figure 4.15: Compressed subset graph that is equivalent to that in Figure 4.14

spectively, where $A = \{a_1, \ldots, a_m\}$, $B = \{b_1, \ldots, b_n\}$, and $C = \{c_1, \ldots, c_o\}$ are nonempty column sets. In the following, we assume that the table r_1 has more rows than the r_2 table and that r_2 can fit into main memory. If r_2 had more rows than r_1 and if r_1 could fit into main memory, we would build a subset index on r_1, as we will discuss in Section 4.5.6.

The following descriptions refer to the pseudo code samples of Algorithms 20 and 21 in Appendix B on pages 177–178. We define a data structure called *subset index* as follows: A subset index $I_{\subset}(r_2)$ is a subset graph for a table $r_2(B \cup C)$, where C is the list of columns representing the set identifier and B is the list of columns representing the element identifiers. In the subset index, a vertex of the subset graph is called a *node*. A node has a flag called *isMarked*, whose value can be set to *true* to indicate that the node has been visited before when traversing the index. A similar flag called *isInResult* is used to indicate that the elements of a node belong to a quotient and should not be added more than once to a set. This flag is only needed when the subset index is compressed. A node is represented by an object of the class *SubsetIndexNode*, as specified

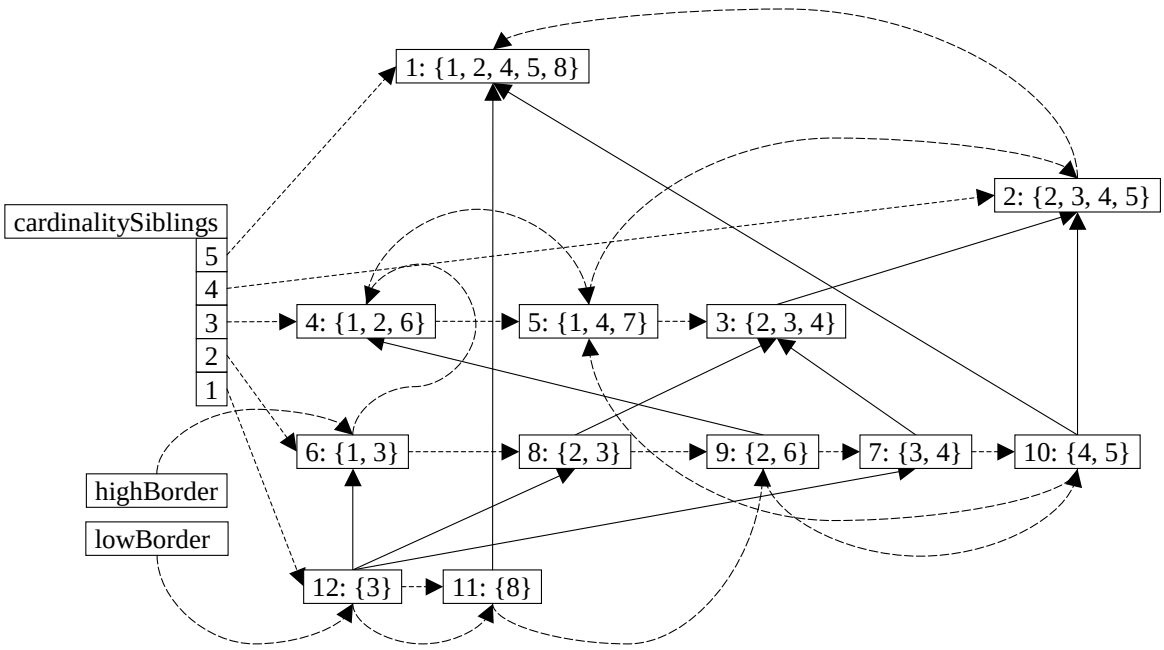


Figure 4.16: Uncompressed subset index $I_{\subset}(r_2)$ built for divisor table r_2 in Figure 4.13

in the Java-style pseudo code in Algorithm 20 in Appendix B on page 177. The *set* attribute is the D value of a divisor group in r_2. The *subsets* attribute represents the collection of edges in the subset graph from a superset to a direct subset according to the Hasse diagram definition mentioned before, i.e., not *all* possible subsets have an edge to a superset. The *elements* attribute is a collection of rows, each row representing a C value of a row in r_2.

The subset index superimposes the subset graph by further data structures that we call *cardinalitySiblings*, *lowBorder*, *highBorder*, and *markedNodes*. The object *cardinalitySiblings* is an array of sorted lists. The ith list contains all nodes of cardinality i. The array is used to restrict the number of nodes that have to be visited when a new node is added to the index: When a node n of cardinality c is added to the index, we first check all nodes of size $c+1$ if they contain n. If yes, we establish an edge from n to each of these superset nodes. When we test a node, we mark it and all its subset nodes as "visited." Thereby, we avoid establishing an additional edge between n and a subset of a node, which would violate the subset graph property that there is an edge only between "direct" subsets. Then, we check all nodes of size $c+2$, $c+3$, and so on, until we have checked all nodes of a cardinality that is higher than that of n. The subset index's data structures as well as the logic of adding a node is shown in the pseudo code in Algorithm 21 in Appendix B on page 178.

Another data structure, *lowBorder*, is a list of all nodes that have no subsets. Analogously, the data structure *highBorder* comprises all nodes that have no supersets. In Figure 4.16, *lowBorder* consists of the nodes 12, 11, 9, 10, and 5, while *highBorder* contains nodes 6, 4, 5, 2, and 1. Node 5 is contained in both lists because it stores a singleton set, i.e., it has neither a subset nor a superset. The arrows connecting the *lowBorder*/*highBorder* nodes start and end in the bottom/top

of a node box, respectively.

During the index creation and index probing phase, we mark index nodes to remember if they have been visited before. This is needed because, as mentioned before, the subset index is not necessarily a tree, i.e., we can reach a node via different paths. For example, in Figure 4.16, node 3 can be reached from node 12 either via node 7 or node 8. When we add a new node during index creation or when we probe the index against a node, we first need to unmark the nodes that have been marked previously. By maintaining a list of marked nodes called *markedNodes* while traversing the graph, we avoid scanning the entire index for marked nodes when we want to reset all nodes to unmarked.

4.5.4 Index Creation

We make the following assumptions on the structure of the divisor table r_2:

- Its table schema is $R_2(B \cup C \cup \{card\})$, i.e., the divisor has an additional column, *card*, that indicates the number of rows, belonging to a group defined by the columns D, i.e., the group cardinality. This column could be a derived column that was created by an aggregation operator below the subset containment operator in the query execution plan.

- It is first *sorted* on *card* in descending order, second *grouped* on C, and third, *sorted* on B in ascending order.

The *card* column is merely used to sort r_2 such that the largest groups occur with decreasing size when we fetch rows from the table. Figure 4.13(b) illustrates such a table with the required properties.

In the following, we ignore the case that duplicate divisor *groups* are contained in r_2. A duplicate group occurs in r_2 if there are two values $c_1 \neq c_2$ such that $\pi_B(\sigma_{C=c_1}(r_2)) = \pi_B(\sigma_{C=c_2}(r_2)) \neq \emptyset$. However, it is straightforward to deal with this case as well by extending the data structure and index population logic slightly. Furthermore, we ignore the case of duplicate *rows* in r_2.

The subset index is built by first calling the constructor of the *SubsetIndex* class and then fetching the rows of r_2, transforming a group into a node and then adding the node to the index, as shown in Algorithm 2 on page 81.

All these descriptions are illustrated by Figure 4.16, which represents the result of a subset index creation for the divisor table r_2 in Figure 4.13(b). The first group that is processed has group value $c = 1$ and b values 1, 2, 4, 5, and 8. Since no other node is contained in the index, we can simply add it by linking it to *cardinalitySiblings*[5]. The second group ($c = 2$, $b \in \{2,3,4,5\}$) is added to the index like the first node. The third node ($c = 3$, $c \in \{2,3,4\}$) is more interesting. We first add it to the empty *cardinalitySiblings*[3]. Then, we check all nodes in $I_{\subset}(r_2)$ that have a higher cardinality, i.e., nodes 1 and 2, if they are a superset of node 3. We start with cardinality 4, where we find node 2. We find out that node 3 is a subset of node 2 and therefore, we mark node 2 as well as all its supersets (in this case none), and we add node 3 to the list of children of node 2. The subset test between nodes 3 and 1 yields false, hence we are done with node 3.

```
Table       cursor1     = new Table("r2");
SubsetIndex subsetIndex = new SubsetIndex(10);

if (cursor1.hasNext()) {
  boolean isNextGroup = false;
  Row row = (Row) cursor1.next();
  SubsetIndexNode node = null;

  // Process divisor table group-by-group.
  do {
    Row firstRow = row; // The first row of a group.
    node = new SubsetIndexNode((Row) firstRow.getObject("C"));
    node.addElement((Row) firstRow.getObject("B"));
    isNextGroup = false;

    // Process group row-by-row.
    while (cursor1.hasNext() && !isNextGroup) {
      row = (HashCodeArrayTuple) cursor1.next();
      // Check if this row still belongs to the same group as
      // the first row of the group.
      isNextGroup = (((Row) row.getObject("C")) !=
                     ((Row) firstRow.getObject("C"));
      if (isNextGroup)
        // Finally, we add the node to the index.
        subsetIndex.add(node);
      else
        node.addElement((Integer) row.getObject("B"));
    }

    // Do not forget to add the last group of the table.
    if (!cursor1.hasNext())
      subsetIndex.add(node);
  } while (isNextGroup);
}
```

Algorithm 2: Creation of a subset index on the divisor table in pseudo code

The advantage of processing top-down from the largest to the smallest sets helps during the insertion of a node because, given a node *n*, we have to check only which *subsets* of *n* exist in $I_{\subset}(r_2)$ and not also for *supersets* of it.

The data structure *cardinalitySiblings* is merely used for an efficient insertion of divisor sets into the subset index, it is no longer needed for probing. However, the *lowBorder* data structure is important for the probing phase.

4.5.5 Index Probing

Having created the subset index $I_{\subset}(r_2)$ on the divisor table, we probe each group of the dividend table against the index in order to find the quotients. To make the processing easier, we assume that the dividend r_1 is grouped on *A*. For example, the dividend table r_1 in Figure 4.13(a) has the same contents as the divisor table r_2 except that is has no *card* column. It is grouped on the set-identifying column *a* like the divisor is grouped on the set-identifying column *c*. In contrast to the index creation phase, we do without the *card* column because the probing phase does not require the input table be ordered by group cardinality. Therefore, the ordering of the groups in r_1 in Figure 4.13(a) is arbitrary.

We compute the quotient table r_3 as follows. We scan r_1 group by group. For each group, we build a node n in the same way as for a divisor group. We start by checking each index node in the *lowBorder* list. When an index node i is tested against n, i first checks if it has been visited before. If yes, nothing needs to be done. Otherwise, i marks itself as "visited," checks if its elements are contained in n, and recursively tests its superset nodes. If i finds that its elements are contained in n, it returns its *set* attribute value. The value pair $(n.set, i.set)$, which corresponds to $(r_1.a, r_2.c)$, is then transformed into a row and added to the quotient table.

Figure 4.13(c) illustrates an example quotient table that was produced by probing the index for each group defined by $r_1.a$. Since the dividend and divisor have the same content (except for the *card* column), the quotient describes the *transitive closure* of $G_{\subset}(r_2)$. The transitive closure is the collection of vertex pairs (v_1, v_2) such that v_1 can be reached from v_2 along the directed edges in $G_{\subset}(r_2)$. Here, v_1 and v_2 correspond to a and d, respectively. For example, since there is a path in the subset graph from vertex 12 to 2 via the vertices 7 (or 8) and 3, we find row $(2, 12)$ in the quotient table. In addition, for each vertex v of $G_{\subset}(r_2)$, there is a row (v, v) in r_3, since each group v occurs in both input tables of the set containment division and set *equality* is a special case of set *containment*.

4.5.6 Implementation Details

The *lowBorder* and *highBorder* data structures are so-called *TreeSets*, provided by the Java class library. A *TreeSet* is an ordered set of elements and guarantees a $\log(n)$ time for the add, remove and containment test operations, where n is the number of elements. This is helpful when we probe the index. We define the natural ordering of nodes in a *TreeSet* first by cardinality and for equal cardinality nodes by element values, starting with the first element. This ordering can be seen in Figure 4.16: *lowBorder* has node 12 as the smallest and node 5 as the largest element. We assume the elements of a set are sorted. Hence, given a dividend node n, we can stop searching for index nodes in the *lowBorder* object as soon as we find an index node $i > n$.

In this section, we have described the case, where the divisor has fewer rows than the dividend and the divisor (as an index $I_{\subset}(r_2)$) fits into main memory. Of course, our approach can equally be applied for the symmetric case, where the dividend is smaller than the divisor and the dividend's subset index $I_{\subset}(r_1)$ fits into main memory. In this case, we build the subset index in the same way as we would do for the divisor. However, the index probing phase starts with nodes in the *highBorder* object and recursively checks which index node is a *superset* of the given divisor node.

4.5.7 Complexities

In this section we discuss the time and space complexities of subset index set containment division. In the following, we assume that we have created a subset index on the divisor table r_2 and that it is probed with the dividend table r_1. Let $s_1 = \frac{|r_1|}{\theta_1}$ and $s_2 = \frac{|r_2|}{\theta_2}$ be the number of sets in r_1 and r_2, respectively. Remember from Table 4.2 on page 62 that θ_1 and θ_2 are the average set cardinalities in r_1 and r_2, respectively.

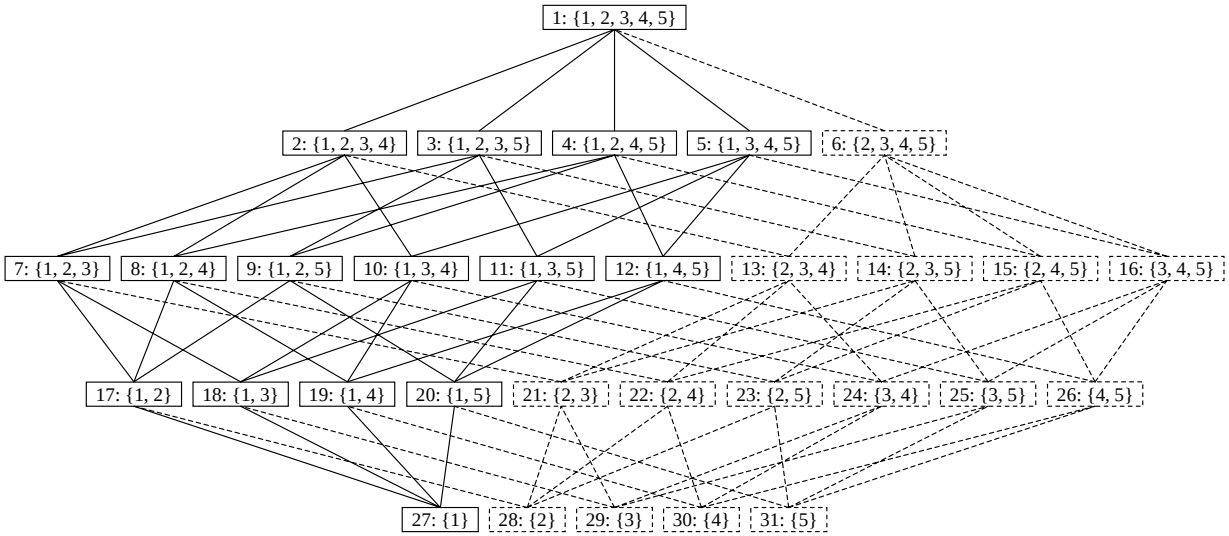


Figure 4.17: Subset graph for the powerset of $\{1,2,3,4,5\}$, excluding the empty set

Let us first consider an uncompressed subset index. The build and probe time complexity is the time complexity of the build and probe phase, respectively. The best best case is when all sets have a different cardinality and the sets have no overlap like in $\{\{5\},\{2,7\},\{1,3,4,6\}\}$. In this case, at most one set is stored in each entry of the *cardinalitySiblings* array (some entries may remain empty). Thus, when the index is probed with a single dividend set, only one set comparison is required, yielding a total probe time complexity of $O(s_1)$ and a build time complexity of $O(s_2)$. The space complexity is $O(s_2)$.

The worst case is more complex. It occurs when the sets represent the powerset of the largest set, excluding the empty set. It has $s_2 = 2^n - 1$ nodes, where $n = \log(s_2 + 1)$ is the size of the largest set. For example, consider the powerset of $\{1,2,3,4,5\}$ without the empty set. The corresponding subset graph is illustrated in Figure 4.17. Here, the largest set is $\{1,2,3,4,5\}$ and $n = 5$. To avoid overloading the figure, we show the subset *graph* instead the subset index with its auxiliary data structures and we omit arrowheads.

Our second assumption for the worst case is that all sets of the dividend r_1 are a superset of the largest set of the divisor r_2. A probe of the subset index on r_2 with a single set of r_1 works as follows. First, we try to find a match in the *lowBorder* data structure, which is $O(log(n))$ because we use a *TreeSet* for the implementation, which has a logarithmic search complexity, and the number of elements in *lowBorder* is n. Then, we have to visit all superset nodes of the current node. The number of these nodes is the number of nodes in the original graph minus the size of the subset graph for the powerset of the set that is reduced by the current element, i.e., $(2^n - 1) - (2^{n-1} - 1) = 2^{n-1}$. In Figure 4.17, the straight-lined boxes comprise the supersets of set $\{1\}$. Here, the number of nodes to be visited is $2^4 = 16$.

A node can be visited only from unmarked child nodes. How many times is each node visited? A node of cardinality c is visited $c - 1$ times, once from each child. This is equal to

the number of straight-lined edges from a superset to a subset node in Figure 4.17. There are $\binom{n}{c} - \binom{n-1}{c} = \binom{n-1}{c-1}$ different nodes that are visited for each cardinality. For example, the number of nodes with cardinality 3 that are visited starting from node 27 is $\binom{5-1}{3-1} = \binom{4}{2} = 6$ (nodes 7 to 12). Thus, the total number of visits v is $v(n) = \sum_{c=1}^{n}(c-1)\binom{n-1}{c-1} = \sum_{c=0}^{n-1} c\binom{n-1}{c}$. Let us give an upper bound for v:

$$\begin{aligned}
v(n) &= \sum_{c=0}^{n-1} \frac{c(n-1)!}{c!(n-1-c)!} \\
&\leq (n-1)! \sum_{c=0}^{n-1} \frac{c}{c!} \\
&= (n-1)! \sum_{c=1}^{n-1} \frac{c}{c!} \\
&= (n-1)! \sum_{c=1}^{n-1} \frac{1}{(c-1)!} \\
&= (n-1)! \sum_{c=0}^{n-2} \frac{1}{c!}.
\end{aligned}$$

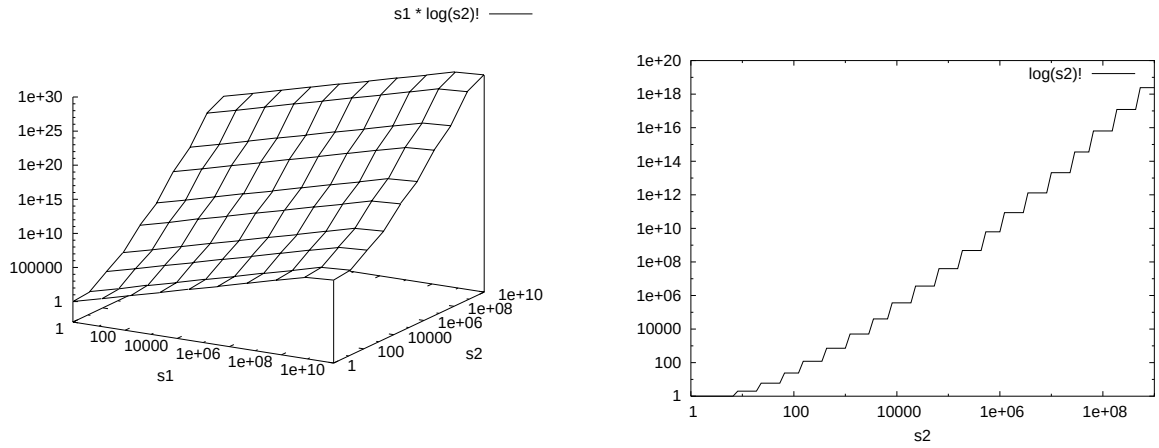
Since $\sum_{c=0}^{\infty} \frac{1}{c!} = e$, one can derive an upper bound for v when $n \to \infty$:

$$v(n) \to e(n-1)!.$$

Let us find a lower bound of $v(n)$:

$$\begin{aligned}
v(n) &= \sum_{c=0}^{n-1} \frac{c(n-1)!}{c!(n-1-c)!} \\
&= (n-1)! \left(\frac{0}{0!(n-1)!} + \frac{1}{1!(n-2)!} + \cdots + \frac{\lfloor \frac{n}{2} \rfloor}{\lfloor \frac{n}{2} \rfloor ! \left(n - \lfloor \frac{n}{2} \rfloor\right)!} + \cdots + \frac{n-1}{(n-1)!0!} \right) \\
&\geq (n-1)! \frac{\lfloor \frac{n}{2} \rfloor}{\lfloor \frac{n}{2} \rfloor ! \left(n - \lfloor \frac{n}{2} \rfloor\right)!} \\
&\approx (n-1)! \frac{\frac{n}{2}}{\frac{n}{2}!^2} \\
&= \frac{\frac{n!}{2}}{\frac{n}{2}!^2}
\end{aligned}$$

In other words, for large values of n, the total number of visits approaches a complexitiy function that lies between $O(\frac{n!}{\frac{n}{2}!^2})$ and $O(n!)$. Assuming the upper bound for O, namely $O(n!)$, the effort to probe the index with a single set of r_1 is proportional to $n! = \log(s_2+1)!$. In summary, the



(a) Upper bound of time complexity

(b) Space complexity

Figure 4.18: Worst case time and space complexities for subset index probing

worst-case computational complexity of probing an uncompressed subset index on r_2 having s_2 sets with a table r_1 having s_1 sets is in $O(s_1 \log(s_2)!)$. The plot of this function is illustrated in Figure 4.18(a).

The space complexity can be computed as follows. The *lowBorder* data structure contains n nodes and the *highBorder* has one node, the largest set. The number of edges in the *cardinalitySiblings* data structure is 2^n. However, the number dominating the total number of edges is the number of subset edges between the nodes: $\sum_{c=1}^{n-1} \binom{n}{c} c$. We have just seen that for large values of n, this expression is proportional to $n!$. Hence, the worst case space complexity is $O(\log(s_2)!)$.

During the build phase, the sets of r_2 are inserted in decreasing order of cardinality, as described in Section 4.5.4. At level c, we insert a node into *cardinalitySiblings*$[c]$. We have seen that when *cardinalitySiblings*$[c]$ is filled, it contains $m = \binom{n}{c}$ sets. Inserting each of the m nodes causes an effort of $\sum_{k=1}^{m} \frac{log(k)}{m}$. It is easy to see that this expression does not dominate the overall complexity. We test the set to be inserted whether it is a subset of any set that has a greater cardinality. The number of visits for a graph for the powerset of a set that a largest cardinality of n is $v(n)$, as we have seen before. In the case of index creation, we make this test repeatedly for growing sizes of n. The total number of visits V is

$$V(n) = \sum_{c=1}^{n} \sum_{k=1}^{m} v(k) = \sum_{c=1}^{n} \sum_{k=1}^{\binom{n}{c}} v(k).$$

Let us find an upper bound for this expression. The value of $\binom{n}{c}$ is maximized for $c = \lfloor \frac{n}{2} \rfloor$. The

value of $\binom{2l}{l} = \frac{(2l)!}{(l!)^2}$ is called the lth *central binomial coefficient*. Here, $l = \lfloor \frac{n}{2} \rfloor$. Hence,

$$V(n) \leq n \sum_{k=1}^{\binom{n}{\lfloor \frac{n}{2} \rfloor}} v(k) \leq n \frac{(2n)!}{\lfloor n \rfloor !^2} v(n)$$

Using the upper bound of $v(n) = e(n-1)!$, we find that for large values of n,

$$V(n) \to n \frac{(2n)!}{\lfloor n \rfloor !^2} e(n-1)! = \frac{en!(2n)!}{\lfloor n \rfloor !^2} \leq (2n)!$$

In summary, the worst case computational complexity of the build phase of the subset index on table r_2 is $O((2 \log s_2)!)$.

When we analyze the compressed version of the subset index, the best case space complexity occurs when all sets overlap such that only one element per set has to be stored. We assume that there are no duplicate sets in r_2. For example, consider the sets $\{\{1\},\{1,2\},\{1,2,3\}\}$. Only the sets $\{\{1\},\{2\},\{3\}\}$ need to be stored. The best case space complexity is thus $O(s_2)$. The worst case space complexity is similar to that for an uncompressed index. The only difference is that we may not require as much memory to store the elements due to overlapping items. However, the number of edges does not decrease.

We leave as future work an exact computation of the worst case time complexity for a compressed subset index. It is obvious that the complexity is worse than that of the uncompressed subset index.

4.6 Disk-Resident Indexes

We can use a index structures to access the elements in the sub- or supersets efficiently. Many different index structures come into consideration. For example, the *universal B-tree (UB-tree)* [Bay96] is a multi-dimensional index that allows to efficiently access the rows in a table in the sort order of any index attribute combination. For example, for the dividend table r_1 with schema $R_1(a_1,\ldots a_m,b_1,\ldots,b_n)$, the UB-tree allows to access the rows ordered by $(a_1),\ldots,(b_n)$, (a_1,a_2), ..., (b_n,b_{n-1}), ..., $(a_1,\ldots,a_m,b_1,\ldots,b_n)$, ..., $(b_n,\ldots,b_1,a_m,\ldots,a_1)$. For example, this includes the combination (b_7,a_3,b_5).

Bayer notes that division can be efficiently processed using the *UB-cache* algorithm [Bay97] when the divisor fits into memory and at least one of the attributes in A is contained in the set of attributes used to index the dividend table. The intention is to use a merge-sort division algorithm (Section 3.3.2.2), where the dividend, which is typically larger than the divisor, is indexed by a UB-tree and the divisor is sorted on-the-fly and held in memory since it is relatively small. The technique used to read the rows of a table indexed by a UB-tree in a sorted order is called the *Tetris algorithm* [MZB99].

4.7 Summary

This section gave an overview of algorithms realizing set containment tests between two collections of sets. In case of set containment join algorithms, the test is a join between (a single) set-valued column in each of the input tables. In case of set containment division, the matching is defined on one or more columns in each input. First, we highlighted several physical storage representations for sets in an RDBMS. For example, the representation required for set containment division is unnested internal. Then, we discussed two important techniques for set containment join algorithms: signatures to improve the cost of each set comparison and partitioning to reduce the number of comparisons. We went on by studying two set containment join algorithms that employ these techniques, namely partitioning set join and adaptive pick-and-sweep join. For the latter algorithm, we were able to prove the optimality of two qualifiers that characterize efficiency: The comparison factor indicates the relative number of signatures compared to the product of the input table cardinalities and the replication factor is the relative number of signatures that are temporarily stored into partitions compared to the number of original sets. We presented a generic execution template of set containment division as well as two instances of it: merge sort set containment division and hash-based set containment division. Then, several strategies for parallelizing set containment division algorithms were investigated, which rely on a horizontal partitioning of the dividend and divisor. We proposed a new approach for set containment division that relies on an in-memory data structure called subset index. This approach does not employ signatures.

Several of the algorithms discussed in this chapter have been implemented in Java and were subject to performance experiments that we will present in in Chapter 6. Before that, we look at an interesting application area for set containment tests in the following chapter.

"We have fast, scalable algorithms for the [data mining] operations and we can parallelize them. The part still missing is a nice algebra for composing the operations. We should also be able to combine data mining operations with traditional database operations."

R. Agrawal [Win03b]

5

A New Approach to Frequent Itemset Discovery with SQL

In this chapter, we analyze and compare SQL-based algorithms to compute frequent itemsets, including a new approach, whose relational algebra representation exploits the set containment division operator.

5.1 Introduction

The discovery of frequent itemsets is a computationally expensive preprocessing step for association rule discovery [AIS93], which finds rules in large transactional datasets. Frequent itemsets are combinations of items that appear frequently together in a given set of transactions. Association rules characterize, e.g., a purchase pattern of retail customers or the click pattern of web site visitors. Such information can be used to improve marketing campaigns, retailer store layouts, or the design of a web site's contents and hyperlink structure.

Most commercial data mining systems and research prototypes employ algorithms that run on data stored in flat files. However, database vendors begin to act against the general lack of mining functionality to support business intelligence and integrate new "primitives" into their

systems [CDH+99]. Companies employing data mining tools for their business [HGG01] realize the need for integrating data mining algorithms with DBMS.

Some authors in research as well as in industry have suggested special data mining languages like *MSQL* [IV99], *DMQL* [HFKZ96] or a variant of DMQL [HIL00], and *ATLaS* [WZ02]. Others have enriched query languages with mining functionality, like the *MINE RULE* operator [MPC96] or *OLE DB for Data Mining* [NCFB01]. However, the query processing power offered by modern database systems for mining purposes has been widely neglected in the past. Some research results show that algorithms for frequent itemset discovery based on SQL are less efficient than those based on sophisticated in-memory data structures [STA98]. Others claim that "even SQL *as is* is adequate even for complex data mining queries and algorithms" [Zan02]. Nevertheless, it becomes ever more important for database system vendors to offer novel analytic functionalities to support business intelligence applications.

In this chapter, we analyze several approaches to compute frequent itemsets using SQL. We also propose a new SQL-based approach and compare it to the other approaches.

The remainder of this chapter is organized as follows. In Section 5.2, we briefly highlight general pros and cons of data mining using an RDBMS. Section 5.3 introduces the problem of frequent itemset discovery. Then, in Section 5.4, we argue why our new algorithm is not just "yet another" SQL-based approach for frequent itemset mining. In Section 5.5, we discuss alternative ways to store and process data in tables of a relational database system. Section 5.6 highlights important known approaches using SQL-92 before we introduce our approach that makes use of a vertical table layout in Section 5.7. Section 5.8 summarizes this chapter.

5.2 Database Mining

In business intelligence applications, several data mining and OLAP techniques are employed to extract novel and useful information from huge corporate datasets. Typically, the datasets are managed by a data warehouse that is based on relational database technology. Although the terms *data mining* and, even more so, *knowledge discovery in databases (KDD)* suggest that the algorithms explore databases, most commercial tools merely process flat files. If they do access a database system, then database tables are used as a container to read and write data, similar to a file system. The query optimization and processing facilities of current database systems are hardly ever exploited by current data mining tools. The reasons for this certainly include:

- Portability: A data mining application that does not rely on a query language can be deployed more easily because no assumptions on the language's functionality have to be made.

- Performance: A highly tuned black-box algorithm with in-memory data structures will always be able to outperform any query processor that employs a combination of generic algorithms.

- Secrecy: A data mining tool vendor does not want to reveal application logic. By employing SQL-based algorithms, the database administrator will be able to see these queries.

Despite these arguments against SQL-based data mining algorithms, exploiting the query language power for expressing data mining (sub)problems can solve several important problems:

- Data currency: The latest updates applied to the data warehouse are reflected in the query result. No replications, i.e. copies of the original dataset, have to be maintained such that they reflect the freshest database content.

- Scalability: If extremely large datasets are to be mined, it is much easier to design a scalable SQL-based algorithm than designing an algorithm that has to manage data in external files. The storage management is one of the key strengths of a database system.

- Adaptability to data: A database optimizer tries to find the best possible execution strategy based on the current data characteristics for a given query. Of course, in some situations this will not help. Similar to choosing a different proprietary algorithm for certain data characteristics, we may get higher performance gains when we employ different data mining queries.

The latter three arguments motivated our research on SQL-based algorithms for frequent itemset discovery.

5.3 The Problem of Frequent Itemset Discovery

We briefly introduce the widely established terminology relevant for frequent itemset discovery originating from [AIS93]. An *item* is an object of analytic interest like a product of a shop or a URL of a document on a web site. An *itemset* is a set of items and a *k-itemset* contains k items. A *transaction* is an itemset representing a fact like a purchase of products or a collection of documents requested by a user during a web site visit.

Given a set of transactions T, the *frequent itemset discovery problem* is to find itemsets within T that appear at least as frequently as a given threshold s_{min}, called *minimum support*. An itemset i is *frequent* if

$$\frac{|\{t \in T \mid i \subseteq t\}|}{|T|} \geq s_{min}.$$

For example, a user can define that an itemset is frequent if it appears in at least 2% of all transactions.

Almost all itemset discovery algorithms consist of a sequence of steps that proceed in a bottom-up manner. The result of the kth step is the set of frequent k-itemsets, denoted as F_k. The first step computes the set of frequent items (1-itemsets). Each following step $k \geq 2$ consists of two phases:

1. The *candidate generation phase* computes a set of potential frequent k-itemsets from F_{k-1}. The new set is called C_k, the set of *candidate* k-itemsets. It is a superset of F_k.

2. The *support counting phase* filters out those itemsets from C_k that appear more frequently in the given set of transactions than the minimum support and adds them to F_k, the set of *frequent k*-itemsets.

All known SQL-based algorithms follow this "classical" two-phase approach. There are other, non-SQL-based approaches, like *frequent-pattern growth*, which do not require a candidate generation phase [HK01]. The frequent-pattern growth algorithm, however, employs a (relatively complex) main-memory data structure, called frequent-pattern tree, which disqualifies it for a straightforward comparison with SQL-based algorithms.

5.4 Motivation for a New SQL-Based Approach

The key problem in frequent itemset discovery is: "How many transactions contain a certain given itemset?" This question can be answered in relational algebra using the division operator that we discussed in Section 2.3.1. Suppose that we have a relation *t*(*t#*, *i#*) containing a set of transactions, where *t#* is a transaction number or identifier and *i#* is an item number, and a relation *i*(*i#*) containing a single itemset, i.e., each tuple contains one item. We want to collect those *t#* values in a result relation *r*(*t#*), where *for all* tuples in *i*, there is a corresponding tuple in *t* that has a matching *i#* value together with that *t#*. In relational algebra, this problem can be stated as $t(t\#, i\#) \div i(i\#) = r(t\#)$.

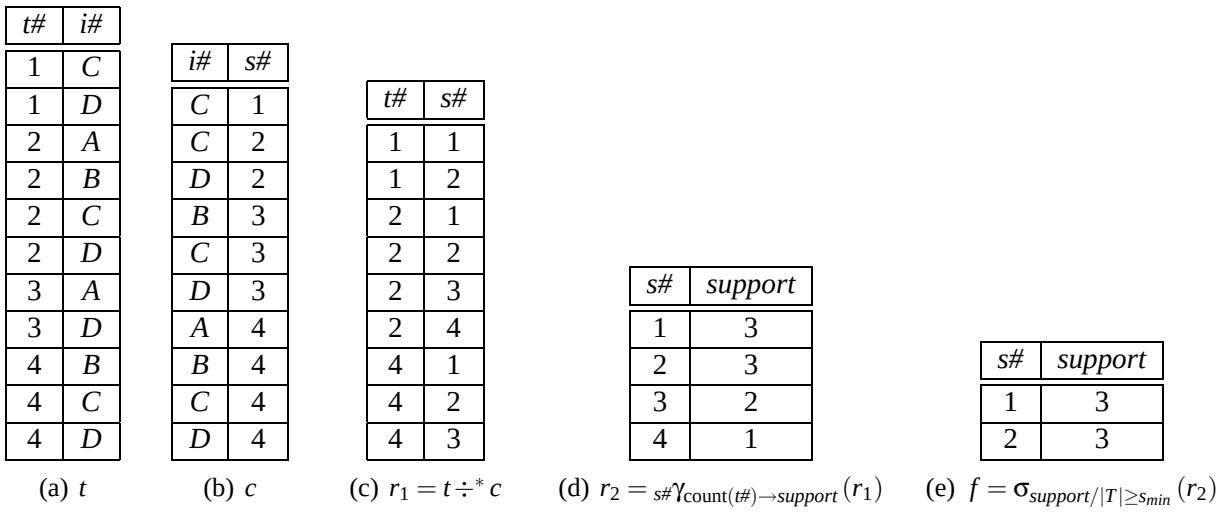
Unfortunately, frequent itemset discovery typically poses the additional problem that we have to check *many* (candidate) itemsets for sufficient frequency, i.e., we do not have a constant divisor relation but we need to divide *t* by several divisors. This is exactly the functionality of the set containment divisor operator that we introduced in Section 2.3.2. Instead of a relation *i* that holds only a single divisor (a single candidate itemset), we use a candidate itemsets relation *c*(*s#*, *i#*), where *s#* is an itemset number or identifier and *i#* is an item belonging to an itemset. Using set containment division, we can formulate the itemset containment test as $t \div^* c = r$. Now, each tuple of the result relation *r*(*t#*, *s#*) characterizes which itemset is contained in which transaction. The frequent itemsets are those values of *s#* that occur in more tuples of *r* than specified by the minimum support. In summary, we can find the relation of frequent itemsets *f*(*i#*,*support*) as follows:

$$f = \sigma_{support/|T| \geq s_{min}} \left({}_{s\#}\gamma_{\text{count}(t\#) \rightarrow support} \left(t \div^* c \right) \right). \tag{5.1}$$

Note that $|T|$ denotes the number of transactions in relation *t*, not the number of tuples. Figure 5.1 illustrates the support counting approach using set containment division with an example.[1] The candidate itemsets relation *c* contains 4 itemsets of different size. We assume a minimum support s_{min} of 75%. Two of the four itemsets in *s* are frequent, *C* and *CD*, because $|T| = 4$, their *support* count is 3, and hence the condition $support/|T| \geq s_{min}$ is fulfilled.

Based on the idea of using set containment division to specify the itemset containment problem, we devised a complete algorithm in SQL using a vertical table layout and universal quan-

[1] We use letters instead of numbers for the *i#* values in the examples throughout the chapter for easier comprehension.



t#	i#
1	C
1	D
2	A
2	B
2	C
2	D
3	A
3	D
4	B
4	C
4	D

(a) t

i#	s#
C	1
C	2
D	2
B	3
C	3
D	3
A	4
B	4
C	4
D	4

(b) c

t#	s#
1	1
1	2
2	1
2	2
2	3
2	4
4	1
4	2
4	3

(c) $r_1 = t \div^* c$

s#	support
1	3
2	3
3	2
4	1

(d) $r_2 = {}_{s\#}\gamma_{\text{count}(t\#)\rightarrow support}(r_1)$

s#	support
1	3
2	3

(e) $f = \sigma_{support/|T| \geq s_{min}}(r_2)$

Figure 5.1: An example showing the relationship between frequent itemset discovery and the set containment division operator

tifications. We use the term "universal quantification" instead of "division" because we will first specify the queries of our approach using tuple relational calculus, which includes the mathematical universal quantifier ($\forall$). In addition, we will present relational algebra expressions—we have just shown the algebra expression for the support counting phase—as well as SQL queries that are equivalent to the calculus expressions.

Although there are two main approaches to express universal quantification in SQL, as mentioned in Section 2.3.1, one based on counting, the other based on value comparisons, we focused on the latter approach. The reason is that the counting approach puts extra restrictions to the quantification problem and it is less intuitive. To see why, suppose we want to test if an itemset i is contained in a transaction t. The counting approach compares the number of items in i with those in t. This comparison makes sense only if we require that t contains only items that are contained in i. In other words, we have to remove those items from t in a preprocessing step that are not contained in i. After that, we count the number of items in i and t and find that $i \subseteq t$ if the numbers are equal. The SQL statements based on value comparisons that will be explained later in this chapter are a more intuitive formulation of the quantification problem.

5.5 Table Layout

Before data can be mined with SQL, it has to be made available as relational tables. Typically, the data is stored in tables within that database system or, if the data has to be kept outside of the database system, it has to be made accessible to the database system by using wrappers that provide a relational view on the data.

Layout

Transactions

horizontal/ single-row

t#	i#$_1$	i#$_2$	i#$_3$
1	B	C	D
2	C	D	NULL

Itemsets

s#	i#$_1$	i#$_2$
1	B	C
2	B	D

vertical/ multi-row

t#	i#
1	B
1	C
1	D
2	C
2	D

s#	pos	i#
1	1	B
1	2	C
2	1	B
2	2	D

Table 5.1: Table layout alternatives for storing the items of transactions and itemsets

5.5.1 Layout Types

Two types of data objects are relevant for frequent itemset discovery: transactions and itemsets. For each type, there are basically two main layouts (schemas) for representing these objects in a table. In particular, the items of an object can be stored either in a single row, which we call *horizontal* layout, or in several rows, which we call *vertical* layout, illustrated in Table 5.1. Note that the vertical layout for itemsets has a position attribute *pos* associated with each item. This is necessary because most algorithms assume a lexicographic order of items within an itemset and they need to access an item at a specific position.

Almost all known SQL-based approaches assume that transactions are stored in a vertical layout. To the best of our knowledge, only the approach proposed by Rajamani et al. [RCIC99] assumes a horizontal layout. In that approach, the horizontal/vertical layout for transactions is called multi-column/single-column data model, respectively. No layout alternatives for itemsets are discussed in that paper because the focus is on *input* data (i.e., only the transactions) for association rule discovery algorithms, not on *intermediate* or *result* data (itemsets) representations, as in this chapter.

Analogous to transactions, there are two different table layouts for itemsets. All known approaches for frequent itemset discovery based on SQL-92 assume a horizontal itemset table layout.

A third, hybrid way of storing items is possible, combining the vertical and horizontal approaches. It may happen that the size of itemsets or, more likely, the size of some transactions is larger than the number of item attributes that have been defined for the tables of a horizontal layout. For example, if 99% of all transactions to be stored in a database have up to ten items but only 1% has ten items or more then a database designer may decide that a horizontal layout with ten item attributes is reasonable. For the few long transactions, however, the remaining items can be stored in additional rows. For example, if the transaction table layout has ten item attributes and we want to store an itemset of size 33, then at least four rows are required to store

the entire transaction. Of course, all rows belonging to the same transaction/itemset must have a common *t#*/*s#* value, as with the vertical approach. We will not further discuss this approach in this chapter.

Further, rather exotic approaches have been proposed to represent transaction data. For example, the decomposed data structure described in [HKMT95] is a non-relational data layout, called *decomposed storage structure*, used in the database system *Monet* where the transactions are stored as follows. Let I denote the number of distinct items in the transactions. The transactions are stored in I columns where each column contains the set of *t#* values that contain the item. Hence, the frequent 1-itemsets are simply the columns that contain a sufficient number of *t#* values. Other layouts are used by SQL-based algorithms with object-relational features like user-defined functions, as defined in SQL:1999. These layouts have a column with a container data type (like BLOB or VARCHAR) to store lists of objects. One example approach, called *Vertical* [STA98], uses a transaction table layout $t(t\#, i\#set)$, where *i#set* contains the list of items of a transaction. A similar approach, called *Horizontal* [STA98], uses the layout $t(i\#, t\#set)$, similar to the decomposed structure described above. For each distinct item, there is a list of all *t#* values that contain the item. However, the Horizontal approach uses a row instead of a column for each item, as in the decomposed layout. In this chapter, we restrict our discussion to approaches based on SQL-92, i.e., we focus on the vertical and horizontal layout.

5.5.2 Vertical vs. Horizontal Layout

In the following, we will use the term *object* to denote itemsets and transactions alike. The vertical approach differs from the horizontal approach in several ways, like the maximum object size and the number of tables and indexes used for the objects.

5.5.2.1 Object Size

The size of an object does not need to be specified in the vertical layout. If we want to store very large objects using a horizontal layout, it could happen that the maximum number of table columns allowed in the database system is lower than the desired object size.[2] Not only the storage of objects may be restricted in a horizontal layout but also the processing of queries may cause problems. The number of attributes allowed in a SELECT clause of SQL may also be lower than required by an SQL-based algorithm. Therefore, we have to take care of the fact that we should also avoid very long attribute lists in projections, i.e., the SQL-based approach should not produce an intermediate result that has a horizontal layout inside the queries even if the outcome of a query is in a vertical layout.

5.5.2.2 Number of Tables

Objects of different size can be stored in the same table if a vertical layout is used. We could even store all objects in a single table. In this case, we only need to make sure that the data types of the

[2] For example, IBM DB2 Universal Database 7.2 allows up to 1 012 columns per table, Microsoft SQL Server 2000 a maximum of 1 024 columns per table, and Oracle 8i a maximum of 1 000 columns per table.

columns *i#* and *t#* are compatible, that their values are unique, and that the position attribute values are set according to the lexicographical ordering of transaction items. In a vertical approach, the support counters of frequent itemsets can be kept in a separate support table $s(s\#, support)$, where the *s#* value corresponds to the itemset number of the respective frequent itemsets table *f*(*s#*, *pos*, *i#*). In this case, it is reasonable to define a foreign key *f.s#* referencing table *s*. In a horizontal layout, the support counter can also be added as an additional attribute to the *f* table: $(s\#, i\#_1, \ldots, i\#_n, support)$. It has been reported that the horizontal layout for transactions seems to allow faster algorithms [RCIC99]. However, the vertical layout is much more common for market basket analysis, the most popular field of application for association rule discovery.

5.5.2.3 Number of Indexes

Fewer indexes come into consideration and are required to improve performance in the vertical layout. An itemset table in a vertical layout has three attributes. Hence, only 15 column combinations for indexes are possible.[3] The larger number of potential indexes for the horizontal layout requires a more thorough analysis on which subset of indexes could actually be exploited by the queries given the current characteristics of data to be mined. On the other hand, one may argue that the most expensive phases in frequent itemset discovery are the first three and hence this discussion is more of theoretical than of practical relevance.

5.6 Overview of SQL-Based Algorithms

There is a multitude of algorithms for frequent itemset discovery. Most approaches do not consider the query functionality of a database system but merely its storage capability. Implementations of these approaches, including commercial mining systems, use a database system like a file system for retrieving input transaction data and in rare cases also for storing intermediate and result itemsets. The focus of most research on new algorithms lies on main-memory data structures that allow an efficient candidate generation and support counting phase. Such algorithms have to provide scalability in addition to the core functionality itself. In contrast, SQL-based approaches can rely on the query execution engine to handle a scalable processing of the queries [RS99]. However, they often lack the efficiency of main-memory-based approaches.

Even the subclass of algorithms that use SQL queries is large. A couple of approaches employ queries containing user-defined functions, which are processed by an object-relational database system. Furthermore, several approaches do not employ any user-defined procedural code at all. Such algorithms use only queries that conform to the SQL-92 standard. In this chapter, we focus on approaches based on SQL-92.

The *SETM* algorithm is the first SQL-based approach [HS95] for frequent itemset discovery described in the literature. Several researchers have suggested improvements of SETM. For example, in [YPK00] the use of views is suggested instead of some of the tables employed, as well

[3] There are $n!\sum_{k=0}^{n-1}\frac{1}{k!}$ combinations for n attributes. This is at least exponential, since $2^n - 1 \leq n!\sum_{k=0}^{n-1}\frac{1}{k!} < en!$. Here, we do *not* take into account the types of indexes, like clustered, secondary, bitmap, etc., but we focus on the attribute combinations only.

s#	i#$_1$	i#$_2$	i#$_3$
1	A	B	C
2	A	B	D
3	A	B	E
4	A	C	D
5	B	C	D

(a) F_3

s#	i#$_1$	i#$_2$	i#$_3$	i#$_4$
1	A	B	C	D

(b) C_4

Figure 5.2: An example candidate itemset generation

as a reformulation using sub-queries. The performance of SETM on a parallel DBMS has been studied further [PSTK99]. The results have shown that SETM does not perform well on large datasets and new approaches have been devised like *K-Way-Join*, *Three-Way-Join*, *Subquery*, and *Two-Group-Bys* [STA98]. These new algorithms differ only in the statements used for support counting. They use the same SQL statement for generating C_k, as shown in Algorithm 3 for the example value $k = 4$. The statement creates a new candidate k-itemset by exploiting the fact that all of its k subsets of size $k-1$ have to be frequent. This condition is called *Apriori property*. It was originally introduced in the *Apriori* algorithm [AS94, MTV94]. Two frequent subsets are picked to construct a new candidate. These itemsets must have the same items from position 1 up to $k-1$. The new candidate is further constructed by adding the kth items of both itemsets in a lexicographically ascending order. In addition, the statement checks if the $k-2$ remaining subsets of the new candidates are frequent as well. We show an SQL statement in Algorithm 4 that is equivalent to the flat query in Algorithm 3. Unlike the flat query, it makes the Apriori checks more visible through the use of sub-queries. For example, suppose we are given the table of frequent 3-itemsets in Figure 5.2(a). Let us further suppose that the current state of query processing binds the tuple variable a_1 to the first row (itemset *ABC*) and a_2 to the second (itemset *ABD*). These variables together represent a potential candidate *ABCD* in the SELECT clause. The WHERE clause of the query in Algorithm 4 checks whether the itemset *ABCD* is a candidate. The first predicate of the WHERE clause ensures that we do not create duplicates. In our example, without this predicate we would generate *ABCD* a second time when a_1 is bound to the second row and a_2 to the first. The first Apriori predicate tests if *BCD* is contained in F_3. We construct *BCD* by skipping the first item in *ABCD*. The second predicate tests if *ACD* is a frequent 3-itemset and the third one searches for itemset *ABD*. There are four subsets of size three for itemset *ABCD*. We do not need to test the existence of the fourth subset *ABC* in the WHERE clause because *ABC* is guaranteed to exist in F_3 since we have picked it in the FROM clause for variable a_1. Since all four subsets of *ABCD* exist in F_3, we add *ABCD* to C_4 (Figure 5.2(b)). In this example, there is no further candidate 4-itemset. Although *ABD* and *ABE* have a common prefix (*AB*) which would yield candidate *ABDE*, the remaining subsets *ADE* and *BDE* are absent in F_3. We will come back to the Apriori property in Section 5.7.1.1 when we discuss the Quiver approach.

The algorithms presented in [STA98] perform differently depending on the data characteris-

```
INSERT
INTO   c4 (s#, i#1, i#2, i#3, i#4)
SELECT newid(), i#1, i#2, i#3, i#4
FROM (
  SELECT a1.i#1, a1.i#2, a1.i#3, a2.i#3
  FROM   f3 AS a1, f3 AS a2, f3 AS a3, f3 AS a4
  WHERE  -- Common prefix:
         a1.i#1 = a2.i#1 AND
         a1.i#2 = a2.i#2 AND
         -- Avoid duplicates:
         a1.i#3 < a2.i#3 AND
         -- Test Apriori property:
         -- Skip first item.
         a3.i#1 = a1.i#2 AND
         a3.i#2 = a1.i#3 AND
         a3.i#3 = a2.i#3 AND
         -- Skip second item.
         a4.i#1 = a1.i#1 AND
         a4.i#2 = a1.i#3 AND
         a4.i#3 = a2.i#3) AS temporary;
```

Algorithm 3: Candidate generation phase for horizontal approaches as a flat SQL query

```
INSERT
INTO   c4 (s#, i#1, i#2, i#3, i#4)
SELECT newid(), i#1, i#2, i#3, i#4
FROM (
  SELECT a1.i#1, a1.i#2, a1.i#3, a2.i#3
  FROM   f3 AS a1, f3 AS a2
  WHERE  -- Avoid duplicates:
         a1.i#3 < a2.i#3 AND
         -- Test Apriori property:
         -- Skip first item.
         (a1.i#2, a1.i#3, a2.i#3) IN (
           SELECT i#1, i#2, i#3 FROM f3) AND
         -- Skip second item.
         (a1.i#1, a1.i#3, a2.i#3) IN (
           SELECT i#1, i#2, i#3 FROM f3) AND
         -- Skip third item.
         (a1.i#1, a1.i#2, a2.i#3) IN (
           SELECT i#1, i#2, i#3 FROM f3)
```

Algorithm 4: Horizontal K-Way-Join candidate generation phase with sub-queries

tics. The Subquery algorithm is reported to be the best algorithm overall compared to the other approaches based on SQL-92. The reason is that it exploits common prefixes between candidate k-itemsets when counting the support. We illustrate the generation of frequent k-itemsets in Algorithm 5 for the example value $k = 4$ instead of giving a generic definition of q_k as in [STA98] or [Mis02].

Another approach presented in [STA98], called *K-Way-Join*, uses k instances of the transaction table and joins it k times with itself and with a single instance of C_k. Algorithm 10 on page 107 illustrates an example SQL statement of this approach. In contrast, Algorithm 11 on page 107 shows an equivalent approach using a vertical layout. We will further discuss this approach in Section 5.7.2.

More recently, an approach called *Set-oriented Apriori* has been proposed [TC99]. The au-

```
INSERT
INTO     f4 (i#1, i#2, i#3, i#4, support)
SELECT   i#1, i#2, i#3, i#4, COUNT(t#) AS support
FROM     (
  SELECT i#1, i#2, i#3, i#4, t#
  FROM   t, (SELECT DISTINCT i#1, i#2, i#3, i#4 FROM c4) AS c, (
    SELECT i#1, i#2, i#3, t#
    FROM   t, (SELECT DISTINCT i#1, i#2, i#3 FROM c4) AS c, (
      SELECT i#1, i#2, t#
      FROM   t, (SELECT DISTINCT i#1, i#2 FROM c4) AS c, (
        SELECT i#1, t#
        FROM   t, (SELECT DISTINCT i#1 FROM c4) AS c
        WHERE  t.i# = c.i#1
             ) AS q1
      WHERE  q1.i#1 = c.i#1 AND
             t.i#   = c.i#2 AND
             t.t#   = q1.t#
           ) AS q2
    WHERE  q2.i#1 = c.i#1 AND
           q2.i#2 = c.i#2 AND
           t.i#   = c.i#3 AND
           t.t#   = q2.t#
         ) AS q3
  WHERE  q3.i#1 = c.i#1 AND
         q3.i#2 = c.i#2 AND
         q3.i#3 = c.i#3 AND
         t.i#   = c.i#4 AND
         t.t#   = q3.t#
         ) AS q4
GROUP BY i#1, i#2, i#3, i#4
HAVING   COUNT(t#) >= @minimum_support;
```

Algorithm 5: Subquery support counting phase

thors argue that too much redundant computation is involved in each support counting phase. They claim that it is beneficial to save the information about which item combinations are contained in which transaction, i.e., in kth iteration, Set-oriented Apriori generates an additional table $t_k(t\#, i\#_1, \ldots, i\#_k)$. They also write that the size of the transaction table t is a major factor in the cost of joins with t. Therefore, they suggest to use a modified transaction table t_f that is a subset of the transaction table containing only frequent items (frequent 1-itemsets), i.e., $t_f = t \ltimes f_1$. The algorithm derives the frequent itemsets by grouping on the k items of t_k and it generates t_{k+1} using t_k. Algorithm 6 shows the SQL statements used to derive the frequent itemsets of size $k = 4$. Their performance results have shown that Set-oriented Apriori performs better than Subquery, especially for high values of k.

A recent study by Mishra [Mis02] compared the performance of some of the above described algorithms for two commercial RDBMS, namely IBM DB2 7.2 and Oracle 8i. The K-Way-Join algorithm was attributed the best algorithm of those based on SQL-92 that they compared (K-Way-Join, Two-Group-By, Subquery). Algorithms based on object-relational features have been studied, too, and were published separately [MC03].

In this section, we have presented the SQL statements only for the algorithms K-Way-Join, Subquery, and Set-oriented Apriori because they are subject to performance tests discussed in Chapter 6. In particular, we chose Subquery and Set-oriented Apriori because they are considered

```
INSERT
INTO   t4 (t#, i#1, i#2, i#3, i#4)
SELECT t3.t#, t3.i#1, t3.i#2, t3.i#3, tf.i#
FROM   c4, t3, tf
WHERE  t3.i#1 = c4.i#1 AND
       t3.i#2 = c4.i#2 AND
       t3.i#3 = c4.i#3 AND
       tf.i#  = c4.i#4 AND
       tf.t#  = t3.t#;

INSERT
INTO     f4 (i#1, i#2, i#3, i#4, support)
SELECT   i#1, i#2, i#3, i#4, COUNT(t#) AS support
FROM     t4
GROUP BY i#1, i#2, i#3, i#4
HAVING   COUNT(t#) >= @minimum_support;
```

Algorithm 6: Set-oriented Apriori support counting phase

the "fastest" algorithms based on SQL-92 according to the literature. In addition, we chose K-Way-Join because of its structural similarity to our new approach, discussed next.

5.7 Quiver

In this section, we suggest a new approach called *Quiver (quantified itemset discovery using a vertical table layout)* for computing frequent itemsets using SQL [Ran02]. It requires a vertical table layout for computing candidate and frequent itemsets, as defined in Section 5.5. In addition, it employs universal and existential quantifications of tuple variables. In the following, we will discuss the two phases of frequent itemset discovery according to Quiver, candidate generation and support counting.

5.7.1 Candidate Generation Phase

Before we show how to accomplish the entire candidate generation in SQL, we explain the key ideas of the Quiver approach using the tuple relational calculus notation used in [RG00]. We do this because the calculus is more concise than SQL and the universal quantification used in Quiver becomes apparent.

5.7.1.1 Tuple Relational Calculus

The generation of candidate k-itemsets can be expressed in a single calculus expression. However, we have decomposed it into several sub-expressions for a clearer presentation.

In the following, we assume that we have computed F_{k-1}, the set of $(k-1)$-itemsets for some $k \geq 2$ during the previous support counting phase. The sub-expressions use the tuple variables a_1, a_2, a_3, b_1, b_2, and c referring to the same relation F_{k-1}. All candidate k-itemsets have to fulfill the calculus query C_k shown in Algorithm 7. Note that the expressions are actually *templates* of expressions because they are parameterized. For example, the template $a(k, p)$, explained below,

$$
\begin{aligned}
C_k &= \{c \mid candidate(c,k)\} \\
candidate(c,k) &= \exists a_1 \in F_{k-1}\ \ \exists a_2 \in F_{k-1}\,(\\
&\quad [c.s\# = unique()] \wedge \\
&\quad [((c.pos = a_1.pos) \wedge (1 \leq a_1.pos \leq k-1) \wedge (c.i\# = a_1.i\#)) \vee \\
&\quad ((c.pos = k) \wedge (a_2.pos = k-1) \wedge (c.i\# = a_2.i\#))] \wedge \\
&\quad prefix\text{-}pair\,(a_1,a_2,k)) \\
prefix\text{-}pair\,(a_1,a_2,k) &= \forall b_1 \in F_{k-1}\ \ \forall b_2 \in F_{k-1}\,(\\
&\quad [(b_1.s\# = a_1.s\#) \wedge (b_2.s\# = a_2.s\#) \wedge \\
&\quad (b_1.pos < k-1) \wedge (b_1.pos = b_2.pos)] \rightarrow (b_1.i\# = b_2.i\#)) \wedge \\
&\quad \exists b_1 \in F_{k-1}\ \ \exists b_2 \in F_{k-1}\,(\\
&\quad [(b_1.s\# = a_1.s\#) \wedge (b_2.s\# = a_2.s\#) \wedge \\
&\quad (b_1.pos = k-1) \wedge (b_1.pos = b_2.pos)] \rightarrow (b_1.i\# < b_2.i\#)) \wedge \\
&\quad apriori\,(a_1,a_2,k) \\
apriori\,(a_1,a_2,k) &= \bigwedge_{p=1}^{k-2} a\,(a_1,a_2,p,k) \\
a\,(a_1,a_2,p,k) &= \exists a_3 \in F_{k-1}\ \ \forall b_1 \in F_{k-1}\ \ \forall b_2 \in F_{k-1}\,(\\
&\quad [((b_1.s\# = a_1.s\#) \wedge (b_2.s\# = a_3.s\#) \wedge \\
&\quad (b_2.pos < p) \wedge (b_1.pos = b_2.pos)) \rightarrow (b_1.i\# = b_2.i\#)] \wedge \\
&\quad [((b_1.s\# = a_1.s\#) \wedge (b_2.s\# = a_3.s\#) \wedge \\
&\quad (p = b_2.pos < k-1) \wedge (b_1.pos = b_2.pos + 1)) \rightarrow \\
&\quad (b_1.i\# = b_2.i\#)] \wedge \\
&\quad [((b_1.s\# = a_2.s\#) \wedge (b_2.s\# = a_3.s\#) \wedge \\
&\quad (b_2.pos = k-1) \wedge (b_1.pos = b_2.pos)) \rightarrow (b_1.i\# = b_2.i\#)])
\end{aligned}
$$

Algorithm 7: Quiver candidate generation phase in tuple relational calculus (see Figure 5.4 for an example)

has as input parameters k, the size of the candidate itemsets to be created, as well as p, the item position within an itemset. However, in the rest of the chapter we will use term "expression" instead of "expression templates" for simplicity.

The *candidate* expression relies on two sub-expressions, *unique* and *prefix-pair*. The *unique* expression can be regarded as a function that creates a new *s#* value that must be distinct from all existing values and that is guaranteed to be different from any value that is returned for a different input value pair. When we use SQL for generating such a unique identifier, we could simply use the current timestamp.

The second sub-expression of *candidate*, called *prefix-pair*, finds *s#* value pairs (a_1, a_2) of frequent $(k-1)$-itemsets that have a common prefix of size $k-2$. Such an itemset pair has the same *i#* value at each position from 1 to $k-2$, and the *i#* value of the first itemset at position $k-1$ is lexicographically ordered before that of the second itemset. For example, we will create a new itemset *ABCD* for C_4 if we find the itemsets *ABC* and *ABD* in F_3, which have the common prefix *AB*.

The *prefix-pair* calculus expression contains universal quantifications and logical implications. An implication of the form $f \rightarrow g$ expresses the fact that if f holds then g must hold, too. For example, we can phrase the for-all expression in *prefix-pair* as follows: "For all item combinations $(b_1.i\#, b_2.i\#)$ of itemsets a_1 and a_2, if we look at the same position of any but the last item of both itemsets, then they must have the same *i#* value at this position." The existential expression that follows the universal expression can be phrased as: "At position $k-1$, the item value *i#* of the first itemset that we aim to find has to be lexicographically less than that of the second itemset."

The Quiver approach tries to reduce the number of candidates by ignoring all itemsets that do not fulfill the Apriori property, like in the K-Way-Join algorithm, described in Section 5.6. Thus, *prefix-pair* contains another expression called $apriori(k)$, which has to hold as well. This expression checks if, apart from a_1 and a_2, which we know are frequent, all other subsets of size $k-1$ are frequent, too. For example, for the potential candidate *ABCD*, $apriori(4)$ checks if the subsets *ACD* and *BCD* are contained in F_3. We already know that *ABC* and *ABD* have to be frequent because we used them for the construction of the potential candidate. Hence, we can ignore these checks in $apriori(4)$. In general, $apriori(k)$ tests the existence in F_{k-1} of those $(k-1)$-itemsets that we get when skipping a single item at the positions 1 to $k-2$ from a potential candidate k-itemset. For candidate itemset *ABCD*, e.g., we skip position 1 to get *BCD* and position 2 to get *ACD*. This Apriori check is represented by a conjunction of k similar expressions $a(k,p)$, each having a different value of the position parameter p, where $1 \leq p \leq k-2$, i.e., $apriori(k) = a(k,1) \wedge a(k,2) \wedge \ldots \wedge a(k,k-2)$. Each expression $a(k,p)$ checks if such a $(k-1)$-itemset is frequent that we get when we skip the item at position p of the potential candidate k-itemset.

5.7.1.2 Relational Algebra

In the following, we describe step-by-step how to compute C_{k+1} using relational algebra. First, we find pairs of itemset values $(s\#_1, s\#_2)$ that have a common prefix of size $k-1$. This can be achieved by a set containment division between F_k and F_k, where the itemset values represent the group attributes and we match the values of $(i\#, pos)$ with each other, resulting in the intermediate relation r_1. Second, we use a new unique itemset value for potential candidate itemsets that will be used instead of the itemset values of the frequent itemset prefix pair of r_1. We achieve this by assuming a function *unique()* that assigns a unique value for every tuple in r_1 to the new attribute name *s#*, resulting in r_2. The extended version of the projection operator, as described in [GMUW02], allows such an assignment. Third, we construct a vertical representation of potential $(k+1)$-candidates by merging the first k tuples of the first prefix partner with the last tuple of the second, making sure that the position value (k) of the latter is incremented by one.

$$
\begin{aligned}
r_1(s\#_1, s\#_2) &= \sigma_{s\#_1 < s\#_2}\left(\pi_{s\# \to s\#_1, pos, i\#}(F_k) \div^* \sigma_{pos \neq k}\left(\pi_{s\# \to s\#_2, pos, i\#}(F_k)\right)\right) \\
r_2(s\#, s\#_1, s\#_2) &= \pi_{\text{unique()} \to s\#, s\#_1, s\#_2}(r_1) \\
r_3(s\#, i\#, pos) &= \pi_{s\#, i\#, pos}\left(\pi_{s\# \to s\#_1, pos, i\#}(F_k) \bowtie r_2\right) \cup \\
&\quad \pi_{s\#, i\#, pos \to pos+1}\left(\sigma_{pos=k}\left(\pi_{s\# \to s\#_2, pos, i\#}(F_k) \bowtie r_2\right)\right) \\
r_4(s\#, i\#, pos) &= \bigcup_{p=1}^{k}\left(\pi_{s\#, i\#}\left(\sigma_{pos \neq p}(r_3)\right)\right) \times \rho_{temporary(pos)}((p)) \\
r_5(s\#_1, s\#_2, pos) &= \pi_{s\# \to s\#_1, i\#}(F_k) \div^* \pi_{s\# \to s\#_2, pos, i\#}(r_4) \\
r_6(s\#) &= \pi_{s\#}(r_3) - \pi_{s\#}\left(\pi_{s\#, pos}(r_3) \,\overline{\ltimes}\, \pi_{s\#_2 \to s\#, pos}(r_5)\right) \\
C_{k+1}(s\#, i\#, pos) &= r_3 \ltimes r_6
\end{aligned}
$$

Algorithm 8: Quiver candidate generation phase in relational algebra (see Figure 5.3 for an example)

Fourth, we generate for each potential $(k+1)$-candidate all subsets of size k in r_4. Fifth, we collect combinations of itemset values of F_k and of those potential candidates that are equivalent (contain the same items) in r_5. Sixth, we keep only those *s#* values of potential candidates in r_3 where *all* subsets of size k are contained in F_k. We achieve this by finding all position values *pos* of potential candidates in r_3 that have no join partner in r_5 using anti-semi-join. Those *s#* values that have a join partner for all k positions are the real candidates. We obtain them by removing the itemset values that have a join partner. Finally, we extract the tuples (items) of the remaining, true candidates.

The relational algebra expressions shown in Algorithm 8 realize the ideas just described. Refer to Table 2.3 on page 7 for details on the operators used. To illustrate these complex definitions, we give an example in Figure 5.3 for $k = 3$ and a given set of frequent itemsets $F_k = \{ABC, ABD, ABE, ACD, BCD\}$. Only one candidate can be constructed from this set: $C_{k+1} = \{ABCD\}$.

5.7.1.3 SQL

We can easily derive SQL statements from the tuple relational calculus expressions specified before. For example, an implication can be replaced by a disjunction, i.e., we transform $f \to g \equiv \neg f \vee g$ into "NOT f OR g." Unfortunately, there is no universal quantifier available in SQL. Therefore, we translate $(\forall x \in T : f(x)) \equiv (\neg \exists x \in T : \neg f(x))$ into "NOT EXISTS (SELECT * FROM T AS x WHERE NOT $f(x)$)." In addition, we can use De Morgan's rule for pushing a negation into a conjunction or a disjunction, for example $\neg(f \wedge g) = \neg f \vee \neg g$.

Algorithm 9 shows the SQL-92 statements of the candidate generation phase in Quiver that are equivalent to the tuple relational calculus given above. While the calculus expressions to compute C_k are generic, the SQL statements are shown for the example value $k = 3$.

s#	*i#*	*pos*
1	*A*	1
1	*B*	2
1	*C*	3
2	*A*	1
2	*B*	2
2	*D*	3
3	*A*	1
3	*B*	2
3	*E*	3
4	*A*	1
4	*C*	2
4	*D*	3
5	*B*	1
5	*C*	2
5	*D*	3

(a) F_3

$s\#_1$	$s\#_2$
1	2
1	3
2	3

(b) r_1

s#	$s\#_1$	$s\#_2$
6	1	2
7	1	3
8	2	3

(c) r_2

s#	*i#*	*pos*
6	*A*	1
6	*B*	2
6	*C*	3
6	*D*	4
7	*A*	1
7	*B*	2
7	*C*	3
7	*E*	4
8	*A*	1
8	*B*	2
8	*D*	3
8	*E*	4

(d) r_3

s#	*i#*	*pos*
6	*B*	1
6	*C*	1
6	*D*	1
7	*B*	1
7	*C*	1
7	*E*	1
8	*B*	1
8	*D*	1
8	*E*	1
6	*A*	2
6	*C*	2
6	*D*	2
7	*A*	2
7	*C*	2
7	*E*	2
8	*A*	2
8	*D*	2
8	*E*	2
6	*A*	3
6	*B*	3
6	*D*	3
7	*A*	3
7	*B*	3
7	*E*	3
8	*A*	3
8	*B*	3
8	*E*	3
6	*A*	4
6	*B*	4
6	*C*	4
7	*A*	4
7	*B*	4
7	*C*	4
8	*A*	4
8	*B*	4
8	*D*	4

(e) r_4

$s\#_1$	$s\#_2$	*pos*
1	6	4
2	6	3
4	6	2
5	6	1
1	7	4
3	7	3
2	8	4
3	8	3

(f) r_5

s#
6

(g) r_6

s#	*i#*	*pos*
6	*A*	1
6	*B*	2
6	*C*	3
6	*D*	4

(h) C_4

Figure 5.3: An example computation of candidate 4-itemsets in Quiver using relational algebra

The first INSERT statement populates the prefix-pair table $p(s\#, s\#_1, s\#_2)$, where *s#* is a newly created unique identifier and the other attributes belong to the value pair of frequent $(k-1)$-

```
SET @k = 3;
INSERT
INTO   P (s#new, s#old1, s#old2)
SELECT newid(), s#1, s#2
FROM   (
SELECT DISTINCT a1.s# AS s#1, a2.s# AS s#2
FROM   F2 AS a1, F2 AS a2
WHERE  NOT EXISTS (
  SELECT * FROM F2 AS b1, F2 AS b2
  WHERE  ((b1.s# = a1.s#) AND (b2.s# = a2.s#) AND
          (b1.pos < @k-1) AND (b1.pos = b2.pos)) AND NOT (b2.i# = b1.i#)
       ) AND
       EXISTS (
  SELECT b1.i#, b2.i#
  FROM   F2 AS b1, F2 AS b2
  WHERE  (b1.s# = a1.s#) AND (b2.s# = a2.s#) AND
         (b1.pos = @k-1) AND (b1.pos = b2.pos) AND (b1.i# < b2.i#)
       )
       AND
       -- In the following, we skip the item at position p of
       -- itemset a1.
       -- The following EXIST clause has to be added for each
       -- value of p, where 1 <= p <= k-2.

       -- Skip item at position p = 1.
       EXISTS (
  SELECT a3.s#
  FROM   F3 AS a3
  WHERE  NOT EXISTS (
    SELECT b1.i#, b2.i#
    FROM   F3 AS b1, F3 AS b2
    WHERE  NOT (
             -- Condition 1: 1 <= i < p
             ( NOT (
                 (b1.s# = a1.s#) AND (b2.s# = a3.s#) AND
                 (1 <= b2.pos) AND (b2.pos < 1) AND (b1.pos = b2.pos)
               ) OR
               (b1.i# = b2.i#)
             ) AND
             -- Condition 2: p <= i < k-1
             ( NOT (
                (b1.s# = a1.s#) AND (b2.s# = a3.s#) AND
                (1 <= b2.pos)   AND
                (b2.pos < @k-1) AND (b1.pos = b2.pos + 1)
               ) OR
               (b1.i# = b2.i#)
             ) AND
             -- Condition 3: i = k-1
             ( NOT (
                 (b1.s# = a2.s#) AND (b2.s# = a3.s#) AND
                 (b2.pos = @k-1) AND (b1.pos = b2.pos)
               ) OR
               (b1.i# = b2.i#)
             )
           )
         )
    )
) AS temporary;

INSERT
INTO   C3 (s#, pos, i#)
SELECT p.s#new, f.pos, f.i#
FROM   F2 AS f, P AS p
WHERE  f.s# = p.s#old1
UNION
SELECT p.s#new, @k, f.i#
FROM   F2 AS f, P AS p
WHERE  f.s#  = p.s#old2 AND
       f.pos = @k-1;
```

Algorithm 9: Quiver candidate generation phase in SQL (see Figure 5.4 for an example)

itemsets with a common $(k-2)$-prefix. In this query, we have merged the identifier generation with the computation of prefix pairs, i.e., the calculus expression *Unique* is translated into the *newid()* function returning a unique identifier (included in Microsoft SQL Server 2000, for ex-

s#	*i#*	*pos*
1	*A*	1
1	*B*	2
1	*C*	3
2	*A*	1
2	*B*	2
2	*D*	3
3	*A*	1
3	*B*	2
3	*E*	3
4	*A*	1
4	*C*	2
4	*D*	3
5	*B*	1
5	*C*	2
5	*D*	3

(a) F_3

s#	$s\#_1$	$s\#_2$
6	1	2

(b) p

s#	*i#*	*pos*
6	*A*	1
6	*B*	2
6	*C*	3
6	*D*	4

(c) C_4

Figure 5.4: An example computation of candidate 4-itemsets in Quiver using tuple relational calculus and SQL

ample).

The second statement derives the candidate k-itemsets. For each row in the table p, we copy each *i#* value and its corresponding *pos* value belonging to $s\#_1$ into the target table $c_k(s\#, pos, i\#)$, together with the newly created *s#* value. In addition, we add another row $(s\#, k, i\#)$ to c_k, where the *i#* value of $s\#_2$ is taken from position $k-1$. This procedure is similar to the algebraic approach. However, the SQL approach employs the Apriori test *before* the prefix-pair table generation, which has the same schema as relation r_2 in Figure 5.3(c), while the algebraic approach tests the Apriori property *after* building r_2. We can see the difference between p and r_2 in Figures 5.3(c) and 5.4(b): The relation r_2 in Figures 5.3(c) contains three tuples, where two of them will turn out to specify no candidates. In contrast, the prefix-pair table p in Figure 5.4(b) consists of only a single row, referencing the two itemsets used to construct the *real* (not merely potential) candidate itemset *ABCD*.

5.7.2 Support Counting Phase

We have seen how universal quantification is used to generate C_k in the Quiver approach. These candidates now have to be checked if they appear frequently enough in the transactions to qualify for F_k. We propose a new approach for support counting that uses universal quantification as well.

Before we discuss the new approach, we show a vertical version for support counting that is equivalent to the original, horizontal approach K-Way-Join, described in Section 5.6. Algorithm 10 illustrates the horizontal K-Way-Join, which joins the candidate k-itemsets in table c_k

```
INSERT
INTO     s3 (s#, support)
SELECT   c.s#, COUNT(*)
FROM     c3 AS c, t AS t1, t AS t2, t AS t3
WHERE    c.i#1 = t1.i# AND
         c.i#2 = t2.i# AND
         c.i#3 = t3.i# AND
         t1.t# = t2.t# AND
         t1.t# = t3.t#
GROUP BY c.s#
HAVING   COUNT(*) >= @minimum_support;

INSERT
INTO   f3 (s#, i#1, i#2, i#3)
SELECT c.s#, c.i#1, c.i#2, c.i#3
FROM   c3 AS c, s3 AS s
WHERE  c.s# = s.s#;
```

Algorithm 10: Horizontal K-Way-Join support counting phase

```
INSERT
INTO     s3 (s#, support)
SELECT   a1.s#, COUNT(*)
FROM     c3 AS c1, c3 AS c2, c3 AS c3, t  AS t1, t  AS t2, t  AS t3
WHERE    c1.s#  = c2.s# AND
         c1.s#  = c3.s# AND
         t1.t#  = t2.t# AND
         t1.t#  = t3.t# AND
         c1.i#  = t1.i# AND
         c2.i#  = t2.i# AND
         c3.i#  = t3.i# AND
         c1.pos = 1     AND
         c2.pos = 2     AND
         c3.pos = 3
GROUP BY c1.s#
HAVING   COUNT(*) >= @minimum_support;

INSERT
INTO   f3 (s#, pos, i#)
SELECT c.s#, c.pos, c.i#
FROM   c3 AS c, s3 AS s
WHERE  c.s# = s.s#;
```

Algorithm 11: Vertical K-Way-Join support counting phase

with k tuple variables of table t, where each join condition matches a different column $i\#_j$ of the same itemset with the item value of a different row belonging to the same transaction t_j. In the vertical version of K-Way-Join, shown in Algorithm 11, both transactions and candidate itemsets have a vertical table layout. Therefore, we replace the joins with different *columns* $i\#_j$ of the same itemset in the horizontal approach by joins with the item value of different *rows* (tuple variables) of the same itemset. Both algorithms are shown for the example value $k = 3$.

Algorithms 10 and 11 show how to compute the intermediate result table s_k containing only the support count information for each frequent itemset. The final result table f_k is derived by joining c_k with s_k. If we were not interested in storing the support counters into a separate table, then the statement for deriving s_k could be merged with the second query, which computes f_k,

$$query = \{(c_1, t_1) \mid c_1 \in C \wedge t_1 \in T \wedge contains\}$$
$$contains = \forall c_2 \in C \;\; \exists t_2 \in T \, (c_2.s\# = c_1.s\#) \rightarrow ((t_2.t\# = t_1.t\#) \wedge (t_2.i\# = c_2.i\#))$$

Algorithm 12: Quiver support counting phase in tuple relational calculus

by replacing the reference to base table s_k in the FROM clause with the query defining s_k.

We compared the horizontal and vertical approach of K-Way-Join because Quiver's support counting phase is similar to the vertical version of K-Way-Join: We replace the explicit check of an item value at each of the k positions (*pos* values 1, 2, and 3 in Algorithm 11) with a universal quantification that checks *all* positions of a candidate k-itemset, independent of the parameter k.

5.7.2.1 Tuple Relational Calculus

The Quiver approach for support counting is defined in tuple relational calculus as sketched in Algorithm 12. The expression *contains* derives combinations of transactions and candidates. It has two free tuple variables c_1 and t_1, where c_1 represents a candidate itemset and t_1 is a transaction that contains the itemset. The quantified (bound) tuple variables c_2 and t_2 represent the items corresponding to c_1 and t_1, respectively. The universal quantification is reflected by the condition that *for each* item $c_2.i\#$ belonging to itemset $c_1.s\#$, there must be an item $t_2.i\#$ belonging to transaction $t_1.t\#$ that matches with c_2.

A combination of values (c_1, t_1) in the query result indicates that the itemset $c_1.s\#$ is contained in the transaction $t_1.t\#$. We can find the support of each candidate by counting the number of distinct values $t_1.t\#$ that appear in a combination $c_1.s\#$. We do not show the actual counting because the basic tuple relational calculus does not include aggregate functions.

5.7.2.2 Relational Algebra

During the motivation of our approach in Section 5.4, we have already shown the relational algebra expression for the support counting phase in Equation 5.1 on page 92.

5.7.2.3 SQL

The calculus query can be translated into SQL in the same manner as explained in Section 5.7.1.3 for the candidate generation phase. The resulting SQL statements are shown in Algorithm 13. It is important to note that the aggregation is applied to the set of *distinct t#* values because duplicates can occur as a result of the query processing.

In addition to the direct translation of the calculus expressions, we show how to formulate the same idea using set containment division in Algorithm 14, assuming the hypothetical SQL syntax for this operator, introduced in Section 2.5.2. Since the ordering of items within an itemset is irrelevant for the set containment division operator, we have to eliminate the *pos* column of candidate itemsets table c when we use it as a divisor input of the GREAT DIVIDE. No DISTINCT

```
INSERT
INTO     s (s#, support)
SELECT   s#, COUNT(DISTINCT t#) AS support
FROM     (
  SELECT c1.s#, t1.t#
  FROM   c AS c1, t AS t1
  WHERE  NOT EXISTS (
    SELECT *
    FROM   c AS c2
    WHERE  NOT EXISTS (
      SELECT *
      FROM   t AS t2
      WHERE  NOT (c1.s# = c2.s#) OR
             (t2.t# = t1.t# AND
              t2.i# = c2.i#)))
         ) AS contains
GROUP BY s#
HAVING   COUNT(DISTINCT t#) >= @minimum_support;

INSERT
INTO   f (s#, pos, i#)
SELECT c.s#, c.pos, c.i#
FROM   c, s
WHERE  c.s# = s.s#;
```

Algorithm 13: Quiver support counting phase in SQL using NOT EXISTS

```
INSERT
INTO     s (s#, support)
SELECT   s#, COUNT(t#) AS support
FROM     t
         GREAT DIVIDE BY (
           SELECT i#, s#
           FROM c
         ) AS c1
         ON (t.i# = c1.i#)
GROUP BY s#
HAVING   COUNT(t#) >= @minimum_support;

INSERT
INTO   f (s#, pos, i#)
SELECT c.s#, c.pos, c.i#
FROM   c, s
WHERE  c.s# = s.s#;
```

Algorithm 14: Quiver support counting phase in SQL using GREAT DIVIDE

keyword is needed for the aggregation because we require that any physical set containment division operator produces a duplicate-free quotient table.

In both Algorithm 13 and 14, the parameter k for the candidate table C_k is omitted, i.e., the two statements are the same for every iteration of the Quiver algorithm because in Quiver we use a vertical layout for itemset tables, whose schema is independent of the value k.

5.8 Summary

In this chapter, we have investigated the problem of frequent itemset discovery and compared several solutions based on SQL-92. The discovery of frequent itemsets is generally composed of two phases: candidate generation and support counting. Support counting can be regarded as a typical set containment test problem, and, in particular, as a set containment division problem.

There are two main approaches to represent transactions and itemsets in SQL-92: a horizontal table layout, where all items of an object are stored in a single row, and a vertical table layout, where an object spans as many rows as its number of items. We have presented a new approach called *Quiver* that employs a vertical table layout for *both* transactions and itemsets by specifying its idea using tuple relational calculus, relational algebra, as well as SQL. All known algorithms based on SQL-92 use a horizontal table layout for itemsets. Because of the vertical table layout, the queries for both phases of frequent itemset discovery can employ universal quantification, as shown in our Quiver approach. The reason why we investigated such an approach using for-all quantifiers is because it allows a natural formulation of the frequent itemset discovery problem—counting the number of transactions that contain *all* elements of a given itemset.

We believe that approaches similar to Quiver that are based on a natural representation of the mining problem in SQL will narrow the gap between data mining algorithms and database systems.

The performance of the SQL-based frequent itemset discovery algorithms, in particular Quiver, as well as the performance of set containment division algorithms in general will be studied in the following chapter.

"I don't count instructions any more. I do count I/Os, but maybe the day is coming when I stop counting I/Os."

J. Gray [Win03a]

6 Performance Evaluation

In this chapter, we report on performance experiments in the two main directions of our investigations. First, we examine set containment division and set containment join algorithms. These experiments have been conducted with a prototype of a query execution engine implemented in Java. Second, we compare the performance of SQL-based algorithms for the frequent itemset discovery problem using two commercial RDBMSs. We also show how to improve the performance of such queries when a set containment division operator is employed in the query execution strategy. The experimental results are supplemented by a description of implementation details.

6.1 Implementation of a Java Query Execution Engine

In this section, we give an overview of the implementation of our query execution engine prototype.

6.1.1 Overview of Physical Operators

We have implemented several algorithms for the operators

- division, namely
  - merge-sort division (Section 3.3.2.2) and
  - hash-division (Section 3.3.2.4),

- set containment division, namely
  - merge-sort set containment division (Section 4.3.2),
  - hash-based set containment division (Section 4.3.3), and
  - subset index set containment division with and without compression (Section 4.5), as well as
- set containment join, namely
  - adaptive pick-and-sweep join (Section 4.2.5).

The above algorithms were implemented in the Java programming language. They make use of the Java class library *XXL (eXtensible and fleXible Library for data processing)*, under development of the database research group at the University of Marburg [BBD+01, CHK+03]. XXL offers a variety of classes for building query processors. To give an idea of the XXL library, we just name a few exemplary classes: *BTree*, *Buffer*, *HashGrouper*, *NestedLoopsJoin*, and *Predicate*. Interestingly, XXL already provides a class called *SortBasedDivision* that implements the sort-merge division algorithm, discussed in Section 3.3.2.2. We have used version 0.99 Build 2002-08-05 of XXL.[1]

The division algorithms have been implemented so they can be used as an internal agorithm for the set containment join algorithms. We do not present performance experiments for classical division algorithms. We have not realized set containment join approaches other than adaptive pick-and-sweep join because they are considered inferior [MGM03].

6.1.2 Set-Valued Attributes

For the implementation of the adaptive pick-and-sweep join algorithm, we had to devise a data structure for set-valued attributes because XXL offers only atomic data types. Instead of realizing a straightforward approach, we actually implemented *nested tables*, mentioned in Section 4.1, i.e., an attribute can be a multi-set of rows, where each row may have one or more columns. We decided that the main-memory representation of a nested table in our prototype is a *Vector* of *rows*. In our scenario of adaptive pick-and-sweep join, a set element consists of a single value, hence each row has a *single* column. Of course, a row can be composed of *multiple* columns but this was not needed for our experiments.

In all experiments, an unnested table has two columns: one column representing the set identifier and the other column the set element value. In a nested layout, a row also consists of two columns, one for the set identifier and the other for the Vector of one-column rows for the set elements.

The idea of using a row of size one instead of a set value alone causes a memory overhead because we store in each row the *number of colums*, which is always equal to one, in addition to the value, as we will explain in the next section. We consider this as negligible. In our

[1] XXL is free software, running under GNU Public License. Hence, everyone is allowed to modify the software.

implementation, both the row size information and the set value are integer numbers and thus the memory occupied is merely doubled.

6.1.3 Table Storage

The input data of the query execution engine is stored in files. The synthetic data generation tool, like the original tool by IBM [AS94], produces text data. Text data is also the format that is used for many real-life datasets that are publicly available.[2] The dataset BMS3 is also provided in a text format [ZKM01].

Our prototype was built to simulate the query execution engine of a real RDBMSs. Hence our system does not process base tables that are stored as a text file. In fact, it could do so. In contrast, as with any RDBMS, the base tables are stored in a proprietary, binary format. In the following, we briefly explain the text and binary formats for the input tables required by our prototype. While text files are only used to import datasets, binary files are used throughout all query processing. In Section 6.1.4, we will describe the transformation of text data into binary data in more detail.

6.1.3.1 Text Format

We have used the XXL operator *FileMetaDataCursor* to read a text file from disk. Each line of the text represents a row where the column values are separated by whitespace and the first two lines specify the attribute names and data types. For example, the following excerpt represents the normalized table r_1 in Figure 2.1(a) on page 10.

```
a       b
Integer Integer
1       1
1       4
2       1
2       2
2       3
2       4
3       1
3       3
3       4
```

An equivalent nested representation of this table, shown in Figure 2.3(a) on page 10, is represented in an input text file as follows:

```
a       b1
Integer Array
1       [1,4]
2       [1,2,3,4]
3       [1,3,4]
```

[2]For example, the datasets of the UCI KDD archive [HB99] of the University of California, Irvine, offers a large collection of real-life datasets for data mining experiments. However, most datasets are used for classifications. Only few and small transaction datasets are available.

We have adapted the class *FileMetaDataCursor* such that also data in a nested layout can be imported from a text file.

6.1.3.2 Binary Format

We had to devise a serialized representation of tables having a nested or unnested layout to store them on disk. Since the tables for our performance tests consist of integer numbers only, we decided to use the Java methods *java.io.DataOutputInteger.writeInt()* and *java.io.DataInputInteger.readInt()* to store and retrieve an integer value, respectively. An integer consumes 4 bytes on disk. All of the values described in the following are integers.

In our implementation of an *unnested* table layout, we represent a two-column transaction row (*t#*, *i#*) as follows:

1. number of columns c (4 bytes, $c = 2$),
2. *t#* value (4 bytes), and
3. *i#* value (4 bytes).

The rows are concatenated and stored in a file. Hence, an unnested table with a total of i rows (or items), occupies $4(3i) = 12i$ bytes on disk.

In our implementation of a *nested* table layout, we represent a two-column transaction row (*t#*, *i#set*) as follows:

1. number of columns c (4 bytes, $c = 2$),
2. *t#* value (4 bytes),
3. number of elements in *i#set* (4 bytes), and
4. sequence of items. An item is a "row" itself. Therefore, we store it as follows:
   (a) number of columns c' (4 bytes, $c' = 1$) and
   (b) sequence of item values ($4c'$ bytes $= 4$ bytes).

The rows are concatenated and stored in a file. Hence, a nested table with t transactions and a total of i items (rows) occupies $4(3t + 2i) = 12t + 8i$ bytes on disk.

EXAMPLE 7: Consider the dividend relation r_1 in Figure 2.2 on page 10 that has the schema $R_1(a,b)$, where a is the set identifier and b is a set element, i.e., the table has an *unnested* layout. Our prototype would store this table as the following sequence of integers into a file:

- 2, 1, 1,
- 2, 1, 4,
- 2, 2, 1,

- 2, 2, 2,
- 2, 2, 3,
- 2, 2, 4,
- 2, 3, 1,
- 2, 3, 3,
- 2, 3, 4.

The size of this file would be $4(3 \cdot 9) = 108$ bytes.

Suppose, the table contents would be stored in a *nested* layout, as shown in Figure 2.3 on page 10. Here, the table r_1 has the schema $R_1(a,b)$, where a is the set identifier and b contains the set elements. In our implementation, this table is stored as the following sequence of integers into a file:

- 2, 1, 2, 1, 1, 1, 4,
- 2, 2, 4, 1, 1, 1, 2, 1, 3, 1, 4,
- 2, 3, 3, 1, 1, 1, 3, 1, 4.

The table has $t = 3$ transactions and $i = 9$ rows. Hence, the file size for this table would be $4(3 \cdot 3 + 2 \cdot 9) = 108$ bytes. It is only by chance that the file size for storing the table is the same for both layouts.

□

6.1.4 Buffer Management

The behavior of the buffer manager is specified by three parameters: the size of a single block that is transferred between disk and memory, the number of buffers occupied by the buffer manager with each buffer holding a block, and the replacement strategy for buffers. We have decided to employ the popular least-recently used (LRU) strategy for our implementation.

XXL does not provide a ready-to-use disk-based buffer manager. We implemented a class called *BufferedFileMetaDataCursor* that extends the functionality of the *FileMetaDataCursor*, mentioned in Section 6.1.3.1. Using this new operator, a text file is taken as input and a binary, typically more memory-efficient file is created. It is optional to use a binary file as input. If the buffer is large enough to hold the entire input table, no binary data is written to disk. Otherwise, depending on the replacement strategy, some data are materialized. This transformation from the text format to the binary format is done by the constructor of the class. The first fetch operation from the cursor will access only the *binary* representation of data. Hence, all I/O operations that we measured were accesses to tables in a binary format, except for B-trees, which have a special internal data representation. If a row is still loaded into the buffer, no disk access is required. Otherwise, a block is read from the binary file on disk and the row is extracted from the block in the buffer.

The binary format is usually a more compact representation of the data. For example, the text representation of the XXL input type *Number* is transformed into a Java *BigDecimal* object and then serialized and written to disk. For all datasets of our experiments, we used an integer text representation, which is mapped to a Java *Integer* object. A nested data type, indicated by the new data type *Array*, is first transformed into a Java *Vector* of rows (new XXL class *HashCodeArrayTuple*) and then written to disk by serializing the atomic column values of each row, as explained in Section 6.1.3.2.

After a buffer has been constructed for a base table, the buffer is "warm," i.e., the rows that have been consumed least recently from the text input file are still contained in the buffer slots. If an objects needs to access the first row of a table and the table does not fit into the buffer, then the buffer manager will access the binary file on disk because only the rows that have been least recently read are still in the buffer, which are the last rows.

6.1.5 System Characteristics

The test environment for the Java query processor prototype was a 4-CPU Intel Pentium-III Xeon PC with 900 MHz, 4 GB main memory, and Windows 2000 Server with a local SCSI disk.

The read and write access to a table is buffered by a buffer manager, described in the previous section. Each input table has its own buffer to make the I/O behavior of the algorithms easier to understand. The block size was set to 8 KB (8192 bytes). In order to study the I/O behavior of a real RDBMS for large datasets, we have decided to set the number of buffer slots to 128 for each input buffer. The entire buffer for an input table is thus 1 MB. This number is low, but we consider it as realistic when we put the base table sizes in relation to the size of tables in a real-life database.

Let us briefly explain the intention of our decision. The largest input table used in our experiments occupies (in an unnested layout) 116.8 MB on disk, as shown in Figure 6.2. Hence, an input buffer of 1 MB is two orders of magnitude smaller than the materialized data. The main memory size of a typical database server (approximately 10 GB) is about 1–2 orders of magnitude smaller than the size of a typical data warehouse (circa 1 TB). For example, according to the TPC-H benchmark specification [Tra02], a TPC-H database with a scaling factor of 1 occupies around 956 MB of storage, while the data warehouse fact table *lineitem* alone has a size of 641 MB. Hence, the fact table, which is likely to be subject to a typical containment query, has a share of 67% of the total database size, i.e., it has the same order of magnitude as the entire database. Therefore, the size of a base table compared to the main memory size differs by two orders of magnitude. The same difference is between the buffer size and the base table size in our test environment. The buffer settings are summarized in Table 6.1.

6.1.6 Synthetic Datasets

The synthetic datasets that we created comprise datasets that were primarily used for the dividend, called *original* datasets, as well as datasets for the divisor, called *query* datasets.

Category	Parameter	Value
Table buffer	Block size	8 192 bytes
	Blocks	128
Sort buffer	Block size	8 192 bytes
	Blocks total	128
	Blocks final	44
	Fan-in	43 690
	Fan-in final	15 018

Table 6.1: System parameters of the Java query processor

6.1.6.1 Original Datasets

We have created three transaction datasets with a vertical, unnested layout that were used as the basis from which we picked transactions to create smaller datasets. A row of a dataset consists of two integer numbers, the transaction identifier *t#* and the item identifier *i#*. A row occupies 12 bytes on disk in our implementation, as mentioned in Section 6.1.3.2. Each dataset has a different order of magnitude for the average transaction size of approximately 10, 100, and 1 000 items.

The datasets used for the experiments have been produced using a Java reimplementation of the well-known IBM data generation tool mentioned in [AS94]. To realize our synthetic transaction generation tool, we used some source code of the *SDSU Java Library* [SDS02].[3] Similar datasets have been used in numerous publications for association rule discovery algorithms. We suggest the following naming convention for a dataset: We indicate the size of a transaction (T) and the number of transactions (D) by "*x*E*y*," where $x \geq 0$ is the factor (a real number) and $y \geq 0$ the exponent (an integer number) of the expression $x \cdot 10^y$. For each dataset, we have created another dataset that represents a manipulated collection of subsets of transactions from the original dataset. This extraction procedure will be described in the following section. The original dataset and the derived dataset are abbreviated by the suffix "D" for "data" and "Q" for "query," respectively. For example, the original dataset T1E1.D1E6.D has $1 \cdot 10^6 = 1\,000\,000$ transactions, where each transaction has $1 \cdot 10^1 = 10$ items on the average. The datasets' characteristics are summarized in Table 6.2, where we give the average, minimum, and maximum size of a transaction, as well as its standard deviation σ.

One of the input parameters of the transaction data generator is the number of distinct items. For the datasets T1E1.D1E6 and T1E2.D1E5, this parameter was set to 1 000, while for dataset T1E3.D1E4, it was set to 10 000. However, the real values are 771, 859, and 5 461, respectively, because the data generation procedure is quite complex. The transactions reflect an inherent pattern, the frequent itemsets. Each itemset, which is the basis for a transaction, is picked from a pool of itemsets, where the number of itemsets in the pool is user-specified, and the probability

[3] The library was developed at the San Diego State University and distributed under GNU General Public License (GNU GPL).

Dataset	Trans-actions	Rows	Items	Transaction size				Disk (MB)
				avg.	min.	max.	σ	
T1E3.D1E4.D	10 000	9 959 457	5 461	995.9	869	1 128	33.4	114.0
T1E2.D1E5.D	100 000	9 886 943	859	98.9	54	149	11.2	113.1
T1E1.D1E6.D	1 000 000	10 208 647	771	10.2	1	35	3.9	116.8
T1E3.D1E4.Q	9 546	1 020 893	5 440	106.9	1	1 112	304.1	11.7
T1E2.D1E5.Q	95 353	1 254 606	858	13.2	1	144	29.6	14.4
T1E1.D1E6.Q	989 981	5 612 509	771	5.7	1	30	3.1	64.1

Table 6.2: Overview of datasets for set containment test algorithms

Dataset	Transactions		Frequent itemsets		Items
	number	avg. size	number	avg. size	
T1E3.D1E4.D	10 000	1 000	1 000	20	10 000
T1E2.D1E5.D	100 000	100	1 000	10	1 000
T1E1.D1E6.D	1 000 000	10	1 000	5	1 000

Table 6.3: Input parameters for generating original datasets

function for picking a certain itemset has a certain user-specified distribution. Furthermore, there are user-specified correlation and corruption factors that influence the probability if an item of a previous transaction is repeated in the next transaction to be generated or if an item from an itemset is dropped in the current transaction. The numeric input parameters for the synthetic data generation procedure are summarized in Table 6.3.

During a performance experiment, we measure the execution time of a query execution plan for input tables that are samples of different size for each type of dataset. The sampling procedure scanned the entire table and selected a fraction $p \in [0,1]$ of all rows from the table. The ith row was selected if $\lfloor pi \rfloor \neq \lfloor p(i+1) \rfloor$. For example, suppose we have a table with ten rows with the rows numbered from 0 to 9, and $p = 0.7$. In this case, the rows number 1, 2, 4, 5, 7, 8, and 9 are selected. We decided to create samples that are a fraction of an original dataset for the values 0.5, 0.1, 0.05, 0.01, 0.005, 0.001, 0.0005, and 0.0001. We used the XXL *Selection* operator for the implementation of this procedure.

The original synthetic dataset generator by IBM, mentioned in [AS94], has some default settings that we left unchanged for the dataset generation with our own tool. Let us briefly summarize these settings. The probability function of the transaction size has a Poisson distribution with a mean of the average transaction size minus one. The probability function of the frequent itemset size has a Poisson distribution with a mean of the average frequent itemset size minus one. The probability function of item distribution has an exponential distribution with a mean of 1. The correlation has an exponential distribution with a mean of 0.5. The corruption probability function has a distribution that is based on a normal distribution.

Dataset	Transaction probability		Item probability	
	drop (p_1)	modify (p_2)	drop (p_3)	add (p_4)
T1E1.D1E6.Q	0	0.9	0.500	0
T1E2.D1E5.Q	0	0.9	0.970	0
T1E3.D1E4.Q	0	0.9	0.997	0

Table 6.4: Input parameters for generating query datasets

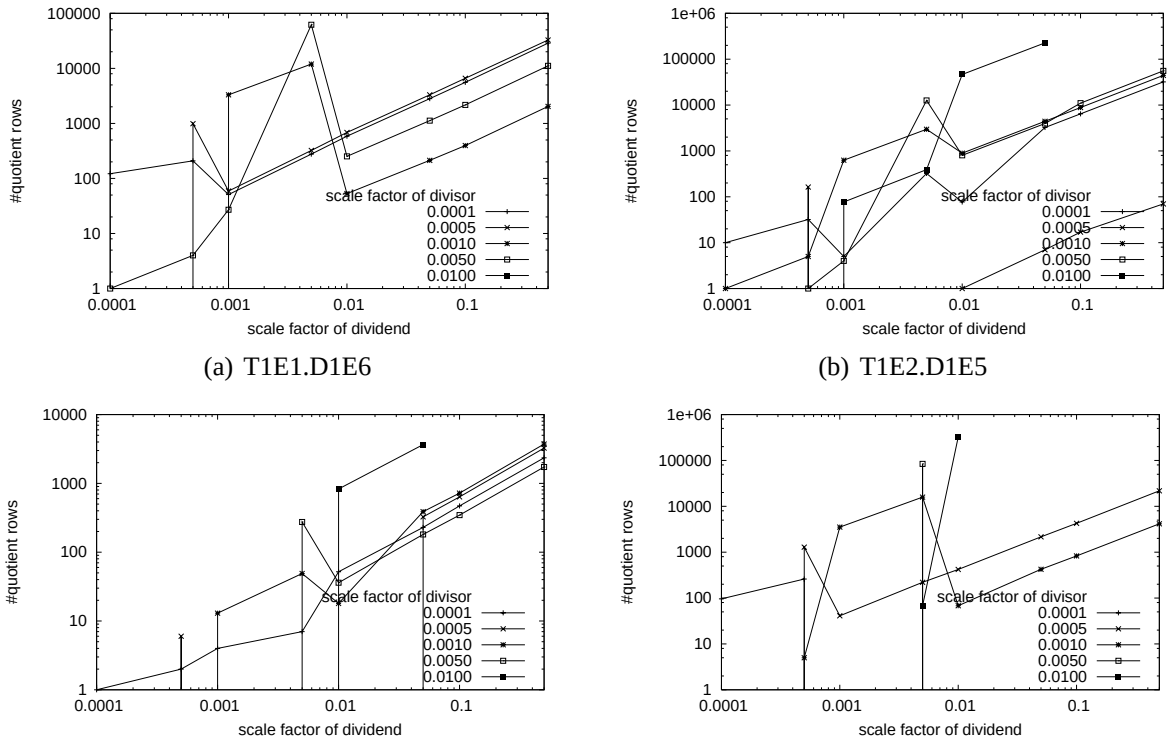
6.1.6.2 Query Datasets

The dividend tables have been generated according to the data generation procedure described in Section 6.3. In order to obtain a certain amount of quotients, we decided to create a manipulated version of each dividend dataset that still bears some resemblance to the dividend's data characteristics. Our generic manipulation procedure can be described as follows. It has four probability threshold values p_1, p_2, p_3 and p_4 as input. The corresponding probability funtions are all uniformly distributed. The probability p_1 is the transaction drop probability, p_2 is the transaction modify probability, p_3 is the item drop probability, and p_3 is the item add probability. For each original transaction, the manipulation procedure does the following:

- Drop the transaction with probability p_1.
- Otherwise, manipulate the transaction with probability p_2 as follows:
  - For each original item, drop the item with probability p_3.
  - Otherwise, for as many times as there are items in the original transaction, add a new random item to the transaction with probability p_4.
- Otherwise, retain the transaction.

The parameter setting for the query dataset generation is summarized in Table 6.4. Tables C.1–C.3 in Appendix C on pages 179–180 give a detailed overview of the dataset sizes for both original and query datasets. The reason for setting the paramters as indicated in the table is that we wanted to create several but not too many true subsets of original transactions. With this setting, we did not drop a transaction as is ($p_1 = 0$). However, a transaction was dropped if it happened that all items were dropped. About $p_2 = 90\%$ of the transactions were subject to tests whether to drop an item or not (p_3). No items were added ($p_4 = 0$).

The number of quotient rows for a varying divisor table cardinality is illustrated in Figures 6.1(a)–(c) for the synthetic datasets. Note that when the dividend is small, we see erratic movements of the plot for an increasing dividend size. Since we deal with relatively small divisors, a slight increase in the dividend size can produce a drastically different quotient table. This is a natural effect since when we compare two *small* divisor datasets, we may find that one of them contains *much more* subsets of some dividend sets than the other because of the randomized query table generation method. This effect is much weaker for large divisors.



(a) T1E1.D1E6 (b) T1E2.D1E5

(c) T1E3.D1E4 (d) BMS3

Figure 6.1: Number of quotients

We have made a few experiments with divisors of a scale factor larger than 0.01. Unfortunately, all algorithms required several days to finish the execution. It is reasonable to focus on the number of quotients produced for *large* dividends with a scale factor of at least 0.05. As we can see from the graphs, an increase in the quotient number is proportional to the size of the dividend, as we expect intuitively.

6.1.7 Real-Life Datasets

We supplemented our collection of synthetic datasets by a real-life dataset that we call *BMS3*,[4] which is provided by Blue Martini Software. It was introduced and compared to other transaction datasets in [ZKM01]. BMS3 consists of several years of point-of-sale transactions of an electronics retailer. The items are product categories. A transaction represents the set of categories from which products have been purchased at one time. While Table 6.5 on page 139 summarizes the data characteristics of BMS3, Table C.4 in Appendix C on page 181 gives an overview of the dataset sizes we extracted from BMS3 by the same sampling method that was used for the synthetic datasets, described in the previous section.

[4]BMS3 is originally called *BMS-POS* in [ZKM01]. We have used two further datasets of Blue Martini Software for the experiments with frequent itemset discovery algorithms. See Section 6.3 for details.

We did not create query datasets from BMS3 but we used the datasets for both dividend and divisor. The reason for this is that we wanted to retain the fact that the dataset is *real-life* and we refrained from artificially distorting the point-of-sales transactions in any way.

The number of quotient rows for the synthetic dataset BMS3 is shown in Figure 6.1(d). The same comments that we made on the plotted graphs of the synthetic datasets apply to the real-life dataset.

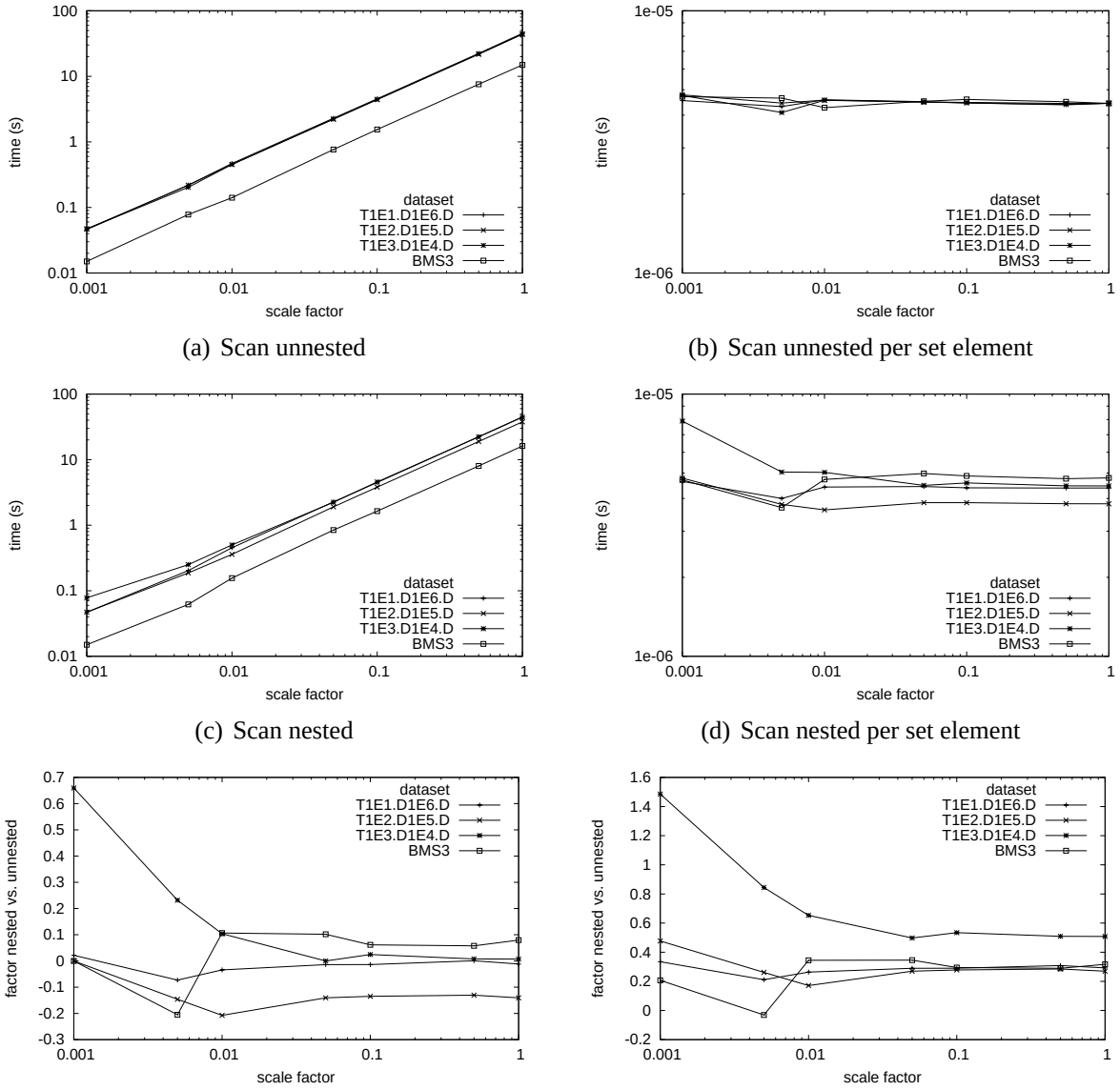
6.2 Overview of Experiments

We have conducted several experiments with the physical operators realizing set containment division and set containment join. The experiments investigate the following scenarios:

- Nesting vs. unnesting
  - Time to scan data
  - Time to switch the table layout on-the-fly
- Sort-based set containment division
  - Time to sort input data
  - Time to execute operator with and without previous sorting
- Hash-based set containment division
  - Time to execute operator (based on unsorted data)
- Subset index set containment division
  - Memory savings by compression
  - Time to execute operator with a subset index on the dividend or divisor
  - Time to execute operator with a compressed or uncompressed subset index on the divisor
- Adaptive pick-and-sweep join
  - Time to execute operator for a varying number of partitions
  - Values of the replication and comparison factors for a varying number of partitions

All experiments cover some or all types of datasets described in Sections 6.1.6 and 6.1.7. We omit the results for some type of dataset when we believe that they duplicate the characteristics found for the other datasets.

Apart from an analysis of each algorithm itself, we compared their performance. However, this comparison does not allow immediate implications on which algorithm is the absolute best



(a) Scan unnested

(b) Scan unnested per set element

(c) Scan nested

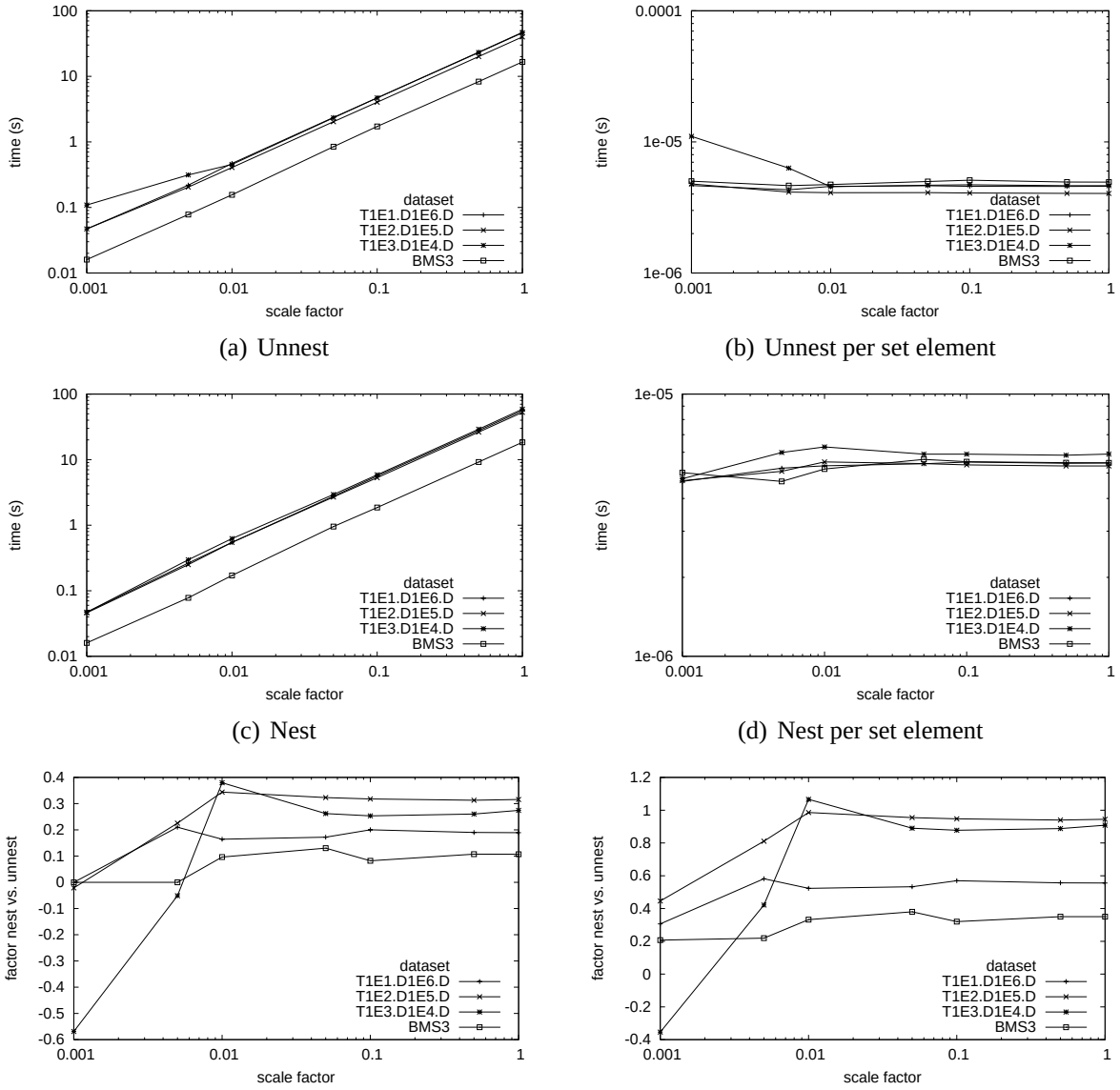
(d) Scan nested per set element

(e) Scan nested vs. unnested

(f) Scan nested vs. unnested per byte

Figure 6.2: Time to scan unnested and nested datasets

one. To make such a statement, we would have to vary too many system and algorithm parameters. Although our experiments cover a wide range of test settings, we cannot conclude general statements from our results. Hence, the reader should focus on the results of each algorithm itself to understand its behavior, not on the comparison results only.



(a) Unnest

(b) Unnest per set element

(c) Nest

(d) Nest per set element

(e) Nest vs. unnest

(f) Nest vs. unnest per bytes

Figure 6.3: Time to unnest and nest datasets

6.2.1 Nesting and Unnesting

We have made several experiments that measure the execution time to scan unnested and nested datasets as well as the cost for nesting and unnesting previously unnested and nested data, respectively. Our motivation for the experiments is that set containment join and set containment division algorithms require input data of a different layout. The same type of dataset occupies a different amout of size on disk for the two layouts. Hence, when we compare these algorithms, we have to take into account that different I/O is incurred.

As described in Section 6.1.5, the tables are buffered, so not every table access causes I/O.

For very small datasets, no I/O was caused at all. Figure 6.2 summarizes the results. For a clearer presentation, we only show the results for the original datasets and omit the results for the scan times of the query datasets. We found similar results for the query datasets.

Since the synthetic datasets occupy almost the same size on disk, the pure scan time of unnested data is almost the same. The scan time for the real-life dataset BMS3 is considerably lower due to a smaller disk size.

We found slight differences in execution time for the scan of nested tables: The larger the average transaction size, the longer the CPU time to build a row in a nested layout. Figure 6.2(d) shows that dataset T1E3.D1E4.D, which has 995.9 transactions on the average (the largest of all datasets) requires more time per element than dataset BMS3, which has an average transaction size of 6.5 (the smallest of all datasets).

The difference in the scan time for unnested and nested data is illustrated in Figure 6.2(e). The graphs visualize the quotient

$$\frac{t_{\text{scan nested}} - t_{\text{scan unnested}}}{t_{\text{scan unnested}}}$$

for different dataset sizes. For large datasets, the scan time for BMS3 is 8% slower when the data is nested compared to unnested. On the other hand, for dataset T1E3.D1E4.D, the scan time is 14% slower when the data are nested.

There is a second aspect to this comparison whose results are sketched in Figure 6.2(f). The quotient does not reflect the fact that the unnested and nested data require a different amount of I/O due to their different sizes on disk. Hence, we also computed the quotient

$$\frac{t_{\text{scan nested}}/s_{\text{nested}} - t_{\text{scan unnested}}/s_{\text{unnested}}}{t_{\text{scan unnested}}/s_{\text{unnested}}},$$

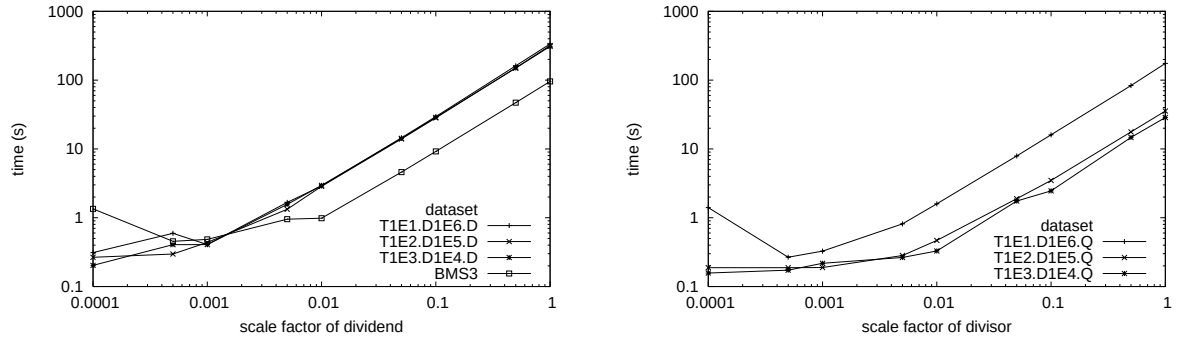
where s_{unnested} and s_{nested} are the dataset disk sizes for the unnested and nested layout, respectively. The exact dataset sizes are given in Appendix C on page 179. We see that for large datasets, the relative cost of scanning nested data is between 27% and 51% higher than for scanning unnested data. By "relative" we mean that we compare the execution times per byte.

We have also studied the performance for nesting and unnesting data that was previously unnested and nested, respectively. The results are illustrated in Figure 6.3.

Similar to the scan experiments, we measured the factor by which the nest operator is more or less efficient than the unnest operator. Again, we analyzed two different aspects of this factor. The first aspect, illustrated in Figure 6.3(e), is the total time difference, irrespective of the fact that the unnested and nested datasets require a different amount of I/O. For this aspect, we compute the quotient

$$\frac{t_{\text{nest}} - t_{\text{unnest}}}{t_{\text{unnest}}}.$$

Taken all datasets together, the effort for nesting was between 9% and 32% higher than for unnesting.



(a) Dividend (original) datasets (b) Divisor (query) datasets

Figure 6.4: Time to sort datasets

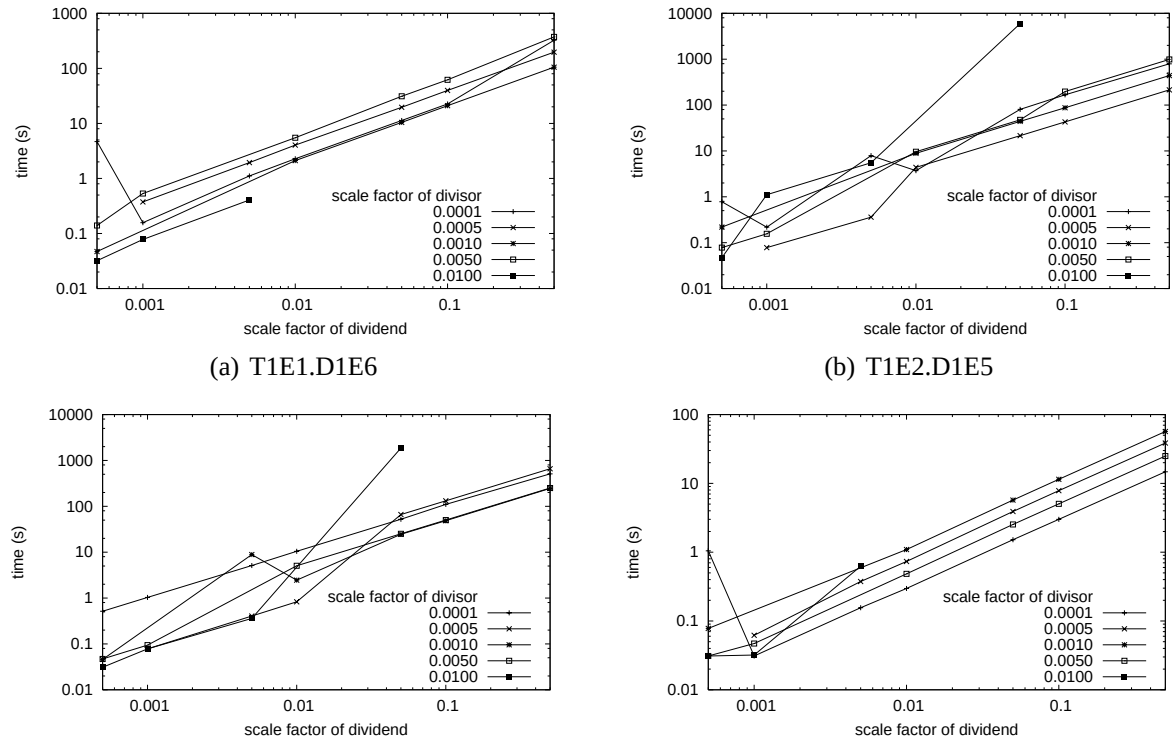
The second aspect compares the execution times while eliminating the difference in dataset size of the two layouts. Hence, we computed the quotient

$$\frac{t_{\text{nest}}/s_{\text{nested}} - t_{\text{unnest}}/s_{\text{unnested}}}{t_{\text{unnest}}/s_{\text{unnested}}}.$$

The results are shown in in Figure 6.3(f). One can see that for large datasets, nesting is between 29% and 94% more expensive than unnesting data. Hence, we learn that, due to our implementation of a binary nested and unnested table layout, the CPU cost for reading a nested table is higher than for an unnested table since each element is transformed into a separate row object. Creating a new row incurs the cost of allocating meta data for it, like the column name (which is a single character by default in our implementation) and the column type (which is always integer). Of course, this time is low compared to the I/O time to read data from disk. However, we could measure the effect of the CPU-intensive task of reading nested data. The CPU cost for reading the same unit of storage of nested data with long transactions (T1E3.D1E4.D) is higher than for short transactions (T1E1.D1E6.D) although the same number of rows have to be created for the *elements*. However, fewer row objects for *transactions* are created for datasets with long transactions. For example, for scale factor 1, the total number of row objects is $10\,000 + 9\,959\,457$ for dataset T1E3.D1E4.D compared to $1\,000\,000 + 10\,208\,647$ for T1E1.D1E6.D, according to Table 6.2. Hence, approximately 12.4% more row objects are created for dataset T1E1.D1E6.D than for T1E3.D1E4.D, which caused the difference in CPU cost in Figure 6.3(f) between these datasets.

6.2.2 Sort-Based Set Containment Division

Sort-based set containment division is an algorithm that requires its input tables be first grouped on the transaction column *t#* and second sorted on the item column *i#*. We measured the performance of this algorithm with base tables that had the required data properties. In practice, it may happen that the inputs are already structured this way either because it was stored with this structure physically on disk, or an index is defined on (*t#*, *i#*), or the input tables have been



(a) T1E1.D1E6 (b) T1E2.D1E5

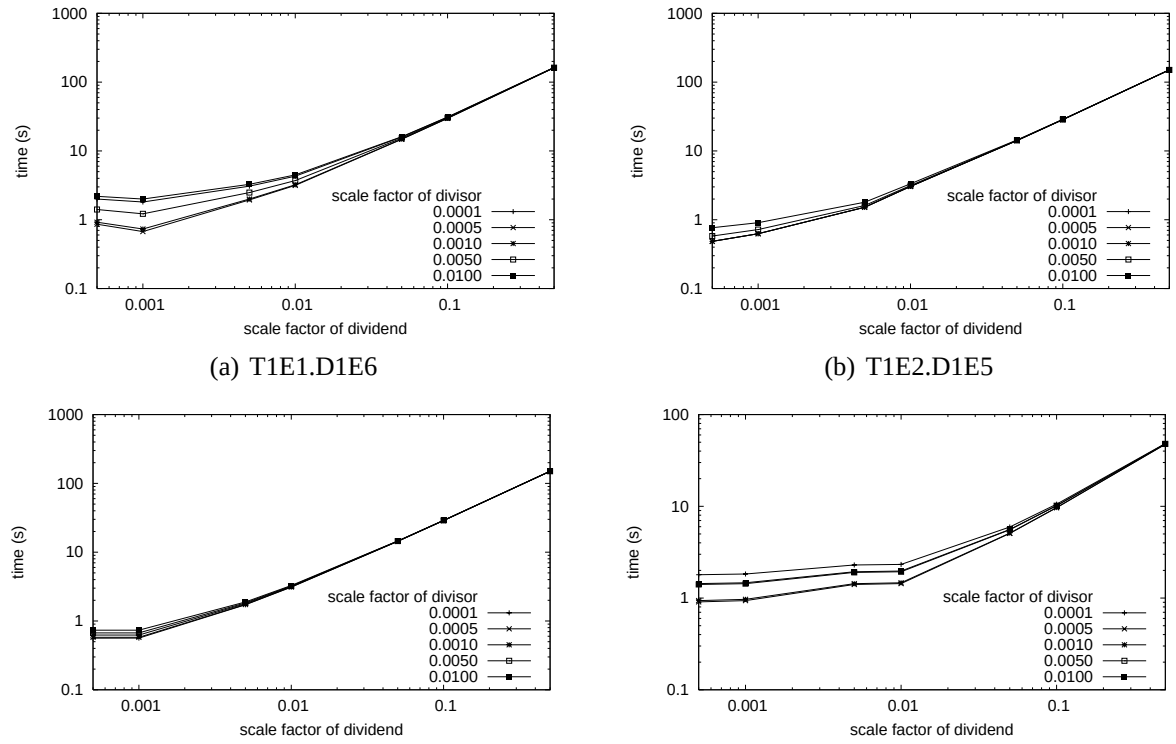
(c) T1E3.D1E4 (d) BMS3

Figure 6.5: Sort-based set containment division for sorted data

produced by *sort* and/or *group* operators.

In addition to measuring the merge-style processing of sort-based set containment division, we have measured the time to sort the input tables on (*t#*, *i#*), both in ascending order, based on a table that was sorted on *i#* in ascending order, i.e., the input for this sort operation was considerably unsorted. Its execution times are illustrated in Figure 6.4 for all datasets. We used an external merge-sort algorithm that is provided by the XXL class *MergeSorter* and which is based on Graefe's description of external sorting [Gra93]. It creates sorted runs and merges them recursively. According to the XXL *javadoc* documentation, the "open-phase" of this algorithm uses replacement selection to create sorted runs, which are on average twice as large as the memory available. Then, the runs are merged until *finalFanIn* intermediate runs are left. This internal parameter of the *MergeSorter* class is derived from several tuning knobs of the algorithm. In the "next-phase" or "fetch-phase" the remaining runs are merged on demand, i.e., no complete sorted run has to be stored temporarily. Except for the sort buffer size, we have left the default settings of XXL unchanged. They are summarized in Table 6.1 on page 117.

The graphs in Figure 6.4 do not appear to expose the typical sort complexity of $O(n\log(n))$ for a dataset of cardinality n. However, we could verify this behavivor for the large scale factors between 0.1 and 1. The performance behavior for the smaller datasets do not show the classical sort complexity function but they are more expensive. Please note that the theoretical complexity and the real behavior come close to each other only for large datasets. We consider this a normal



(a) T1E1.D1E6

(b) T1E2.D1E5

(c) T1E3.D1E4

(d) BMS3

Figure 6.6: Sort-based set containment division: preprocessing time for unsorted data

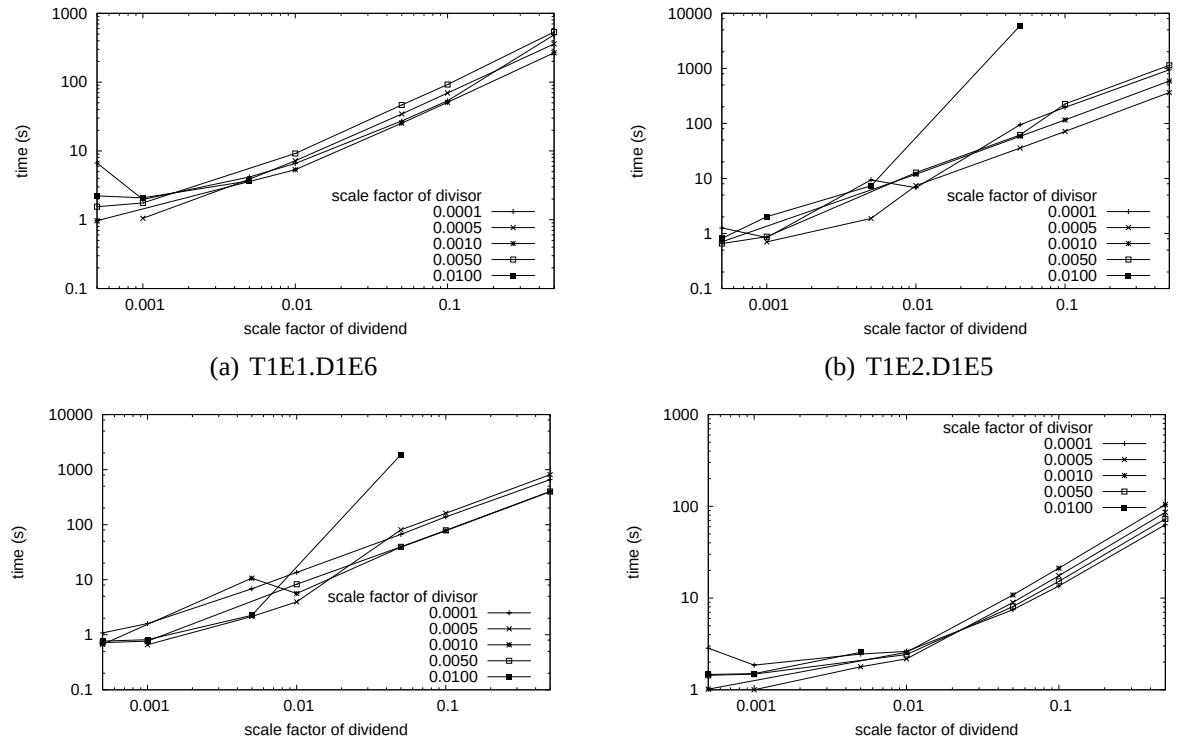
system behavior since the effect of other performance factors like skewed data distribution, and disk I/O have a dominant effect on the experiments with relatively small datasets.

The execution times of sort-based set containment division are shown in Figure 6.5 for each type of dataset, i.e., when the datasets are already sorted. One can clearly see that the execution time is linear with the dividend size for a constant divisor size.

When both input tables are unsorted, we have to sort them first and then apply the sort-based set containment division algorithm. The accumulated time for these two phases are shown in Figure 6.7. These results have been generated by summing up the times to sort the respective original and query dataset shown in Figure 6.4 and the time for the mere division processing in Figure 6.5. For large datasets, the complexity of the set containment division of $O\left(|r_2| + \frac{|r_2|}{\theta_2}|r_1|\right)$ dominates the preprocessing sort effort of $O(|r_1|\log(|r_1|) + |r_2|\log(|r_2|))$.

6.2.3 Hash-Based Set Containment Division

Hash-based set containment division makes no assumption on the ordering of rows in any input table. The algorithm uses the hash-division algorithm internally. Hence, two hash tables are held in memory during each execution of a hash-division operation: the quotient hash table, which stores the quotient candidates, and the divisor hash table, which stores the rows of a single divisor



(a) T1E1.D1E6

(b) T1E2.D1E5

(c) T1E3.D1E4

(d) BMS3

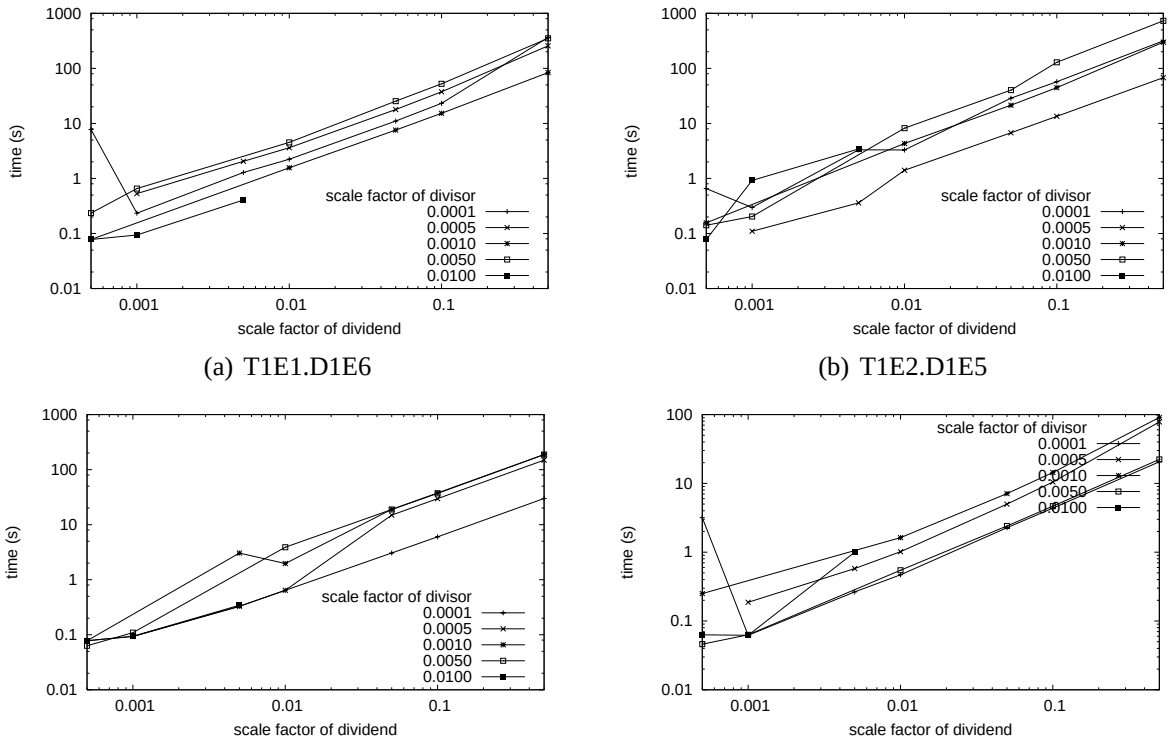
Figure 6.7: Sort-based set containment division for unsorted data

group.

How many buckets should we assign to these hash tables? The number of quotient candidates that are stored in the quotient hash table is equal to the number of transactions in the dividend. Let this number be n_1. Furthermore, let $p(n)$ be a function that returns the smallest prime that is greater than or equal to n. A good estimation for the quotient hash table capacity is $p(n_1)$, provided that the hash-function is good, i.e., it distributes the rows equally across all buckets. For the divisor hash table, a good capacity estimation is $p(n_2)$, where n_2 is the size of the current divisor group. For our performance tests, we decided to keep the hash table capacities constant for all datasets and to use the same capacity for the quotient and divisor hash tables. The largest number of transactions of all datasets is 1 million and occurs in dataset T1E1.D1E6.D, as indicated in Table 6.2. The number of transactions in the real-life dataset BMS3 is only 515 597, as shown in Table 6.5 on page 139. The number of buckets in all hash tables was thus set to $p(1\,000\,000) = 1\,000\,003$.

The experimental results are shown in Figure 6.8. At first sight, they look very similar to the results of sort-based set containment division in Figure 6.7.

We have compared the performance of hash-based set containment division and sort-based set containment division for unsorted datasets, i.e., we have included the time for sorting the datasets first for the sort-based approach. The results are given in Figure 6.9. They reveal, e.g., that for dataset T1E1.D1E6, hash-based set containment division outperforms sort-based set con-



(a) T1E1.D1E6

(b) T1E2.D1E5

(c) T1E3.D1E4

(d) BMS3

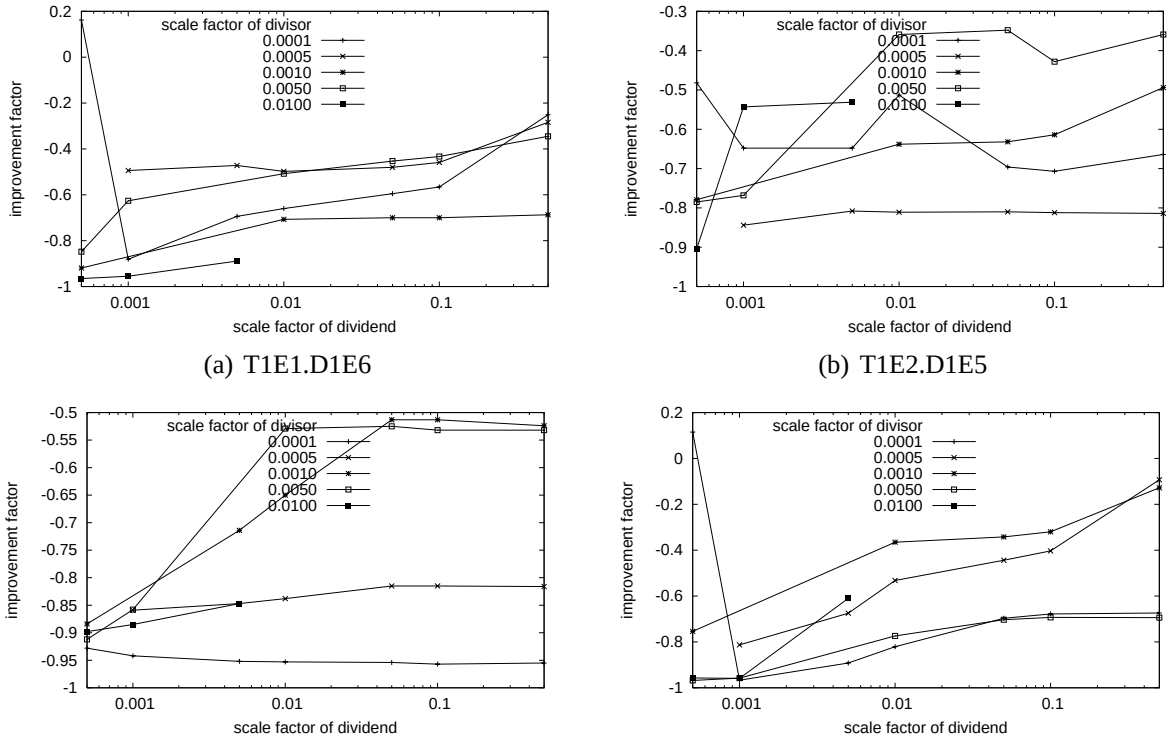
Figure 6.8: Hash-based set containment division

tainment division for medium size dividends, where the scale factor lies between 0.01 and 0.05. It performs worse for very small and very large dividends. A someway similar behavior can be seen for dataset BMS3. Except for the dividend with scale factor 0.005, the hash-based approach never outperforms the sort-based one. However, the difference between the two approaches is smallest for a medium size dividend with a scale factor around 0.05. In theory, one would expect that for very large datasets, any sort-based operator should be able to outperform a hash-based operator because beyond a certain dataset size, the buckets of a hash table will no longer fit into memory and the system performance will degrade due to thrashing. In our implementation of the hash-division (and thus also hash-based set containment division), we did not realize an upper bound of main memory available to the hash tables. Hence, since the test environment offers enough main memory for all datasets, hash-based set containment division did not reveal a degrading performance.

6.2.4 Subset Index Set Containment Division

The table on which we create the subset index has to be first sorted on the group cardinality in descending order, second grouped on the set column, third sorted on the element column, as explained in Section 4.5.4.

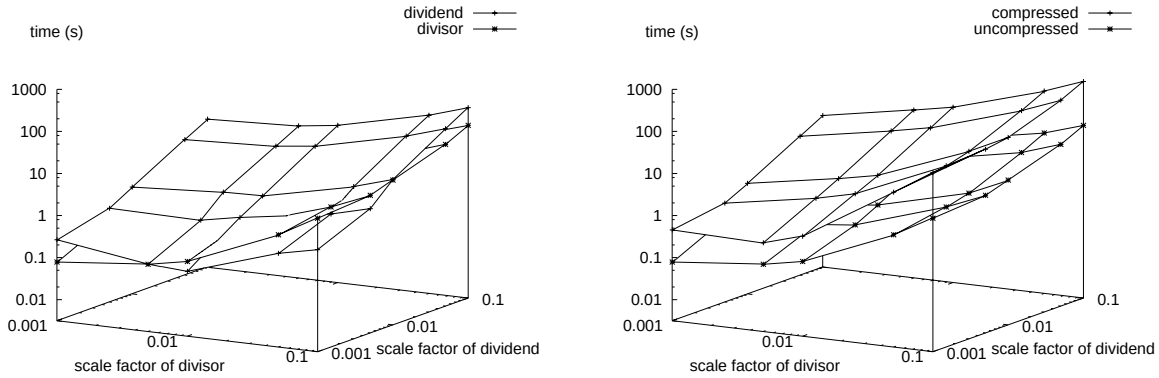
Intuitively, it is reasonable to build an index on the larger of two input tables and to probe the



(a) T1E1.D1E6 (b) T1E2.D1E5

(c) T1E3.D1E4 (d) BMS3

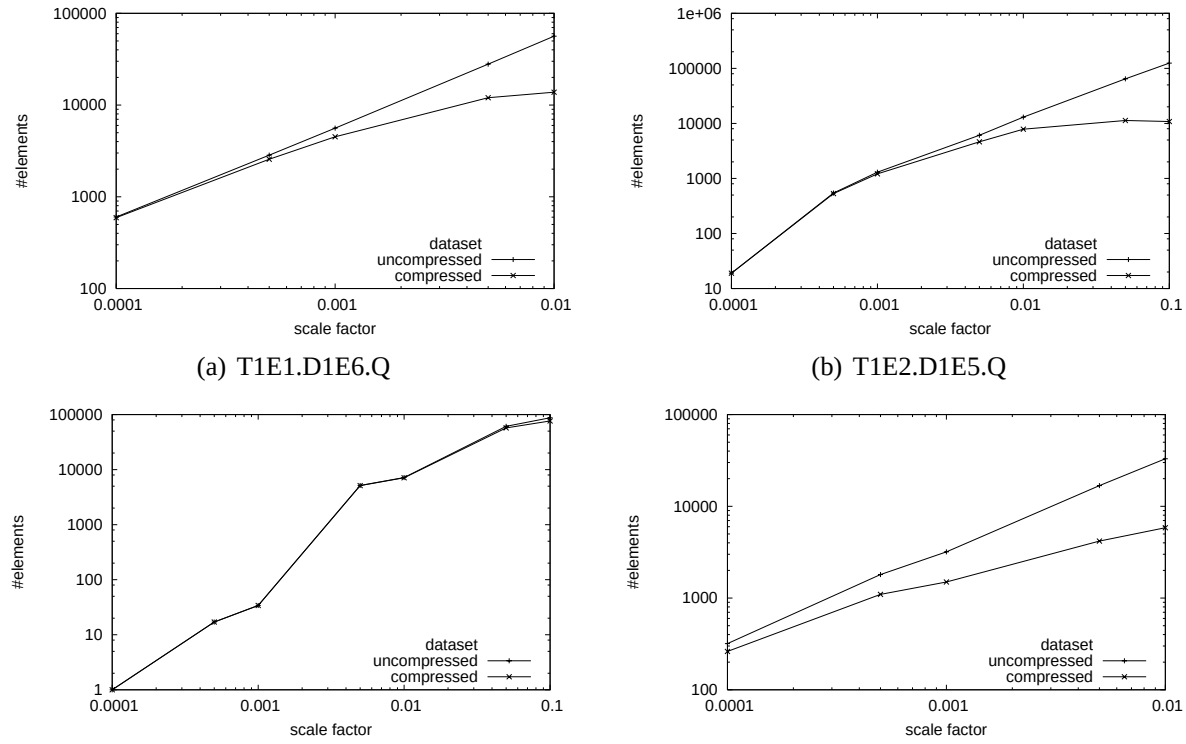
Figure 6.9: Improvement factor of hash-based set containment division compared to sort-based set containment division



(a) Subset index on divisor or dividend? (b) Compressed or uncompressed index on divisor?

Figure 6.10: Influence of subset index options for dataset T1E3.D1E4 on execution time

index with the smaller table. This was confirmed by our experiments with the subset index. For example, in Figure 6.10(a), the execution time of a subset index set containment join is illustrated



(a) T1E1.D1E6.Q (b) T1E2.D1E5.Q

(c) T1E3.D1E4.Q (d) BMS3

Figure 6.11: Subset index size

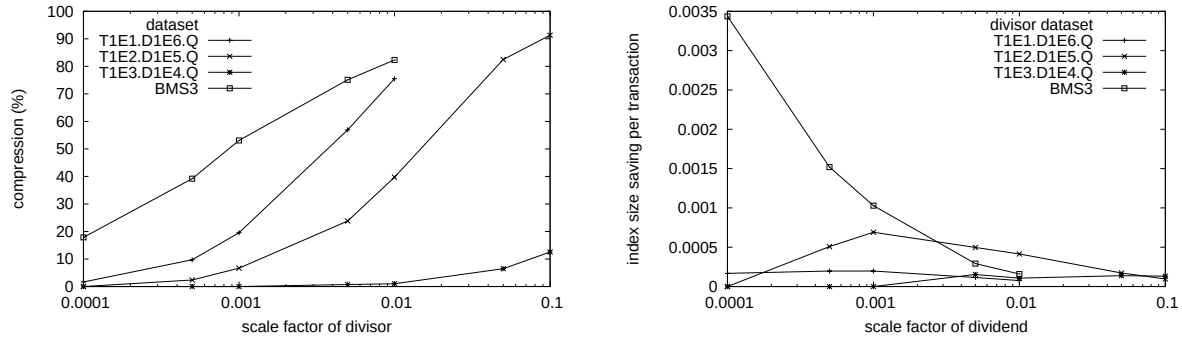
for dataset T1E3.D1E4 for a subset index on the dividend as well as on the divisor. The *xy*-plane represents combinations of dataset sizes for the dividend and the divisor, indicated by the respective scale factors. One can clearly see that a subset index on the dividend is beneficial when the dividend is relatively small and the divisor is relatively large. Whenever the dividend is large and/or the divisor is small, it is better to create the subset index on the dividend.

The performance can be significantly different when we employ an uncompressed or a compressed subset index. For example, consider a divisor subset index on dataset T1E3.D1E4.Q. This scenario is sketched in Figure 6.10(b). We find that compression creates an overhead of about an order of magnitude for a wide range of combinations.

The *absolute* amount of memory saving for compressed nodes in the subset index is illustrated in Figure 6.11. We observe that the smaller the average transaction size, the "earlier" can we save memory by using a compressed subset index for the divisor table, i.e., memory savings come into effect for small datasets.

It is due to the peculiar nature of dataset T1E1.D1E6.Q.P5E-2 that we see a decline in the number of nodes. For large, purely randomly distributed datasets, we would expect a strictly monotonic increasing function, given the size of a randomly generated sample of a randomly generated dataset. The memory saving graphs for the other two datasets do not exhibit such a behavior.

Figure 6.12(a) shows the *relative* size of the compressed index compared to the uncom-



(a) Relative savings (b) Rate of relative savings

Figure 6.12: Memory savings by compression

pressed version, which is assumed to be 100%. One can see that for a small average transaction size (T1E1.D1E6.Q), one can achieve memory savings even for small datasets as compared to datasets with long transactions (T1E3.D1E4.Q). To see the rate at which one can achieve memory savings per transaction, Figure 6.12(b) clearly shows that there are maxima of memory saving per transaction. This figure can be considered the first derivation of the functions plotted in Figure 6.12(a).

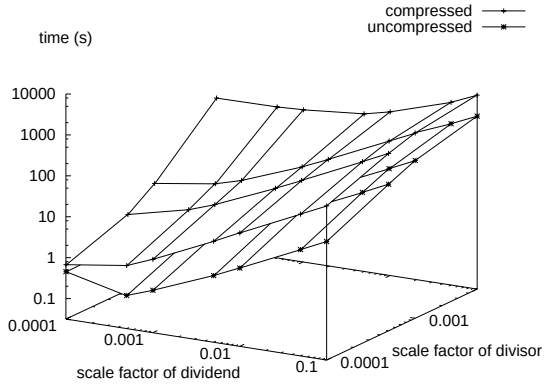
Figure 6.13 illustrates the execution times when an uncompressed and compresses subset index on the divisor table is used.

The time to create a subset index on the divisor is sketched in Figure 6.14 for all types of dataset. We can see that the time for building a compressed index can be up to two orders of magnitude higher than for an uncompressed one.

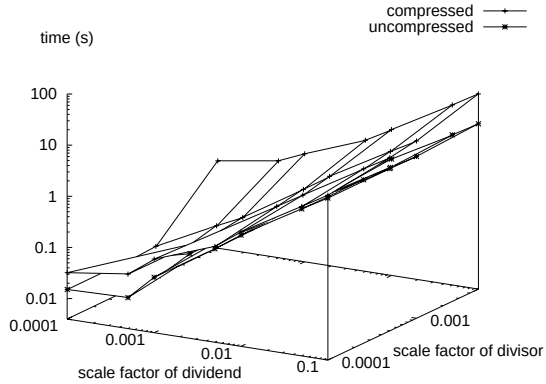
6.2.5 Set Containment Join

The set containment join algorithm has several parameters that have to be set carefully. We have set the signature length to 160 bits. The buffer slot size is 8192 bytes, which the same as for the other experiments. The input data is a buffered nested table in a binary representation with $M = 128$ buffer slots and a block size of 8 KB. We have decided to use four different settings for the number of partitions: $k = 4^i$, where $1 \leq i \leq 4$. The total number of buffer slots assigned to the partitions is $\max\{1, \frac{2M}{k}\}$. It depends on k, the number of partitions as well as on M, the total number of buffer slots assigned to a single input table, i.e., dividend or divisor. In our setting, each of the 4, 16, 64, and 256 parititions had 64, 16, 4, and 1 buffer slots available, respectively.

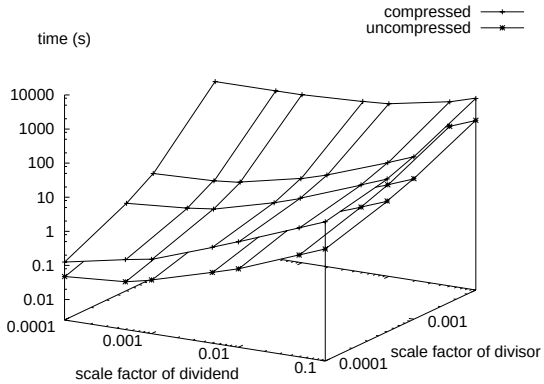
Figures 6.15 and 6.16 show the result of experiments with the synthetical dataset T1E1.D1E6 and the real-life dataset BMS3 for a varying number of partitions. The total execution times are sketched in Figures 6.15(a) and 6.16(a). They show that if both the dividend and divisor have few rows, the best performance is achieved with the fewest number of partitions (4). The longest execution time is produced for the largest number of partitions (256). This effect is reversed if both tables are large. Note that the x- and y-axes in Figures 6.15(d) and 6.16(d) are switched compared to the other coordinate systems to show clearly how the replication factor increases



(a) T1E1.D1E6.Q

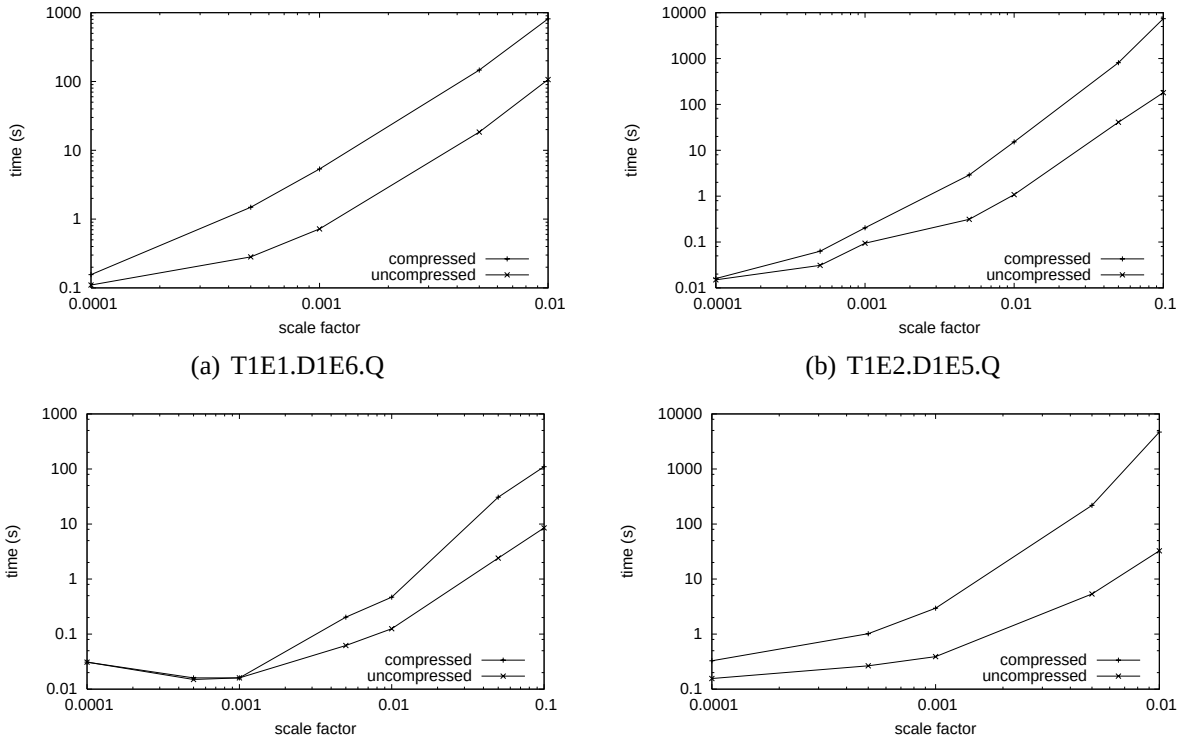


(b) T1E3.D1E4.Q



(c) BMS3

Figure 6.13: Execution time of set containment division with subset index on the divisor table when the dividend size $|r_1|$ increases and/or the divisor size $|r_2|$ decreases.



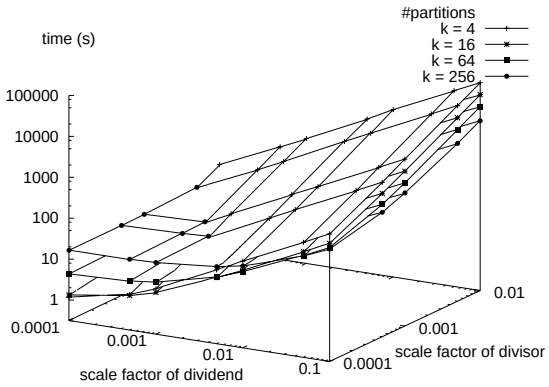
(a) T1E1.D1E6.Q

(b) T1E2.D1E5.Q

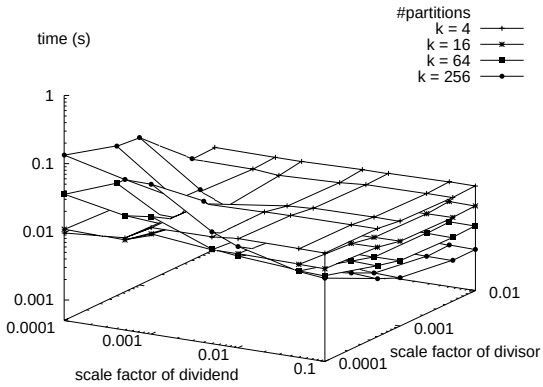
(c) T1E3.D1E4.Q

(d) BMS3

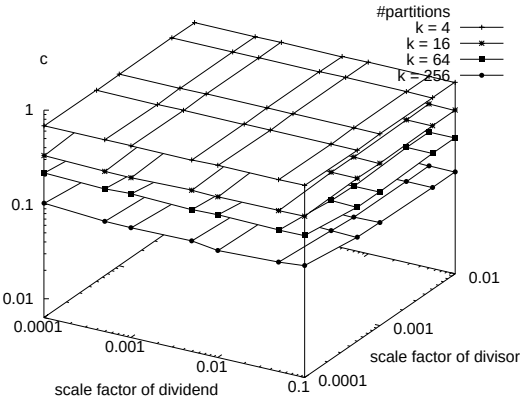
Figure 6.14: Execution time of set containment division with subset index on the divisor table



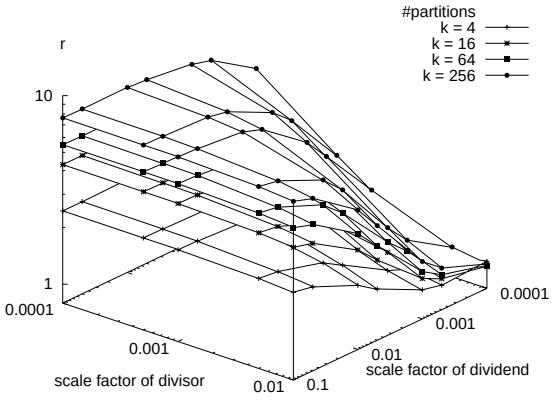
(a) Execution time



(b) Execution time per result row

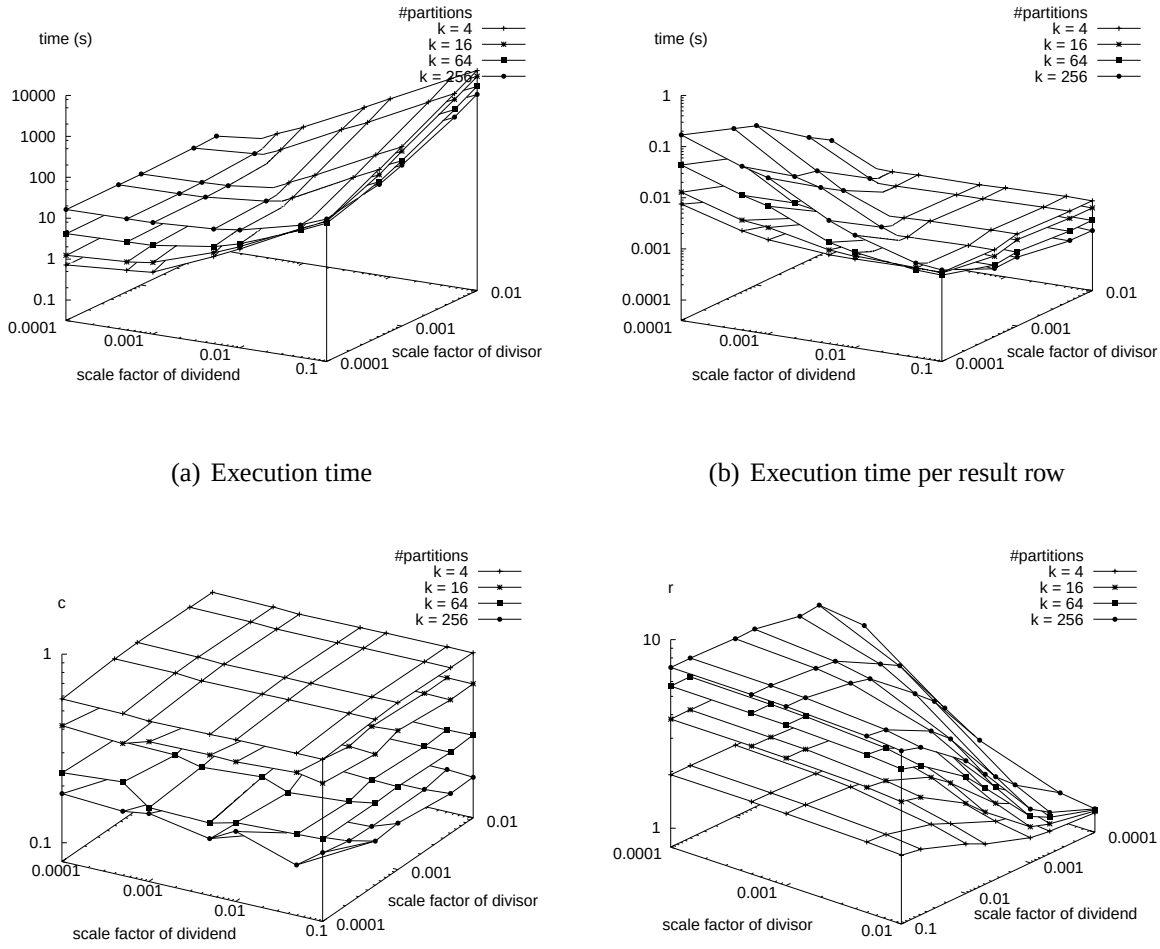


(c) Comparison factor



(d) Replication factor

Figure 6.15: Set containment join with dataset T1E1.D1E6



(a) Execution time

(b) Execution time per result row

(c) Comparison factor

(d) Replication factor

Figure 6.16: Set containment join with dataset BMS3

6.3 Datasets for Frequent Itemset Discovery

In this section, we discuss the characteristics of the datasets used for the performance tests for frequent itemset discovery and explain why we chose them.

We used the same synthetic data generation tool that we used for the experiments with set containment test operators. By convention, i.e., as is done in the literature on association rule discovery, the datasets thus produced are denoted by the average size of a transaction (T), the average size of an itemset pattern used to produce the data (I), and the number of transactions (D). For example, the dataset T5.I5.D100k consists of 100 000 (or 100 kilo) transactions, each containing 5.4459 (roughly 5) items on the average, resulting in 544 590 rows. In this case, the largest transaction produced by the tool has 12 items. The synthetic transaction datasets as well as the real-life datasets discussed next are summarized in Table 6.5.

The datasets occupy 4 bytes for an integer value on disk, i.e., 8 bytes per row. For example, the total space required to store dataset T5.I5.D100k is $8 \cdot 544\,590$ bytes $= 4\,356\,720$ bytes $\approx$ 4.2 MB. This number is the raw size in the binary export file of the data generator. The data are then imported by an RDBMS, which will store the data in data containers that have a larger size than the binary files.

Besides the synthetical datasets, we have used several *real-life* transaction datasets provided by Blue Martini Software [ZKM01] that we call *BMS1* and *BMS2*.[5] BMS1 and BMS2 represent click-stream data of two e-commerce web sites. We used these datasets because the transaction size distribution is different from the synthetic datasets, as illustrated in Figure 6.17 (note the logarithmic scale of the y-axis). The synthetic data has a Poisson distribution of the transaction size with a mean of λ, i.e., there are relatively few transactions of size $1 \leq k < \lambda$, relatively many transactions of size $k \approx \lambda$, and relatively few transactions of size $k > \lambda$. The number of transactions for the real-life datasets, in contrast, is strictly monotonic decreasing with the transactions size. This fact results in different frequent itemset discovery algorithms being optimal for real-life and synthetic datasets, as discussed in [ZKM01].

We have chosen the minimum support threshold for each dataset separately. For the synthetic T5.I5.D10k dataset, we set the minimum support to 1% (100 transactions), for T5.I5.D100k to 0.1% (100 transactions). For the real-life dataset BMS2, we picked a minimum support of 0.2% (155 transactions). The reason why we picked these values is that we wanted to study the performance of the frequent k-itemset generation for very small ($k = 1, 2, 3$) as well as relatively large itemset sizes ($k \geq 4$). To characterize the effect of these parameter settings, Figure 6.18 shows the number of candidate itemsets in C_k and the number of frequent itemsets F_k produced in each iteration k. These numbers are independent of an algorithm. Note the logarithmic scale of the y-axes.

[5] BMS1 and BMS2 are called *BMS-WebView-1* and *BMS-WebView-2* in [ZKM01], respectively. Another dataset, called *BMS3* (or *BMS-POS* in the original source), which is similar to BMS1 and BMS2 w.r.t. to the item distribution, was used for the performance experiments with set containment test algorithms. It was mentioned in Section 6.1.7.

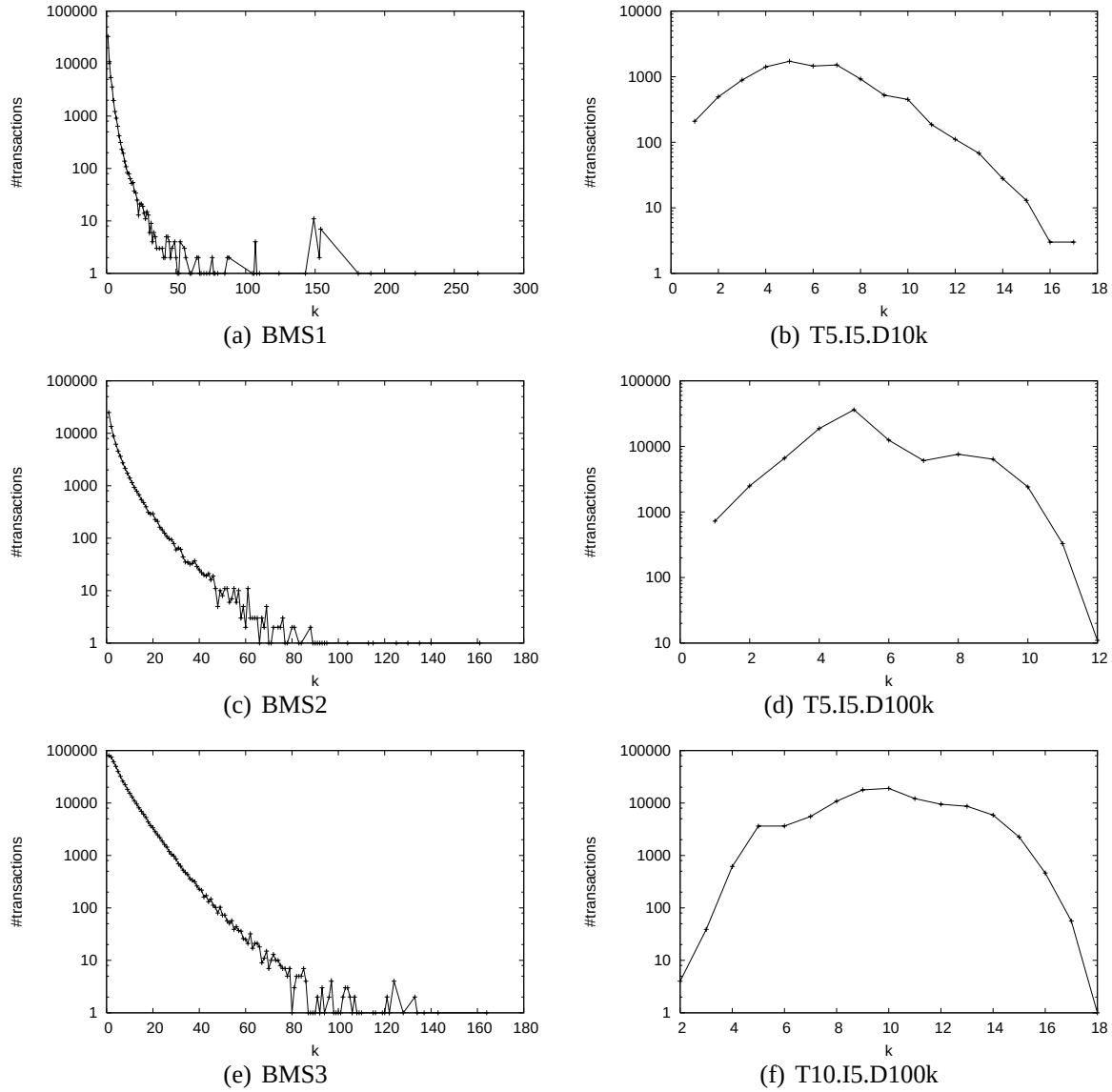
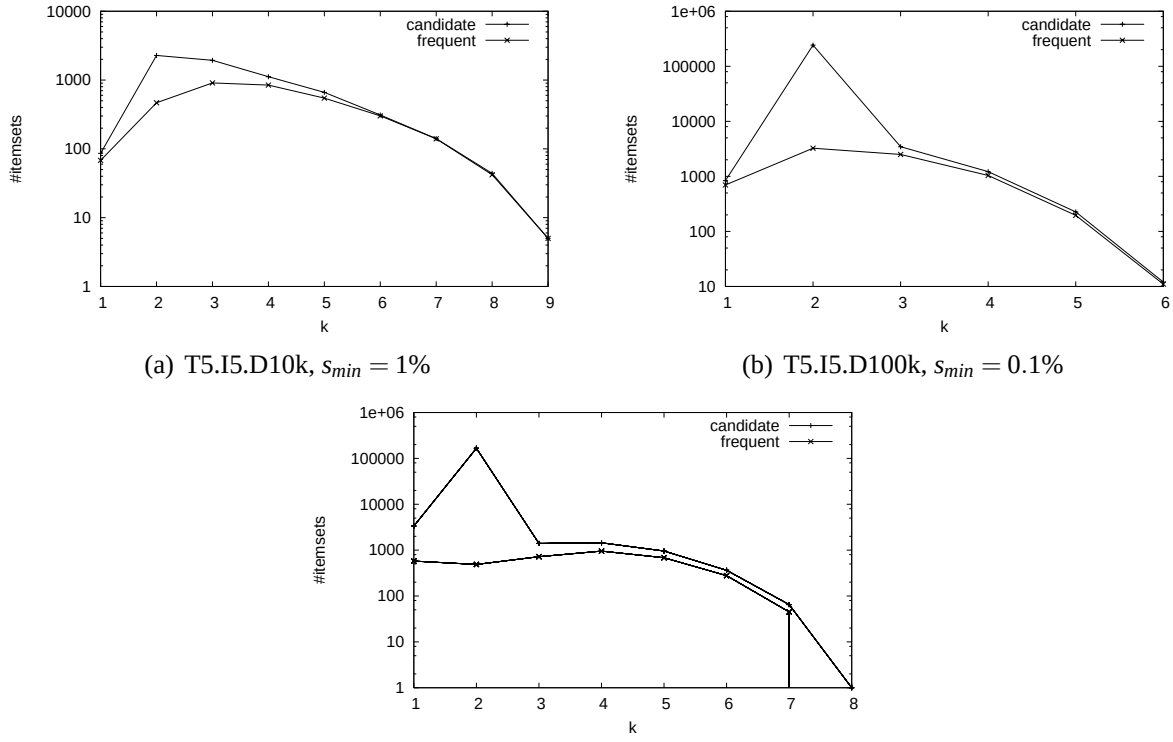


Figure 6.17: Transaction size distribution

Dataset	Trans-actions	Rows	Items	Transaction size		Disk (MB)
				avg.	max.	
BMS1	59 602	149 639	497	2.5	267	1.1
BMS2	77 512	358 278	3 340	4.6	161	2.7
BMS3	515 597	3 367 020	1 657	6.5	164	25.7
T5.I5.D5k	5 000	30 268	86	6.1	18	0.2
T5.I5.D10k	10 000	58 964	86	5.9	17	0.5
T5.I5.D100k	100 000	544 590	839	5.4	12	4.2
T10.I5.D100k	100 000	1 002 226	850	10.0	18	7.6

Table 6.5: Overview of datasets for frequent itemset discovery



(a) T5.I5.D10k, $s_{min} = 1\%$

(b) T5.I5.D100k, $s_{min} = 0.1\%$

(c) BMS2, $s_{min} = 0.2\%$

Figure 6.18: Itemset size distribution

6.4 Frequent Itemset Discovery with a Commercial RDBMS

We compared the performance of the following SQL-based algorithms on two commercial RDBMS:

- K-Way-Join,
- Subquery,

K-Way-Join	c_1^h, s_1, f_1^h	c_2^h, s_2, f_2^h	c_3^h, s_3, f_3^h	...	c_k^h, s_k, f_k^h
Subquery	c_1^h, q_1^h, s_1, f_1^h				
Set-oriented Apriori		t_f^v, c_2^h, s_2, f_2^h	c_3^h, t_3^h, s_3, f_3^h	...	c_k^h, t_k^h, s_k, f_k^h
Quiver	c_1^v, s_1, f_1^v	p, c_2^v, s_2, f_2^v	p, c_3^v, s_3, f_3^v	...	p, c_k^v, s_k, f_k^v
Vertical K-Way-Join					

Table 6.6: Sequence of tables populated by SQL-based algorithms

- Set-oriented Apriori (all discussed in Section 5.6),
- Quiver (introduced in Section 5.7), as well as
- a vertical version of K-Way-Join (proposed in Section 5.7.2).

Remember that Quiver and Vertical K-Way-Join use a vertical table layout for the itemsets, while the other approaches use a horizontal layout.

The first test environment, which we call *MSS*, was Microsoft SQL Server 2000 Standard Edition running on a 4-CPU Intel Pentium-III Xeon PC with 900 MHz, 4 GB main memory, and Windows 2000 Server. It is the same environment that was used for the experiments with the Java query execution engine prototype mentioned in Section 6.1.5. The second one, which we call *DB2*, was IBM DB2 Universal Database V7.2 Enterprise Edition running on a 2-CPU Sun UltraSPARC-III server with 750 MHz, 4 GB main memory, and Sun Solaris Version 8.

For a clearer presentation, we mark tables storing items having a horizontal or vertical layout by the superscript letter *h* and *v*, respectively. These tables are the transactions table *t*, the candidate itemsets table *c*, the frequent itemsets table *f*, as well as some other tables used only for certain algorithms, namely q_1^h, t_f^v, and t_f^v.

An algorithm consists of a sequence of SQL INSERT statements. For each algorithm, the first statement populates the table of candidate 1-itemsets. Table 6.6 shows for each algorithm the tables that are affected by the sequence of INSERT statements. For example, let us consider K-Way-Join. The first INSERT statement populates table c_1^h(*s#*, $i\#_1$). Next, the support table *s*(*s#*, *support*) is filled with the support counts for each *s#* value in c_1^h. Only the itemsets with sufficient support are added to the support table s_1 and subsequently to the frequent 1-itemset table f_1^h. The second iteration begins with an INSERT statement that populates c_2^h, and so on. As with every bottom-up frequent itemset discovery algorithm, the sequence of INSERT statements stops as soon as $C_i = \emptyset$ for some $i \geq 1$ or when a user-defined maximum value of *k* is reached.

The optimization of sequences of SQL statements, and in particular the global optimization of INSERT statement sequences has been a focus of our work [KSRM03]. However, we believe that it goes beyond the work covered in this thesis and therefore we do not discuss it here.

Note that Set-oriented Apriori uses a *vertical* table t_f^v, a subset of *t*, as mentioned in Section 5.6. However, the tables for storing items, c_i^h, f_i^h, and q_1^h, have a *horizontal* layout.

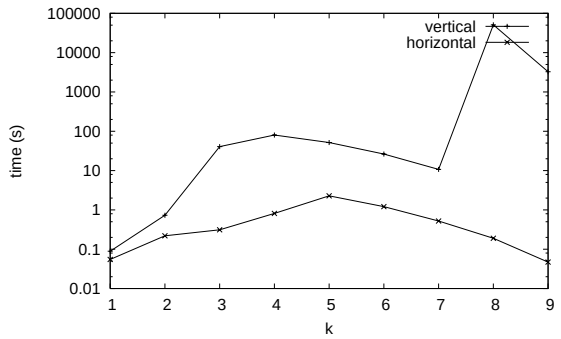
To allow for more query optimization by the RDBMS, several indexes have been created on the intermediate and result tables of the algorithms, as summarized in Table 6.7. Although it is

Layout w.r.t. items	Table	PI: Primary/clustered index SI: Secondary indexes
horizontal	$t_k^{\mathrm{h}}\ (t\#, i\#_1, \ldots, i\#_k)$	PI: $(t\#, i\#_1, \ldots, i\#_k)$ SI: $(t\#), (i\#_1), \ldots, (i\#_k)$
	$q_1^{\mathrm{h}}(t\#, i\#_1)$	PI: $(t\#, i\#_1)$ SI: $(t\#), (i\#_1)$
	$c_k^{\mathrm{h}}\ (s\#, i\#_1, \ldots, i\#_k)$	PI: $(s\#)$ SI: $(i\#_1), \ldots, (i\#_k)$
	$f_k^{\mathrm{h}}(s\#, i\#_1, \ldots, i\#_k)$	PI: $(s\#)$ SI: $(i\#_1), \ldots, (i\#_k)$
vertical	$t^{\mathrm{v}}\ (t\#, i\#)$	PI: $(t\#, i\#)$ SI: $(t\#), (i\#), (i\#, t\#)$
	$t_f^{\mathrm{v}}\ (t\#, i\#)$	PI: $(t\#, i\#)$ SI: $(t\#), (i\#), (i\#, t\#)$
	$c_k^{\mathrm{v}}\ (s\#, pos, i\#)$	PI: $(s\#, i\#)$ SI: $(s\#), (i\#), (pos)$
	$f_k^{\mathrm{v}}(s\#, pos, i\#)$	PI: $(s\#, i\#)$ SI: $(s\#), (i\#), (pos)$
–	$s_k\ (s\#, support)$	PI: $(s\#)$ SI: $(support)$
	$p\ \ (s\#, s\#_1, s\#_2)$	PI: $(s\#)$ SI: $(s\#_1), (s\#_2)$

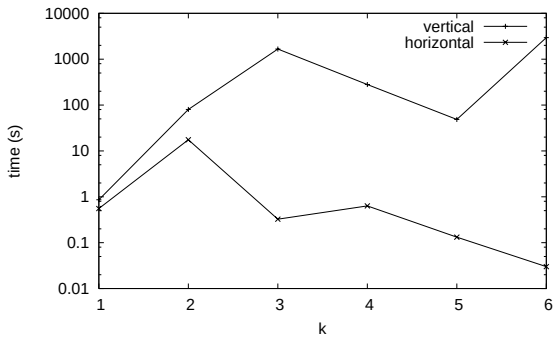
Table 6.7: Overview of indexes created on tables used by SQL-based algorithms

possible to experiment with all attribute combinations for indexing tables of the vertical layout,[6] it requires much effort to do so for large values of k for *horizontal* tables like c_k^{h}. For the Quiver and Vertical K-Way-Join approaches, the tables c_i^{v}, f_i^{v}, and p have only three columns, as can be seen in Table 6.7. Hence, it would be feasible to experiment with all 15 attribute combinations as indexes. On the other hand, horizontal tables in the ith iteration have $i+1$ columns and thus many more attribute combinations for indexes are possible for high values of i, as discussed in Section 5.5. Even for the tables with a vertical layout, simply using all possible indexes might not always be a good idea since index maintenance requires a considerable time and storage space overhead. Hence, our index choice is one out of many possible, but we have made a careful analysis on what indexes to offer to the optimizer. Modern RDBMSs offer an "index advisor" tool that suggest which indexes to create for a given query workload. We have not used such a tool. Note that the performance results reported in [STA98] on algorithms based on SQL-92, mentioned in Section 5.6, are very detailed but they do not cover the entire number of indexes neither. Hence, we must leave as future work a comprehensive study on selecting the most effective indexes for the algorithms.

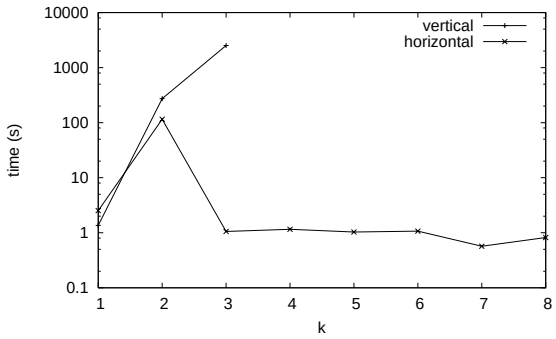
[6] see footnote 3 on page 96



(a) MSS, T5.I5.D10k, $s_{min} = 1\%$



(b) MSS, T5.I5.D100k, $s_{min} = 0.1\%$



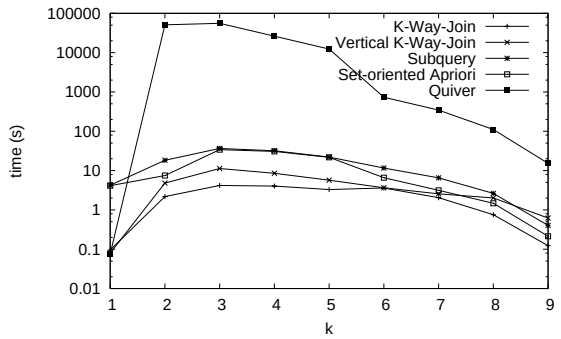
(c) DB2, BMS2, $s_{min} = 0.2\%$

Figure 6.19: Execution times of candidate generation phases

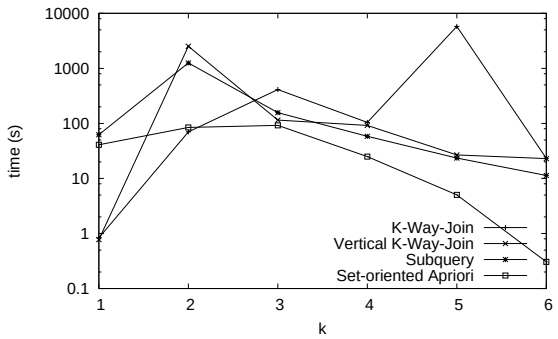
After each INSERT statement on a table, we let the RDBMS recompute the statistics of that table as well as all indexes created on it to enable the optimizer to produce a good query execution plan for each INSERT statement that follows. Of course, the statistics creation consumes some amount of the total execution time for the statement batch but we observed that in all reasonable cases this overhead was negligible compared to the total execution time.

6.4.1 Candidate Generation

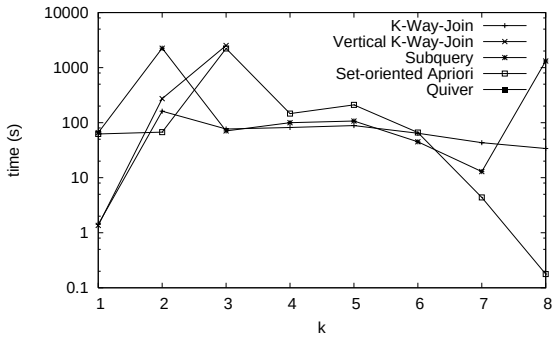
All horizontal approaches use the same SQL INSERT statement for the generation of candidate itemsets. The same holds for all vertical approaches. Figure 6.19 shows the execution times of the candidate generation phase in each iteration of the algorithms. One can see that the candidate generation phase based on a vertical table layout (Quiver and Vertical K-Way-Join) is slower than for the horizontal approach (K-Way-Join, Subquery, Set-oriented Apriori). For the iterations 8–9 using dataset T5.I5.D10k (Figure 6.19(a)) and for the iterations 3–6 using dataset T5.I5.D100k (Figure 6.19(b)), the response time differed by more than two orders of magnitude. This difference is mainly caused by an expensive processing of the numerous correlated (NOT) EXISTS sub-queries that compute the prefix-pair table *p*, shown in Algorithm 9 on page 105. This query has a growing complexity for increasing itemset cardinalities.



(a) MSS, T5.I5.D10k, $s_{min} = 1\%$



(b) MSS, T5.I5.D100k, $s_{min} = 0.1\%$



(c) DB2, BMS2, $s_{min} = 0.2\%$

Figure 6.20: Execution times of support counting phases

6.4.2 Support Counting

Our results for the support counting phase are illustrated in Figure 6.20. They reveal that for dataset T5.I5.D10k, Figure 6.20(a), K-Way-Join is superior to all approaches in all but the last iteration of the support counting phase. Except for Quiver, the algorithms had a performance that stayed within a corridor of at most an order of magnitude from each other. We had to stop the execution for the experiments with Quiver for datasets T5.I5.D100k and BMS2, therefore the graphs are incomplete (BMS2) or are missing (T5.I5.D100k) in this figure. The reason for this is that during the computation of the prefix-pairs table p, the apriori tests are extremely expensive to execute and caused a lot of I/O for writing temporary data.

The experiment with dataset T5.I5.D100k, Figure 6.20(b), confirms the claim in [TC99] that Set-oriented Apriori performs better than Subquery for late passes of the algorithm, i.e., for high values of k.

6.4.3 Query Execution Plans

In this section, we discuss the query execution plans for several SQL statements produced by a commercial RDBMS for the SQL-based frequent itemset discovery algorithms K-Way-Join and Quiver. In addition, we show how the Quiver query for support counting can be realized using a

Operator	Name	Description
Sort	Sort	Sorts the input rows in ascending or descending order of the column values.
RowCount	Row count	Returns the number of rows in the input.
${}_A\mathit{IScan}^S$	Index scan	Retrieves input rows using an index on column list A. The specification S describes the index access more closely, e.g., "clustered."
${}_A\mathit{ISeek}^S_C$	Index lookup	Retrieves those input rows using an index on column list L that fulfill condition C. The specification S describes the index access more closely.
Top_1	First row	Retrieves the first row and skips the rest of the input.
${}_A\times$	Correlated Cartesian product	A is a list of assignments, each assigning an expression based on the columns of the left input to a variable. These variables can be used in the conditions of an operator used in the right expression. Example: $r_{1a+b\to x} \times \sigma_{c=x}(r_2)$ for some relations r_1 and r_2 with the schemas $R_1(a,b)$ and $R_2(c)$, respectively. For each tuple in r_1, the expression builds the Cartesian product only with those tuples in r_2 that fulfill the condition $r_1.a + r_1.b = r_2.c$. This is similar to a theta-join but it allows to push the join condition deeper into the left expression.
$\bowtie^I_C$	Theta-join	I is the implementation and C the condition.
$\div^*_I$	Set containment division	I is the implementation.
${}_L\overline{\ltimes}^I_C$	Left anti-semi-join	I is the implementation and C the condition. The letter "L" denotes *left* anti-semi-join. For each row l of the left input, the operator evaluates the right input. If no row is produced in the right input, it outputs l. The values of l act as "outer references" for the right, "inner" input.
${}_G\gamma^I_A$	Grouping	I is the implementation, G is a list of grouping columns and A is a list of aggregations.

Table 6.8: Overview of the physical operators of the relational algebra used in this thesis

set containment division operator. For an overview of the operators used in the plan, we refer to Table 6.8.

We compare Quiver to K-Way-Join because of their structural similarity. We are aware of the fact that K-Way-Join is not the best algorithm based on SQL-92 overall. In the following, we discuss the query execution strategy chosen by the optimizer in the MSS environment for the INSERT INTO s_4 statement. The statement is similar to that in Algorithm 10 on page 107, which shows the statement for $k = 3$. The transaction table is based on the dataset T5.I5.D5k and the candidate table c^{h}_4 was produced for a minimum support value of 2% (100 transactions). Figure 6.21 shows the query execution plan for this SQL statement. The plan contains four hash-joins to realize the Apriori condition: it joins the candidate table c^{h}_4 four times with the transaction table t^{v}.

Figure 6.22 shows the query execution plan for the INSERT statement into s_4 for the Quiver algorithm using NOT EXISTS, illustrated in Algorithm 13 on page 109. The plan employs anti-

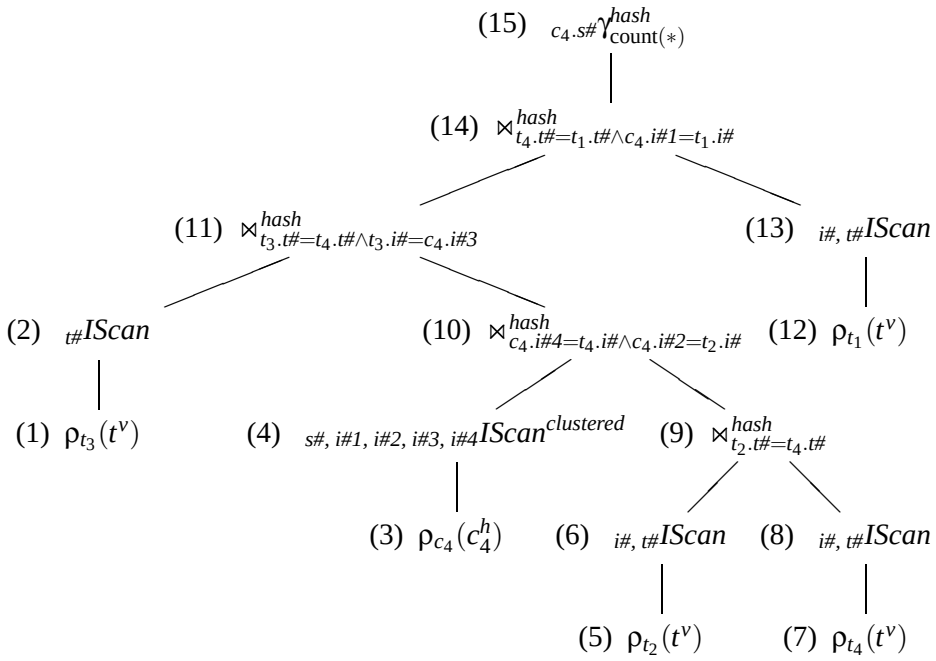


Figure 6.21: Query execution plan of horizontal K-Way-Join with hash-joins

semi-joins. A left anti-semi-join returns all rows of the left input that have no matching rows in the right input. The top left anti-semi-join, operator 20, checks for each combination of values $(c_1.s\#, t_1.t\#)$ on the left if at least one row can be retrieved from the right input. If not, this value pair qualifies for the subsequent grouping and counting, otherwise the left row is skipped. A similar processing is done for the left anti-semi-join, operator 18, with the outer reference $c_2.i\#$. An interesting point to note is that the index scan, operator 10, of t_2 on *i#, t#* is uncorrelated, i.e., no reference to the left input is used in the right sub-plan (operators 9–12). Every access to this table is used to check if the transaction table is empty or not. For our problem of frequent itemset discovery this table is nonempty by default. Hence, the row count always yields the value one.

6.5 Support Counting with a Java Query Execution Engine

To contrast the complex query execution plans that have been derived for *real* SQL queries using a commercial RDBMS, we illustrate a plan for the version of the Quiver query that employs the hypothetical GREAT DIVIDE syntax for the set containment division operator in Algorithm 14 on page 109. Figure 6.23 illustrates the plan, which is similar to the relational algebra representation of the support counting phase in Equation 5.1 on page 92.

We made several experiments to compare the performance of the three types of query execution plans using our Java query execution engine. We used subsets of the transaction datasets and subsets of the candidate 4-itemsets to derive frequent 4-itemsets for certain minimum support values. Figure 6.24 shows the results of our experiments. One can observe that the execution

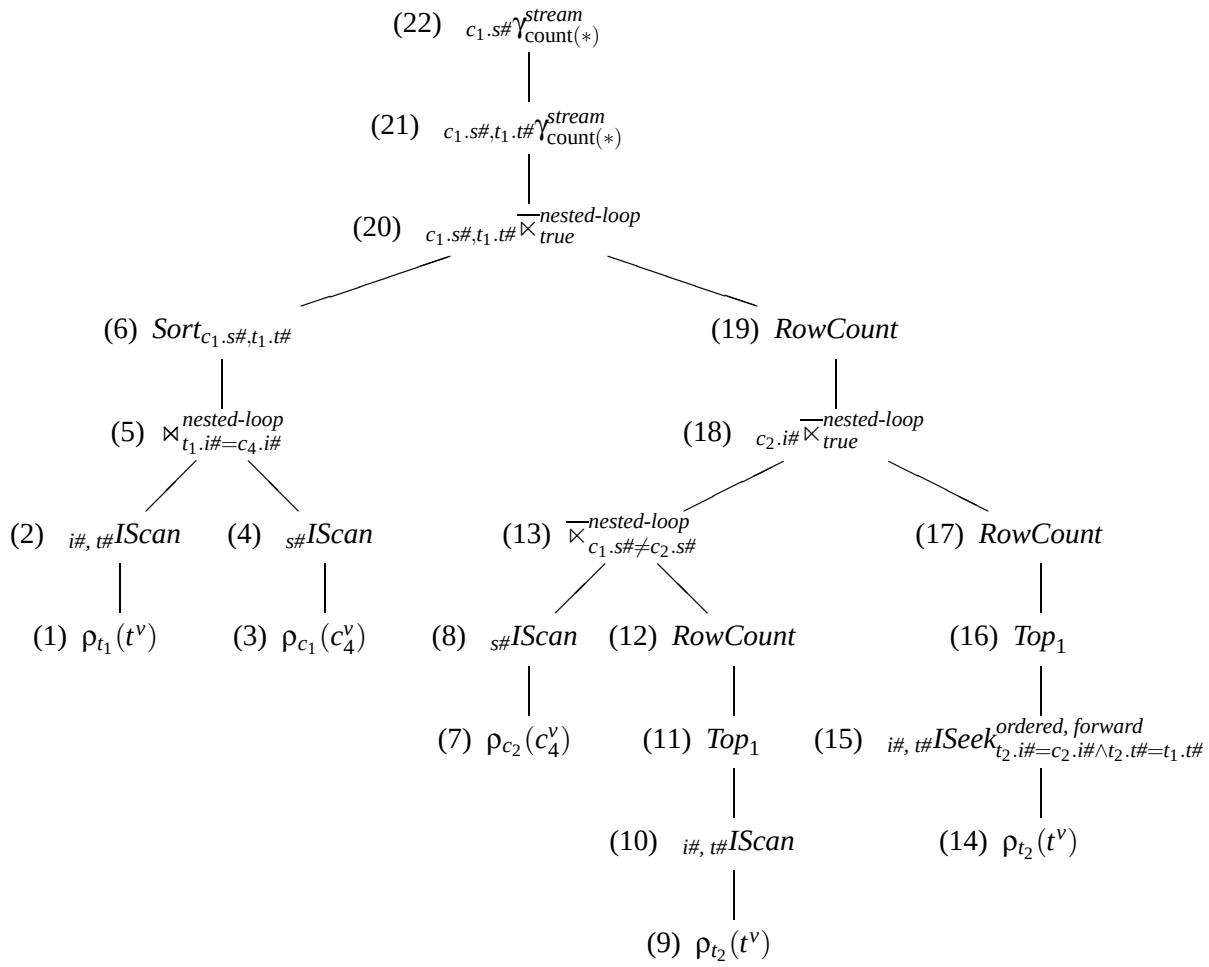


Figure 6.22: Query execution plan of Quiver with nested-loop join and anti-semi-joins

plan using set containment division outperformed the other approaches for small numbers of candidate sets. The plan chosen by the commercial RDBMS based on anti-semi-join was always a bad solution for the given datasets. K-Way-Join outperforms the other approaches for larger numbers of candidates. The query execution plan execution was not parallelized in any way.

6.6 Summary

In the first part of this chapter, we discussed several implementation details of the a query execution engine that was implemented in Java with the help of the XXL class library. Several experiments have been conducted with this prototype to assess the performance characteristics of the algorithms for set containment division (sort-based, hash-based and based on a subset index) and set containment join (adaptive pick-and-sweep join). The experiments demonstrated that the universal quantification problem can be solved efficiently with the help of these algorithms.

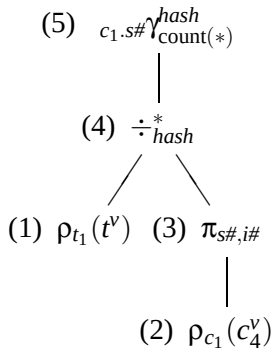


Figure 6.23: Query execution plan of Quiver with hash-based set containment division operator

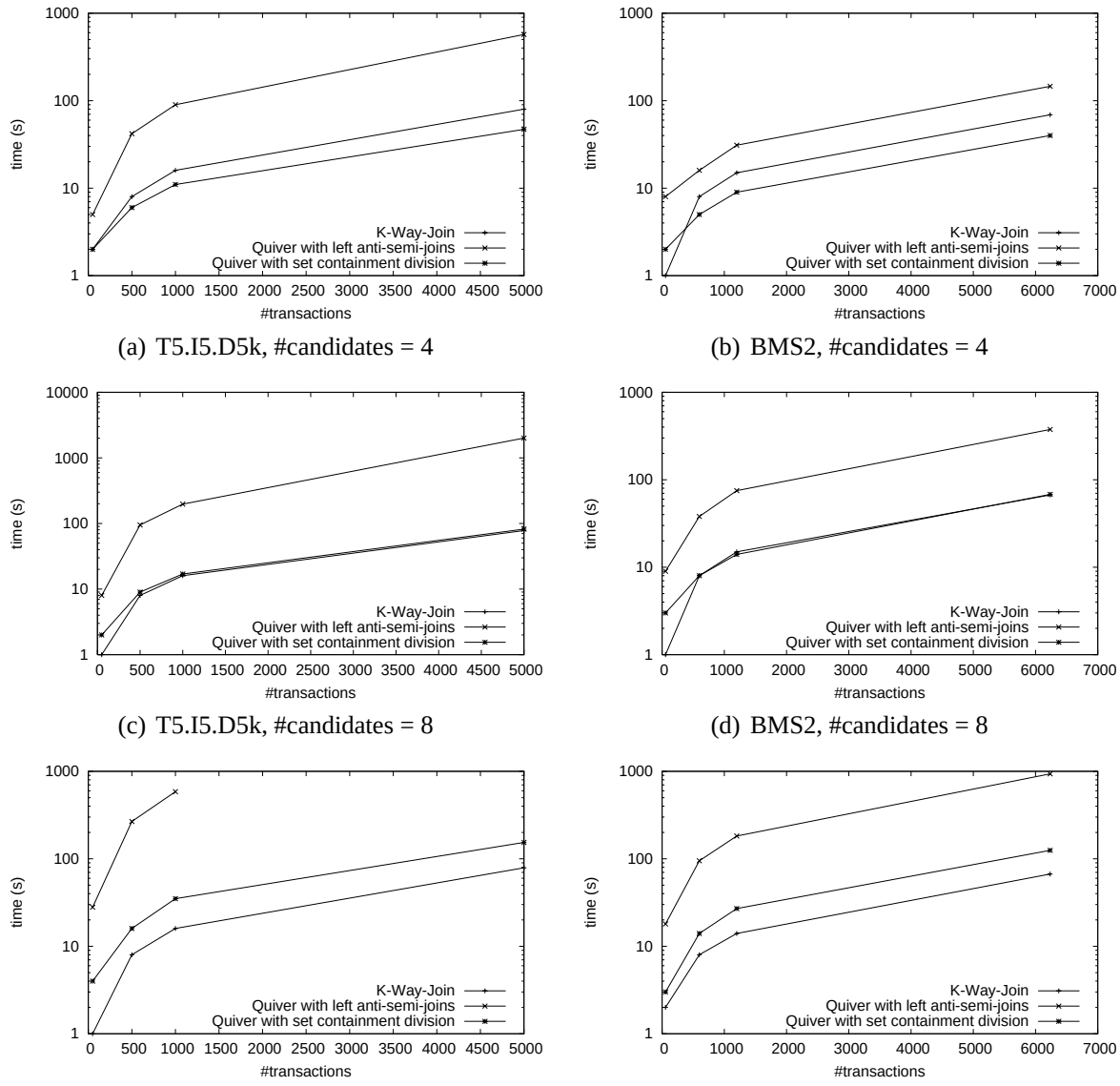
It is impossible to make a fair comparison of the performance between the algorithms because each one relies on several parameters that influence their behavior. For example, hash-based set containment division is affected by the choice of the hash table sizes and the definition of the hash function. Much more tuning knobs are offered by adaptive pick-and-sweep join. However, let us briefly summarize our major findings from the experiments with the set containment test operators:

- The effort for nesting unnested data is higher than for unnesting nested data.
- For unsorted data, hash-based set containment division had lower execution times than merge-sort set containment division plus external sorting.
- For a large divisor and a small dividend, it is beneficial to create a subset index on the dividend and otherwise on the divisor.
- Compression saves up to 90% of memory even for small real-life datasets as opposed to an uncompressed subset index.
- Set containment division using a compressed subset index can be up to an order of magnitude slower than with an uncompressed one.

The second part of the chapter was dedicated to the evaluation of SQL-based frequent itemset discovery algorithms. From the experiments with these approaches, we learned that

- the Quiver algorithm realized by NOT EXISTS predicates in SQL has an execution time that can be several orders of magnitude higher than for other approaches based on SQL-92.
- Quiver, when implemented using set containment division, can be as efficient as other approaches based on SQL-92.

We conclude that universal quantification is not well supported by modern commercial database systems. An integration of set containment join or set containment division operators can improve the performance of the support counting phase.



(a) T5.I5.D5k, #candidates = 4

(b) BMS2, #candidates = 4

(c) T5.I5.D5k, #candidates = 8

(d) BMS2, #candidates = 8

(e) T5.I5.D5k, #candidates = 16

(f) BMS2, #candidates = 16

Figure 6.24: Execution times of query execution plans running in a Java query execution engine prototype

"Finally, in conclusion, let me say just this."
P. Sellers (1925 – 1980)

7

Conclusion and Future Work

Set containment tests are a practical problem in query processing. Many solutions to it have been devised to solve it efficiently. We reviewed some of these approaches in greater detail, in particular those that require that the sets are stored in an unnested table layout—the relational division operators. We presented the myriad of definitions for this derived operator of the relational algebra in a coherent way. We attempted to classify physical division operators according to whether and how the input tables are grouped on some of their columns and discussed their complexities. This classification is valuable for an optimizer of an RDBMS when more than one physical division operator is available.

We suggested a new operator called *set containment division* that generalizes the division operator. The new operator allows the divisor table to store more than one set. Hence we generalized the cardinality of the set containment test relationship between dividend sets and divisor sets from many-to-one (for division) to many-to-many. We showed that set containment division is equivalent to two other operators that have been proposed in the past—*great divide* and *generalized division*—but that have neither been referenced widely nor investigated in any depth.

In order to improve the execution time of queries whose execution plans make use of a physical division operator, an optimizer needs to have rewrite rules for the operator. Several algebraic laws involving division were introduced in this thesis for the first time.

We analyzed set containment join algorithms, which join tables on a set-valued column. They typically rely on two techniques, *signatures* to reduce the cost of a comparison between two sets

and *partitioning* to reduce the number of set comparisons. We discussed the latter technique for set containment division algorithms as well. Thus, it is possible to parallelize the processing of queries that involve set containment division operators.

A new main-memory data structure called *subset index*, which enables an efficient set containment division if at least one of the dividend or divisor is small enough to fit into memory, was proposed. It is possible to compress the representation of sets in this index thus saving an enormous amount of storage. However, the performance experiments have revealed that the processing is significantly slower than without compression.

As a promising application area for set containment division and set containment join algorithms, we studied SQL-based frequent itemset discovery algorithms. The set containment problem is central to this data mining problem, where we ask for the number of transaction sets that contain a given itemset. We devised a new algorithm called *Quiver* that represents itemsets in a vertical table layout, like the transactions, in tuple relational calculus, relational algebra, and SQL. The SQL formulation was subject to several performance experiments with two commercial RDBMSs. Not surprisingly, the experiments revealed that the database systems do not perform well when confronted with queries where the universal quantifications are expressed by negated existential quantifications. By rewriting such Quiver SQL queries into queries that use a (hypothetical) set containment division operator, an RDBMS would be able to find a more efficient execution strategy, provided that the RDBMS has implemented a set containment division or set containment join operator.

In addition to the experiments with frequent itemset discovery queries, we analyzed the performance of several set containment division and set containment join algorithms. The algorithms were implemented in Java and made use of a Java class library for building query processors, called XXL.

We see directions of future research in the following areas. XQuery [W3C03], the de-facto standard query language for XML data, is subject to enormous research efforts, now and in the near future. An interesting question is how to process universal quantifications in XQuery queries efficiently. As mentionend in Section 2.5.1, XQuery provides an *every expression* to realize universal quantification. Furthermore, the language allows to specify whether the query output should preserve the ordering of tuples in the input documents. Therefore, algorithms that realize universal quantification have to be combined with order-preserving algebra operators [MHM03].

Another possible topic of future work is in the area of data streams or continuous queries. Given a large amount of queries that are registered in a query processor and several data sources that produce new rows more or less continuously with data generation rates that are diverse. One of the challenges is how to minimize the amount of processing needed to execute the potentially large amount of queries that have to be processed when a matching input row is produced. Suppose that many of the queries involve set containment division operators that have the same divisor table, i.e., divisor data source, as input but that they work on different dividend tables. Furthermore, assume that the divisor data source produces a new divisor group. How can we compute new quotient rows for each query at the lowest total query processing cost? We could process all dividend tables affected by the set containment division operator and produce new quotient results. However, if many of the dividend tables are similar to each other, we might save some work by analyzing the similarities first and then assigning an intelligent processing

strategy that avoids the brute-force method of rescanning all dividend tables.

A Proofs

A.1 Lemmas

LEMMA 1: *Let X, Y, and Z be sets. Then, $X - Y = X - Z \Leftrightarrow X \cap Y = X \cap Z$.*

PROOF (LEMMA 1): We prove the lemma by deriving implications for both directions of the equivalence. First, we show that $X - Y = X - Z \Rightarrow X \cap Y = X \cap Z$:

$$
\begin{aligned}
t \in (X \cap Y) &\Leftrightarrow t \in X \wedge t \in Y \\
&\Leftrightarrow (t \in X \wedge t \notin X) \vee (t \in X \wedge t \in Y) \\
&\Leftrightarrow t \in X \wedge (t \notin X \vee t \in Y) \\
&\Leftrightarrow t \in X \wedge \neg (t \in X \wedge t \notin Y) \\
&\Leftrightarrow t \in X \wedge \neg (t \in (X - Y)) \\
&\Leftrightarrow t \in X \wedge \neg (t \in (X - Z)) \qquad \text{due to our assumption} \\
&\Leftrightarrow t \in X \wedge \neg (t \in X \wedge t \notin Z) \\
&\Leftrightarrow t \in X \wedge (t \notin X \vee t \in Z) \\
&\Leftrightarrow (t \in X \wedge t \notin X) \vee (t \in X \wedge t \in Z) \\
&\Leftrightarrow t \in X \wedge t \in Z \\
&\Leftrightarrow t \in (X \cap Z)
\end{aligned}
$$

Next, we show that $X \cap Y = X \cap Z \Rightarrow X - Y = X - Z$:

$$
\begin{aligned}
t \in (X-Y) &\Leftrightarrow t \in X \wedge t \notin Y \\
&\Leftrightarrow (t \in X \wedge t \notin Y) \vee (t \in X \wedge t \notin X) \\
&\Leftrightarrow (t \in X \vee (t \in X \wedge t \in X)) \wedge (t \notin Y \vee (t \in X \wedge t \notin X)) \\
&\Leftrightarrow t \in X \wedge (t \notin Y \vee (t \in X \wedge t \notin X)) \\
&\Leftrightarrow t \in X \wedge (t \in X \vee t \notin Y) \wedge (t \notin X \vee t \notin Y) \\
&\Leftrightarrow t \in X \wedge (t \notin X \vee t \notin Y) \\
&\Leftrightarrow t \in X \wedge \neg(t \in X \wedge t \in Y) \\
&\Leftrightarrow t \in X \wedge \neg(t \in (X \cap Y)) \\
&\Leftrightarrow t \in X \wedge \neg(t \in (X \cap Z)) \qquad \text{due to our assumption} \\
&\Leftrightarrow t \in X \wedge \neg(t \in X \wedge t \in Z) \\
&\Leftrightarrow t \in X \wedge (t \notin X \vee t \notin Z) \\
&\Leftrightarrow (t \in X \wedge t \notin X) \vee (t \in X \wedge t \notin Z) \\
&\Leftrightarrow t \in X \wedge t \notin Z \\
&\Leftrightarrow t \in (X - Z)
\end{aligned}
$$

□

LEMMA 2: *Let X_1, X_2, Y_1 and Y_2 be sets. If $X_1 \cap X_2 = \emptyset$ and $X_i \supseteq Y_i$ for $i \in \{1,2\}$ then* $(X_1 - Y_1) \cup (X_2 - Y_2) = (X_1 \cup X_2) - (Y_1 \cup Y_2)$.

PROOF (LEMMA 2):

$$
\begin{aligned}
t \in (X_1 - Y_1) \cup (X_2 - Y_2) &\Leftrightarrow (t \in X_1 \wedge t \notin Y_1) \vee (t \in X_2 \wedge t \notin Y_2) \\
&\Leftrightarrow (t \in X_1 \vee t \in X_2) \wedge (t \in X_1 \vee t \notin Y_2) \wedge \\
&\quad (t \notin Y_1 \vee t \in X_2) \wedge (t \notin Y_1 \vee t \notin Y_2) \\
&\Leftrightarrow (t \in X_1 \vee t \in X_2) \wedge \neg(t \notin X_1 \wedge t \in Y_2) \wedge \\
&\quad \neg(t \in Y_1 \wedge t \notin X_2) \wedge \neg(t \in Y_1 \wedge t \in Y_2) \\
&\Leftrightarrow t \in (X_1 \cup X_2) \wedge t \notin (Y_2 - X_1) \wedge t \notin (Y_1 - X_2) \wedge t \notin (Y_1 \cap Y_2) \\
&\Leftrightarrow t \in (X_1 \cup X_2) \wedge \neg(t \in (Y_2 - X_1) \vee t \in (Y_1 - X_2) \vee t \in (Y_1 \cap Y_2)) \\
&\Leftrightarrow t \in (X_1 \cup X_2) \wedge t \notin ((Y_2 - X_1) \cup (Y_1 - X_2) \cup (Y_1 \cap Y_2))
\end{aligned}
$$

We find that

$(Y_2 - X_1) = Y_2$ since $Y_2 \subseteq X_2$ and $X_2 \cap X_1 = \emptyset$ and that
$(Y_1 - X_2) = Y_1$ since $Y_1 \subseteq X_1$ and $X_1 \cap X_2 = \emptyset$.

Hence, we have

$$\begin{aligned}(Y_2 - X_1) \cup (Y_1 - X_2) \cup (Y_1 \cap Y_2) &= Y_2 \cup Y_1 \cup (Y_1 \cap Y_2) \\ &= Y_2 \cup Y_1.\end{aligned}$$

Thus, we finally find that

$$\begin{aligned}t \in (X_1 \cup X_2) \wedge t \notin ((Y_2 - X_1) \cup (Y_1 - X_2) \cup (Y_1 \cap Y_2)) &\Leftrightarrow t \in (X_1 \cup X_2) \wedge t \notin (Y_1 \cup Y_2) \\ &\Leftrightarrow t \in (X_1 \cup X_2) - (Y_1 \cup Y_2).\end{aligned}$$

□

LEMMA 3: *Set containment division ($\div_1^*$) and great divide ($\div_3^*$) are equivalent operators.*

PROOF (LEMMA 3): In the following, we will show the equivalence of the relational algebra expressions of set containment division in Definition 8 and of great divide used in Definition 10. Let r_1 be a dividend relation and r_2 a divisor relation with schemas $R_1(A \cup B)$ and $R_2(B \cup C)$, respectively, as defined in Section 2.3.2. Let $\{C_1, \ldots, C_k\}$ be the set of (distinct) tuples in $\pi_C(r_2)$. If the divisor is non-empty then $k \geq 1$. We use the following algebraic laws as propositions:

(P1) $\pi_A(r_1 \cup r_2) = \pi_A(\pi_A(r_1) \cup \pi_A(r_2))$ for any relations r_1 and r_2 with the same schema $R(A \cup X)$, where A is a nonempty set of attributes and attribute set X may be empty or not.

Let us start with expression e, the definition of set containment division:

$$\begin{aligned}e &= r_1 \div_1^* r_2 \\ &= \bigcup_{t \in \pi_C(r_2)} (r_1 \div \pi_B(\sigma_{C=t}(r_2))) \times (t)\end{aligned}$$

We replace the division operator by Definition 3:

$$\begin{aligned}e &= \bigcup_{t \in \pi_C(r_2)} (\pi_A(r_1) - \pi_A((\pi_A(r_1) \times \pi_B(\sigma_{C=t}(r_2))) - r_1)) \times (t) \\ &= \left(\bigcup_{t \in \pi_C(r_2)} \pi_A(r_1) \times (t) \right) - \left(\bigcup_{t \in \pi_C(r_2)} (\pi_A((\pi_A(r_1) \times \pi_B(\sigma_{C=t}(r_2))) - r_1)) \times (t) \right) \\ &= \underbrace{(\pi_A(r_1) \times \pi_C(r_2))}_{e_0} - \left(\bigcup_{t \in \pi_C(r_2)} (\pi_A((\pi_A(r_1) \times \pi_B(\sigma_{C=t}(r_2))) - r_1)) \times (t) \right) \\ &= e_0 - \left(\bigcup_{1 \leq i \leq k} (\pi_A((\pi_A(r_1) \times \pi_B(\sigma_{C=C_i}(r_2))) - r_1)) \times (C_i) \right) \\ &= e_0 - \pi_{A \cup C} \left(\bigcup_{1 \leq i \leq k} \pi_{A \cup C}(((\pi_A(r_1) \times \pi_B(\sigma_{C=C_i}(r_2))) \times (C_i)) - (r_1 \times (C_i))) \right)\end{aligned}$$

$$= e_0 - \pi_{A\cup C}\left(\bigcup_{1\le i\le k} \pi_{A\cup C}\left((\pi_A(r_1)\times\sigma_{C=C_i}(r_2)) - (r_1\times(C_i))\right)\right)$$

$$= e_0 - \pi_{A\cup C}\left(\bigcup_{1\le i\le k} \pi_{A\cup C}\left(\underbrace{\pi_{A\cup r_2.B\cup C}(\pi_A(r_1)\times\sigma_{C=C_i}(r_2))}_{e_i^1} - \underbrace{\pi_{A\cup r1.B\cup C}(r_1\times(C_i))}_{e_i^2}\right)\right)$$

Next, let us take a look at expression $\tilde{e}$ representing Todd's great divide:

$$\begin{aligned}\tilde{e} &= r_1 \div_3^* r_2\\ &= \underbrace{(\pi_A(r_1)\times\pi_C(r_2))}_{\tilde{e}_0} - \pi_{A\cup C}((\pi_A(r_1)\times r_2) - (r_1 \bowtie r_2))\\ &= \tilde{e}_0 - \pi_{A\cup C}\left(\left(\pi_A(r_1)\times\bigcup_{1\le i\le k}\sigma_{C=C_i}(r_2)\right) - \pi_{A\cup r_2.B\cup C}\left(r_1 \bowtie_{r_1.B=r_2.B}\bigcup_{1\le i\le k}\sigma_{C=C_i}(r_2)\right)\right)\\ &= \tilde{e}_0 - \pi_{A\cup C}\left(\left(\bigcup_{1\le i\le k}\pi_A(r_1)\times\sigma_{C=C_i}(r_2)\right) - \pi_{A\cup r_2.B\cup C}\left(\bigcup_{1\le i\le k} r_1 \bowtie_{r_1.B=r_2.B}\sigma_{C=C_i}(r_2)\right)\right)\end{aligned}$$

Using Lemma 2 we get

$$\tilde{e} = \tilde{e}_0 - \pi_{A\cup C}\left(\bigcup_{1\le i\le k}\pi_{A\cup r_2.B\cup C}(\pi_A(r_1)\times\sigma_{C=C_i}(r_2)) - \pi_{A\cup r_2.B\cup C}(r_1 \bowtie_{r_1.B=r_2.B}\sigma_{C=C_i}(r_2))\right)$$

Using proposition P1 we get

$$\tilde{e} = \tilde{e}_0 - \pi_{A\cup C}\left(\bigcup_{1\le i\le k}\pi_{A\cup C}\left(\underbrace{\pi_{A\cup r_2.B\cup C}(\pi_A(r_1)\times\sigma_{C=C_i}(r_2))}_{\tilde{e}_i^1} - \underbrace{\pi_{A\cup r_2.B\cup C}(r_1 \bowtie_{r_1.B=r_2.B}\sigma_{C=C_i}(r_2))}_{\tilde{e}_i^2}\right)\right)$$

We see that expressions e and $\tilde{e}$ differ only in the subexpressions e_i^2 and $\tilde{e}_i^2$, respectively. We are now going to show that $e_i^1 - e_i^2 = \tilde{e}_i^1 - \tilde{e}_i^2$. Then we know that $e = \tilde{e}$, i.e., set containment division and great divide are equivalent.

Instead of showing that $e_i^1 - e_i^2 = \tilde{e}_i^1 - \tilde{e}_i^2$, we prove the equivalent statement $e_i^1 \cap e_i^2 = \tilde{e}_i^1 \cap \tilde{e}_i^2$. These statements are equivalent because of Lemma 1. Based on this lemma, we can derive the following expressions:

$$\tilde{e}_i^1 \cap \tilde{e}_i^2 = \pi_{A\cup r_2.B\cup C}(\pi_A(r_1)\times\sigma_{C=C_i}(r_2)) \cap \pi_{A\cup r_2.B\cup C}(r_1 \bowtie_{r_1.B=r_2.B}\sigma_{C=C_i}(r_2))$$

$$
\begin{aligned}
&= \pi_{A \cup r_2.B \cup C}\left(\pi_A\left(r_1\right) \times \sigma_{C=C_i}\left(r_2\right)\right) \cap \pi_{A \cup r_2.B \cup C}\left(\sigma_{r_1.B=r_2.B}\left(r_1 \times \sigma_{C=C_i}\left(r_2\right)\right)\right) \\
&= \pi_{A \cup r_2.B \cup C}\left(r_1 \times \sigma_{C=C_i}\left(r_2\right)\right) \cap \pi_{A \cup r_2.B \cup C}\left(\sigma_{r_1.B=r_2.B}\left(r_1 \times \sigma_{C=C_i}\left(r_2\right)\right)\right) \\
&= \pi_{A \cup r_2.B \cup C}\left(\sigma_{r_1.B=r_2.B}\left(r_1 \times \sigma_{C=C_i}\left(r_2\right)\right)\right) \\
&= \pi_{A \cup r_2.B \cup C}\left(r_1 \bowtie_{r_1.B=r_2.B} \sigma_{C=C_i}\left(r_2\right)\right) \\
&= \pi_{A \cup r_2.B \cup C}\left(\sigma_{r_1.B=r_2.B}\left(r_1 \times \sigma_{C=C_i}\left(r_2\right)\right)\right) \\
&= \pi_{A \cup r_2.B \cup C}\left(\left(r_1 \times \sigma_{C=C_i}\left(r_2\right)\right) \cap \pi_{A \cup r_1.B \cup C}\left(r_1 \times \sigma_{C=C_i}\left(r_2\right)\right)\right) \\
&= \pi_{A \cup r_2.B \cup C}\left(\pi_A\left(r_1\right) \times \sigma_{C=C_i}\left(r_2\right)\right) \cap \pi_{A \cup r_1.B \cup C}\left(r_1 \times \sigma_{C=C_i}\left(r_2\right)\right) \\
&= \pi_{A \cup r_2.B \cup C}\left(\pi_A\left(r_1\right) \times \sigma_{C=C_i}\left(r_2\right)\right) \cap \pi_{A \cup r_1.B \cup C}\left(r_1 \times \left(C_i\right)\right) \\
&= e_i^1 \cap e_i^2
\end{aligned}
$$

□

LEMMA 4: *Great divide ($\div_3^*$) and generalized division ($\div_2^*$) are equivalent operators.*

PROOF (LEMMA 4): In the following, we will show the equivalence of the relational algebra expressions of great divide used in Definition 10 and of generalized division in Definition 9. Let r_1 be a dividend relation and r_2 a divisor relation with schemas $R_1(A \cup B)$ and $R_2(B \cup C)$, respectively, as defined in Section 2.3.2. Let us review expression e, the definition of great divide:

$$
\begin{aligned}
e &= r_1 \div_3^* r_2 \\
&= \left(\pi_A\left(r_1\right) \times \pi_C\left(r_2\right)\right) - \pi_{A \cup C}\left(\underbrace{\left(\pi_A\left(r_1\right) \times r_2\right)}_{e_1} - \underbrace{\left(r_1 \bowtie r_2\right)}_{e_2}\right)
\end{aligned}
$$

Now, we compare expression e to expression $\tilde{e}$, the definition of generalized division:

$$
\begin{aligned}
\tilde{e} &= r_1 \div_2^* r_2 \\
&= \left(\pi_A\left(r_1\right) \times \pi_C\left(r_2\right)\right) - \pi_{A \cup C}\left(\underbrace{\left(\pi_A\left(r_1\right) \times r_2\right)}_{\tilde{e}_1} - \underbrace{\left(r_1 \times \pi_C\left(r_2\right)\right)}_{\tilde{e}_2}\right)
\end{aligned}
$$

We find that e and $\tilde{e}$ differ only in the expression e_2 and $\tilde{e}_2$, respectively. If we can show that $e_1 - e_2 = \tilde{e}_1 - \tilde{e}_2$, we have proved that $e = \tilde{e}_1$. Because of Lemma 1, it suffices to show that $e_1 \cap e_2 = \tilde{e}_1 \cap \tilde{e}_2$:

$$
\begin{aligned}
e_1 \cap e_2 &= \left(\pi_A\left(r_1\right) \times r_2\right) \cap \left(r_1 \bowtie r_2\right) \\
&= \left(\pi_A\left(r_1\right) \times r_2\right) \cap \pi_{A \cup r_2.B \cup C}\left(\sigma_{r_1.B=r_2.B}\left(r_1 \times r_2\right)\right) \\
&= \left(\pi_A\left(r_1\right) \times \pi_{B \cup C} r_2\right) \cap \pi_{A \cup r_2.B \cup C}\left(\sigma_{r_1.B=r_2.B}\left(r_1 \times r_2\right)\right) \\
&= \pi_{A \cup r_2.B \cup C}\left(r_1 \times r_2\right) \cap \pi_{A \cup r_2.B \cup C}\left(\sigma_{r_1.B=r_2.B}\left(r_1 \times r_2\right)\right) \\
&= \pi_{A \cup r_2.B \cup C}\left(\sigma_{r_1.B=r_2.B}\left(r_1 \times r_2\right)\right) \qquad \text{since } \sigma_{r_1.B=r_2.B}\left(r_1 \times r_2\right) \subseteq \left(r_1 \times r_2\right)
\end{aligned}
$$

$$= \pi_{A \cup r_2.B \cup C}(r_1 \times r_2) \cap \pi_{A \cup r_1.B \cup C}(r_1 \times r_2)$$
$$= (\pi_A(r_1) \times r_2) \cap (r_1 \times \pi_C(r_2))$$
$$= \tilde{e}_1 \cap \tilde{e}_2$$

□

A.2 Theorems

PROOF (THEOREM 1, page 12): Lemma 3 shows that set containment division ($\div_1^*$) and great divide ($\div_3^*$) are equivalent, and Lemma 4 shows that great divide ($\div_3^*$) and generalized division ($\div_2^*$) are equivalent. By transitivity we see that all three operators are equivalent.

□

PROOF (THEOREM 2, page 16): Let $R_1(A_1)$, $R_1(A_2)$, and $R_1(A_3)$ be the schemas of relations r_1, r_2, and r_3 in the expression $r_1 \div r_2 = r_3$. According to the definition of division, the divisor has n attributes and the dividend has $m+n$ attributes, where $m > 0$ and $n > 0$. Since $m+n > n$, it is impossible to interchange r_1 and r_2, i.e., $r_2 \div r_1$ is an invalid expression.

□

PROOF (THEOREM 3, page 16): We show that division is non-associative even if we assume that the relation schemas are valid. We prove the theorem by contradiction. Suppose, A_1, A_2, and A_3 are the attributes of the relations r_1, r_2, and r_3, respectively. If the two expressions are equivalent then the corresponding quotient relation schemas are the same. The relation schema of $r_1 \div (r_2 \div r_3)$ is defined by expression $e_1 = A_1 - (A_2 - A_3)$ and the schema of $(r_1 \div r_2) \div r_3$ is $e_2 = (A_1 - A_2) - A_3$. We try to show that $t \in e_1 \leftrightarrow t \in e_2$ is a tautology, i.e., the expression is true for any value of tuple t. Since $t \in e_1 \leftrightarrow t \in e_2 = (t \in e_1 \rightarrow t \in e_2) \wedge (t \in e_2 \rightarrow t \in e_1)$, we can analyze each implication separately:

$$\begin{aligned}
t \in e_2 \rightarrow t \in e_1 &\Leftrightarrow t \notin e_2 \vee t \in e_1 \\
&\Leftrightarrow t \notin ((A_1 - A_2) - A_3) \vee t \in (A_1 - (A_2 - A_3)) \\
&\Leftrightarrow (t \notin (A_1 - A_2) \vee t \in A_3) \vee (t \in A_1 \wedge t \notin (A_2 - A_3)) \\
&\Leftrightarrow (t \notin A_1 \vee t \in A_2 \vee t \in A_3) \vee (t \in A_1 \wedge (t \notin A_2 \vee t \in A_3)) \\
&\Leftrightarrow (t \notin A_1 \vee t \in A_2 \vee t \in A_3 \vee t \in A_1) \wedge \\
&\quad\ (t \notin A_1 \vee t \in A_2 \vee t \in A_3 \vee t \notin A_2 \vee t \in A_3) \\
&\Leftrightarrow \mathit{true} \wedge \mathit{true} \\
&\Leftrightarrow \mathit{true}
\end{aligned}$$

Now, we analyze the opposite direction of the equivalence:

$$\begin{aligned}
t \in e_1 \rightarrow t \in e_2 &\Leftrightarrow t \notin e_1 \vee t \in e_2 \\
&\Leftrightarrow t \notin (A_1 - (A_2 - A_3)) \vee t \in ((A_1 - A_2) - A_3) \\
&\Leftrightarrow t \notin A_1 \vee t \in (A_2 - A_3) \vee (t \in (A_1 - A_2) \wedge t \notin A_3)
\end{aligned}$$

$$
\begin{aligned}
&\Leftrightarrow t \notin A_1 \vee (t \in A_2 \wedge t \notin A_3) \vee (t \in A_1 \wedge t \notin A_2 \wedge t \notin A_3) \\
&\Leftrightarrow (t \notin A_1 \vee (t \in A_2 \wedge t \notin A_3) \vee t \in A_1) \wedge \\
&\quad (t \notin A_1 \vee (t \in A_2 \wedge t \notin A_3) \vee t \notin A_2) \wedge \\
&\quad (t \notin A_1 \vee (t \in A_2 \wedge t \notin A_3) \vee t \notin A_3) \\
&\Leftrightarrow \mathit{true} \wedge (t \notin A_1 \vee (t \in A_2 \wedge t \notin A_3) \vee t \notin A_2) \wedge (t \notin A_1 \vee t \notin A_3) \\
&\Leftrightarrow t \notin A_1 \vee ((t \in A_2 \wedge t \notin A_3) \vee t \notin A_2 \vee t \notin A_3) \\
&\Leftrightarrow t \notin A_1 \vee t \notin A_2 \vee t \notin A_3 \\
&\Leftrightarrow t \notin A_1 \cap A_2 \cap A_3
\end{aligned}
$$

Since $t \in e_1 \leftrightarrow t \in e_2 = t \notin A_1 \cap A_2 \cap A_3 \neq \mathit{true}$ for any value of t (for a value $t \in A_1 \cap A_2 \cap A_3$ it is *false*), we have found a contradiction to our assumption that the expression is a tautology.

□

PROOF (THEOREM 4, page 65): According to [MGM03], the comparison factor c_{APSJ} of the adaptive pick-and-sweep join algorithm is

$$c_{\mathrm{APSJ}}(p) = 1 - p^{\theta_1} + p^{\theta_2 l + \theta_1}.$$

The function c is minimized for some $p = p_0$ when $c'(p_0) = 0$ and $c'(p_0 \mp 0) \lessgtr 0$.

$$
\begin{aligned}
c'(p_0) &= 0 \\
-\theta_1 p_0^{\theta_1 - 1} + (\theta_2 l + \theta_1) p_0^{\theta_2 l + \theta_1 - 1} &= 0 \\
p_0^{\theta_1 - 1} \left((\theta_2 l + \theta_1) p_0^{\theta_2 l} - \theta_1 \right) &= 0 \\
(\theta_2 l + \theta_1) p_0^{\theta_2 l} &= \theta_1 \\
p_0 &= \left(\frac{\theta_1}{\theta_2 l + \theta_1} \right)^{\frac{1}{\theta_2 l}}
\end{aligned}
$$

Now that we have determined p_0, we check if c' is minimal for this value. We have to see if and how the sign changes of $c'(p_0 + \delta)$ for small values of δ:

$$c'(p_0 + \delta) = (p_0 + \delta)^{\theta_1 - 1} \left((\theta_2 l + \theta_1)(p_0 + \delta)^{\theta_2 l} - \theta_1 \right)$$

We see that $\lim_{\delta \to 0 \mp 0} (p_0 + \delta)^{\theta_1 - 1} > 0$. Hence, we only have to check if the sign switches from negative to positive for the expression $\lim_{\delta \to 0 \mp 0} (\theta_2 l + \theta_1)(p_0 + \delta)^{\theta_2 l} - \theta_1$. We replace δ by some ε value such that $p_0 + \delta = p_0 \varepsilon$, i.e., $\varepsilon = 1 + \frac{\delta}{p_0}$:

$$
\begin{aligned}
\lim_{\delta \to 0 \mp 0} (\theta_2 l + \theta_1)(p_0 + \delta)^{\theta_2 l} - \theta_1 &= \lim_{\varepsilon \to 1 \mp 0} (\theta_2 l + \theta_1)(p_0 \varepsilon)^{\theta_2 l} - \theta_1 \\
&= \lim_{\varepsilon \to 1 \mp 0} (\theta_2 l + \theta_1) p_0^{\theta_2 l} \varepsilon^{\theta_2 l} - \theta_1
\end{aligned}
$$

$$= \lim_{\varepsilon \to 1 \mp 0} \theta_1 \left(\varepsilon^{\theta_2 l} - 1 \right) \lessgtr 0$$

Thus, we have shown that $c'(p)$ turns from negative to positive for increasing values $p \approx p_0$. Since there are no values other than p_0 where $c'(p)$ becomes 0, we have proved that c has its (absolute) minimum value for p_0.

By replacing $\frac{|r_1|}{|r_2|}$ by λ, we obtain

$$p_0 = \left(\frac{\lambda}{\lambda + l} \right)^{\frac{1}{\theta_2 l}}.$$

The corresponding optimal comparison factor $c_{\text{APSJ}}^{\text{opt}} = c(p_0)$ is

$$c_{\text{APSJ}}^{\text{opt}} = 1 - \frac{k-1}{\lambda + k - 1} \left(\frac{\lambda}{\lambda + k - 1} \right)^{\frac{\lambda}{k-1}}.$$

□

PROOF (THEOREM 5, page 65): According to [MGM03], the replication factor r_{APSJ} of the adaptive pick-and-sweep algorithm is

$$\begin{aligned} r_{\text{APSJ}} &= \frac{\sum_{i=0}^{l} |r_1^i| + |r_2^i|}{|r_1| + |r_2|} \\ &= \frac{|r_2| + |r_1^0| + \sum_{i=1}^{l} |r_1^i|}{|r_2| + |r_1|} \\ &= \frac{|r_2| + |r_1| + \left(1 - p^{\theta_1}\right) |r_1|}{|r_2| + |r_1|} \\ &= \frac{|r_2|}{|r_2| + |r_1|} + \frac{|r_1|}{|r_1| + |r_2|} \left(1 + l \left(1 - p^{\theta_1} \right) \right). \end{aligned}$$

By replacing $\frac{|r_1|}{|r_2|}$ by ρ, p by p_0, and l by $k-1$ we get the replication factor

$$r_{\text{APSJ}} = \frac{1}{\rho + 1} + \frac{\rho}{\rho + 1} \left(k - (k-1) \left(\frac{\lambda}{\lambda + k - 1} \right)^{\frac{\lambda}{k-1}} \right).$$

How can the replication factor be minimized? We find that the first derivation of $r(p) = \frac{1}{\rho+1} + \frac{\rho}{\rho+1} \left(1 + (k-1)\left(1 - p^{\theta_1}\right)\right)$ is $r'(p) = (1-k)\,\theta_1 p^{\theta_1 - 1} \neq 0$ for $k > 1$. Hence, we only need to check the interval boundaries of $0 \leq p \leq 1$: $r(0) = \frac{k\rho+1}{\rho+1} > 1$ and $r(1) = 1$. The replication factor is thus minimized for $p = 1$.

□

A.3 Laws

PROOF (LAW 1, page 16): Let

$$\begin{aligned} e &= r_1 \ltimes \left(r_1 \div r_2'\right) \\ &= \{t | t \in r_1 \wedge t.A \in \left(r_1 \div r_2'\right)\} \\ &= \{t | t \in r_1 \wedge t.A \in \{u | \exists u_1 : u = u_1.A \wedge u_1 \in r_1 \wedge r_2' \subseteq \{y | \, (u,y) \in r_1\}\}\} \\ &= \{t | t \in r_1 \wedge \exists u_1 : t.A = u_1.A \wedge u_1 \in r_1 \wedge r_2' \subseteq \{y | \, (t.A,y) \in r_1\}\}. \end{aligned}$$

Since $t \in r_1$ already implies $\exists u_1 : t.A = u_1.A \wedge u_1 \in r_1$, we have

$$e = \{t | t \in r_1 \wedge r_2' \subseteq \{y | \, (t.A,y) \in r_1\}\}.$$

Hence,

$$\begin{aligned} \left(r_1 \ltimes \left(r_1 \div r_2'\right)\right) \div r_2'' &= \{s | \exists s_1 : s = s_1.A \wedge s_1 \in e \wedge r_2'' \subseteq \{z | \, (s,z) \in e\}\} \\ &= \{s | \exists s_1 : s = s_1.A \wedge s_1 \in \{t | t \in r_1 \wedge r_2' \subseteq \{y | \, (t.A,y) \in r_1\}\} \wedge \\ &\qquad r_2'' \subseteq \{z | \, (s,z) \in \{t | t \in r_1 \wedge r_2' \subseteq \{y | \, (t.A,y) \in r_1\}\}\}\} \\ &= \{s | \exists s_1 : s = s_1.A \wedge s_1 \in r_1 \wedge r_2' \subseteq \{y | \, (s_1.A,y) \in r_1\}\} \wedge \\ &\qquad r_2'' \subseteq \{z | \, (s,z) \in r_1 \wedge r_2' \subseteq \{y | \, ((s,z).A,y) \in r_1\}\}\} \\ &= \{s | \exists s_1 : s = s_1.A \wedge s_1 \in r_1 \wedge r_2' \subseteq \{y | \, (s,y) \in r_1\}\} \wedge \\ &\qquad r_2'' \subseteq \{z | \, (s,z) \in r_1 \wedge r_2' \subseteq \{y | \, (s,y) \in r_1\}\}\} \\ &= \{s | \exists s_1 : s = s_1.A \wedge s_1 \in r_1 \wedge r_2' \subseteq \{y | \, (s,y) \in r_1\}\} \wedge \\ &\qquad r_2'' \subseteq \{z | \, (s,z) \in r_1\} \wedge r_2'' \subseteq \{z | r_2' \subseteq \{y | \, (s,y) \in r_1\}\}\} \\ &= \{s | \exists s_1 : s = s_1.A \wedge s_1 \in r_1 \wedge r_2' \subseteq \{y | \, (s,y) \in r_1\}\} \wedge \\ &\qquad r_2'' \subseteq \{z | \, (s,z) \in r_1\}\} \\ &= \{s | \exists s_1 : s = s_1.A \wedge s_1 \in r_1 \wedge \left(r_2' \cup r_2''\right) \subseteq \{y | \, (s,y) \in r_1\}\} \\ &= r_1 \div \left(r_2' \cup r_2''\right) \end{aligned}$$

□

PROOF (LAW 2, page 17): We prove that if condition $c_1(r_1', r_1'')$ is *true* then $(r_1' \cup r_1'') \div r_2 = (r_1' \div r_2) \cup (r_1'' \div r_2)$.

We use the following algebraic laws as propositions, where we assume that relations r_1, r_2, s_1, and s_2 have the same schema:

(P1) $\sigma_\theta(r_1 \cup r_2) = \sigma_\theta(r_1) \cup \sigma_\theta(r_2)$ [GMUW02]

(P2) $\pi_A(r_1 \cup r_2) = \pi_A(r_1) \cup \pi_A(r_2)$, where A is any subset of r_1's and r_2's relation schemas.

$$
\begin{aligned}
\left(r_1' \cup r_1''\right) \div r_2 &= \bigcap_{t \in r_2} \pi_A\left(\sigma_{B=t}\left(r_1' \cup r_1''\right)\right) && \text{(Definition 4)} \\
&= \bigcap_{t \in r_2} \pi_A\left(\sigma_{B=t}\left(r_1'\right) \cup \sigma_{B=t}\left(r_1''\right)\right) && \text{(P1)} \\
&= \bigcap_{t \in r_2} \pi_A\left(\sigma_{B=t}\left(r_1'\right)\right) \cup \pi_A\left(\sigma_{B=t}\left(r_1''\right)\right) && \text{(P2)} \\
&= \bigcap_{t \in r_2} \pi_A\left(\sigma_{B=t}\left(r_1'\right)\right) \cup \bigcap_{t \in r_2} \pi_A\left(\sigma_{B=t}\left(r_1''\right)\right) && \text{(see below)} \\
&= \left(r_1' \div r_2\right) \cup \left(r_1'' \div r_2\right) && \text{(Definition 4)}
\end{aligned}
$$

To show the missing step in the above transformation, we restrict ourselves to the case where r_2 contains two tuples, t_1 and t_2, only. This can easily be extended to the general case. Consider

$$
\begin{aligned}
&\bigcap_{t \in \{t_1, t_2\}} \pi_A\left(\sigma_{B=t}\left(r_1'\right)\right) \cup \pi_A\left(\sigma_{B=t}\left(r_1''\right)\right) \\
&\quad = \left(\pi_A\left(\sigma_{B=t_1}\left(r_1'\right)\right) \cup \pi_A\left(\sigma_{B=t_1}\left(r_1''\right)\right)\right) \cap \left(\pi_A\left(\sigma_{B=t_2}\left(r_1'\right)\right) \cup \pi_A\left(\sigma_{B=t_2}\left(r_1''\right)\right)\right) \\
&\quad = \underbrace{\left(\pi_A\left(\sigma_{B=t_1}\left(r_1'\right)\right) \cap \pi_A\left(\sigma_{B=t_2}\left(r_1'\right)\right)\right)}_{S_{r'r'}} \cup \underbrace{\left(\pi_A\left(\sigma_{B=t_1}\left(r_1''\right)\right) \cap \pi_A\left(\sigma_{B=t_2}\left(r_1''\right)\right)\right)}_{S_{r''r''}} \cup \\
&\qquad \underbrace{\left(\pi_A\left(\sigma_{B=t_1}\left(r_1'\right)\right) \cap \pi_A\left(\sigma_{B=t_2}\left(r_1''\right)\right)\right)}_{S_{r'r''}} \cup \underbrace{\left(\pi_A\left(\sigma_{B=t_1}\left(r_1''\right)\right) \cap \pi_A\left(\sigma_{B=t_2}\left(r_1'\right)\right)\right)}_{S_{r''r'}}.
\end{aligned}
$$

To show that this is equal to

$$
\bigcap_{t \in \{t_1, t_2\}} \pi_A\left(\sigma_{B=t}\left(r_1'\right)\right) \cup \bigcap_{t \in \{t_1, t_2\}} \pi_A\left(\sigma_{B=t}\left(r_1''\right)\right),
$$

we need to argue why $S_{r'r''}$ and $S_{r''r'}$ are subsets of $S_{r'r'} \cup S_{r''r''}$. The basic idea is that $S_{r'r''}$ and $S_{r''r'}$ are sufficiently restricted by precondition c_1. In the following we will show with an indirect proof that $S_{r'r''}$ meets this requirement if precondition c_1 is true. The proof for $S_{r''r'}$ is analogous.

Assume that $S_{r'r''} \not\subseteq S_{r'r'} \cup S_{r''r''}$. Remember that we are still in the case where $r_2 = \{t_1, t_2\}$. Hence,

$$
\begin{aligned}
\exists t : t \in S_{r'r''} \wedge t \notin S_{r'r'} \wedge t \notin S_{r''r''} \Leftrightarrow\ & \exists t : t \in \pi_A\left(\sigma_{B=t_1}\left(r_1'\right)\right) \cap \pi_A\left(\sigma_{B=t_2}\left(r_1''\right)\right) \wedge \\
& t \notin \left(\pi_A\left(\sigma_{B=t_1}\left(r_1'\right)\right) \cap \pi_A\left(\sigma_{B=t_2}\left(r_1'\right)\right)\right) \wedge \\
& t \notin \left(\pi_A\left(\sigma_{B=t_1}\left(r_1''\right)\right) \cap \pi_A\left(\sigma_{B=t_2}\left(r_1''\right)\right)\right) \\
\Leftrightarrow\ & \exists t : t.t_1 \in r_1' \wedge t.t_2 \in r_1'' \wedge \\
& \neg\left(t.t_1 \in r_1' \wedge t.t_2 \in r_1'\right) \wedge \\
& \neg\left(t.t_1 \in r_1'' \wedge t.t_2 \in r_1''\right)
\end{aligned}
$$

$$\begin{aligned}
&\Rightarrow \exists t : r_2 \subseteq \pi_B\left(\sigma_{A=t}\left(r_1'\right) \cup \sigma_{A=t}\left(r_1''\right)\right) \wedge \\
&\quad \neg\left(r_2 \subseteq \pi_B\left(\sigma_{A=t}\left(r_1'\right)\right)\right) \wedge \\
&\quad \neg\left(r_2 \subseteq \pi_B\left(\sigma_{A=t}\left(r_1''\right)\right)\right) \\
&\Leftrightarrow \neg c_1(r_1', r_1'') \text{ for } a = t.
\end{aligned}$$

□

PROOF (LAW 3, page 17): We use the following algebraic laws given in [GMUW02] as propositions:

(P1) $\sigma_\theta(r_1 - r_2) = \sigma_\theta(r_1) - \sigma_\theta(r_2)$,

(P2) $\pi_X(\sigma_\theta(r)) = \pi_X(\sigma_\theta(\pi_Y(r)))$, where Y contains X and the attributes mentioned in condition θ, in particular, $\sigma_{p(A)}(\pi_A(r_1)) = \pi_A(\sigma_{p(A)}(\pi_A(r_1))) = \pi_A(\sigma_{p(A)}(r_1))$, and

(P3) $\sigma_\theta(r_1 \times r_2) = \sigma_\theta(r_1) \times r_2$ if θ restricts attributes of r_1, only.

$$\begin{aligned}
\sigma_{p(A)}(r_1 \div r_2) &= \sigma_{p(A)}\left(\pi_A(r_1) - \pi_A\left(\left(\pi_A(r_1) \times r_2\right) - r_1\right)\right) && \text{(Definition 3)} \\
&= \sigma_{p(A)}\left(\pi_A(r_1)\right) - \sigma_{p(A)}\left(\pi_A\left(\left(\pi_A(r_1) \times r_2\right) - r_1\right)\right) && \text{(P1)} \\
&= \pi_A\left(\sigma_{p(A)}(r_1)\right) - \pi_A\left(\sigma_{p(A)}\left(\pi_A\left(\left(\pi_A(r_1) \times r_2\right) - r_1\right)\right)\right) && \text{(P2)} \\
&= \pi_A\left(\sigma_{p(A)}(r_1)\right) - \pi_A\left(\sigma_{p(A)}\left(\pi_A\left(\left(\pi_A(r_1) \times r_2\right)\right) - \sigma_{p(A)}(r_1)\right)\right) && \text{(P1)} \\
&= \pi_A\left(\sigma_{p(A)}(r_1)\right) - \pi_A\left(\pi_A\left(\sigma_{p(A)}\left(\left(\pi_A(r_1) \times r_2\right)\right) - \sigma_{p(A)}(r_1)\right)\right) && \text{(P2)} \\
&= \pi_A\left(\sigma_{p(A)}(r_1)\right) - \pi_A\left(\pi_A\left(\left(\sigma_{p(A)}\left(\pi_A(r_1)\right) \times r_2\right) - \sigma_{p(A)}(r_1)\right)\right) && \text{(P3)} \\
&= \pi_A\left(\sigma_{p(A)}(r_1)\right) - \pi_A\left(\pi_A\left(\left(\pi_A\left(\sigma_{p(A)}(r_1)\right) \times r_2\right) - \sigma_{p(A)}(r_1)\right)\right) && \text{(P2)} \\
&= \sigma_{p(A)}(r_1) \div r_2 && \text{(Definition 3)}
\end{aligned}$$

□

PROOF (LAW 4, page 17):

$$\begin{aligned}
r_1 \div \sigma_{p(B)}(r_2) &= \sigma_{p(B) \vee \neg p(B)}(r_1) \div \sigma_{p(B)}(r_2) \\
&= \left(\sigma_{p(B)}(r_1) \cup \sigma_{\neg p(B)}(r_1)\right) \div \sigma_{p(B)}(r_2) \\
&= \left(\sigma_{p(B)}(r_1) \div \sigma_{p(B)}(r_2)\right) \cup \left(\sigma_{\neg p(B)}(r_1) \div \sigma_{p(B)}(r_2)\right) && \text{(Law 2)} \\
&= \left(\sigma_{p(B)}(r_1) \div \sigma_{p(B)}(r_2)\right) \cup \emptyset \\
&= \sigma_{p(B)}(r_1) \div \sigma_{p(B)}(r_2)
\end{aligned}$$

We can apply Law 2 to the expression in line 2 because the law's precondition $c_2(\sigma_{p(B)}(r_1), \sigma_{\neg p(B)}(r_1))$ (and, of course, also c_1) is obviously fulfilled. □

PROOF (LAW 5, page 18): According to Codd's definition of division in tuple relational calculus (Definition 1), we can derive the following equivalences:

$$\begin{aligned}
(r_1' \div r_2) \cap (r_1'' \div r_2) &= \left\{t \,|\, t = t_1.A \wedge t_1 \in r_1' \wedge r_2 \subseteq \left\{y \,|\, (t,y) \in r_1'\right\}\right\} \cap \\
&\quad \left\{t \,|\, t = t_1.A \wedge t_1 \in r_1'' \wedge r_2 \subseteq \left\{y \,|\, (t,y) \in r_1''\right\}\right\} \\
&= \left\{t \,|\, t = t_1.A \wedge t_1 \in r_1' \wedge t_1 \in r_1'' \wedge r_2 \subseteq \left\{y \,|\, (t,y) \in r_1'\right\} \wedge \right. \\
&\quad \left. r_2 \subseteq \left\{y \,|\, (t,y) \in r_1''\right\}\right\} \\
&= \left\{t \,|\, t = t_1.A \wedge t_1 \in r_1' \wedge t_1 \in r_1'' \wedge \right. \\
&\quad \left. r_2 \subseteq \left\{y \,|\, (t,y) \in r_1'\right\} \cap \left\{y \,|\, (t,y) \in r_1''\right\}\right\} \\
&= \left\{t \,|\, t = t_1.A \wedge t_1 \in r_1' \wedge t_1 \in r_1'' \wedge r_2 \subseteq \left\{y \,|\, (t,y) \in r_1' \wedge (t,y) \in r_1''\right\}\right\} \\
&= \left\{t \,|\, t = t_1.A \wedge t_1 in r_1' \wedge t_1 \in r_1'' \wedge r_2 \subseteq \left\{y \,|\, (t,y) \in r_1' \cap r_1''\right\}\right\} \\
&= (r_1' \cap r_1'') \div r_2
\end{aligned}$$

□

PROOF (LAW 6, page 18): Our assumption that $\pi_A(r_1')$ and $\pi_A(r_1'')$ are disjoint is equivalent to $\pi_A(r_1') \cap \pi_A(r_1'') = \emptyset$. Hence, $(r_1' - r_1'') \div r_2 = r_1' \div r_2$. Therefore, we can show that

$$\begin{aligned}
\left(r_1' \div r_2\right) - \left(r_1'' \div r_2\right) &= \bigcap_{t \in r_2} \pi_A\left(\sigma_{B=t}\left(r_1'\right)\right) - \bigcap_{t \in r_2} \pi_A\left(\sigma_{B=t}\left(r_1''\right)\right) \qquad \text{(Definition 4)} \\
&= \bigcap_{t \in r_2} \pi_A\left(\sigma_{B=t}\left(r_1'\right)\right) \qquad \text{since } \pi_A\left(r_1'\right) \cap \pi_A\left(r_1''\right) = \emptyset \\
&= r_1' \div r_2
\end{aligned}$$

□

PROOF (LAW 7, page 18): We use the following algebraic laws as propositions:

(P1) $\sigma_\theta(r_1 \times r_2) = r_1 \times \sigma_\theta(r_2)$ for relations r_1 and r_2 with schemas $R_1(A)$ and $R_2(B)$, respectively, and θ contains only attributes in B.

(P2) $\pi_{B \cup C}(r_1 \times r_2) = \pi_B(r_1) \times \pi_C(r_2)$ for relations r_1 and r_2 with schemas $R_1(A \cup B)$ and $R_2(C \cup D)$, respectively.

(P3) $(r_1 \times r_2) \cap (r_1 \times r_3) = r_1 \times (r_2 \cap r_3)$ for relations r_1, r_2, and r_3 with schemas $R_1(A)$, $R_2(B)$, and $R_3(B)$, respectively.

$$\begin{aligned}
(r_1^* \times r_1^{**}) \div r_2 &= \bigcap_{t \in r_2} \pi_{A_1 \cup A_2}\left(\sigma_{B=t}\left(r_1^* \times r_1^{**}\right)\right) \qquad \text{(Definition 4)} \\
&= \bigcap_{t \in r_2} \pi_{A_1 \cup A_2}\left(r_1^* \times \sigma_{B=t}\left(r_1^{**}\right)\right) \qquad \text{(P1)}
\end{aligned}$$

$$
\begin{aligned}
&= \bigcap_{t \in r_2} \pi_{A_1}(r_1^*) \times \pi_{A_2}(\sigma_{B=t}(r_1^{**})) && \text{(P2)} \\
&= \pi_{A_1}(r_1^*) \times \bigcap_{t \in r_2} \pi_{A_2}(\sigma_{B=t}(r_1^{**})) && \text{(P3)} \\
&= r_1^* \times (r_1^{**} \div r_2) && \text{(Definition 4)}
\end{aligned}
$$

□

PROOF (LAW 8, page 19):

$$
\begin{aligned}
(r_1^* \times r_1^{**}) \div r_2 &= \bigcap_{t \in r_2} \pi_A(\sigma_{B=t}(r_1^* \times r_1^{**})) && \text{(Definition 4)} \\
&= \bigcap_{t \in r_2} \pi_A(\sigma_{B_1=t.B_1 \wedge B_2=t.B_2}(r_1^* \times r_1^{**})) \\
&= \bigcap_{t \in r_2} \pi_A(\sigma_{B_1=t.B_1 \wedge true}(r_1^* \times r_1^{**})) && \text{(since } \forall t' \in r_2 : t'.B_2 \in r_1^{**}\text{)} \\
&= \bigcap_{t \in r_2} \pi_A(\sigma_{B_1=t.B_1}(r_1^* \times r_1^{**})) \\
&= \bigcap_{t \in \pi_{B_1}(r_2)} \pi_A(\sigma_{B_1=t}(r_1^* \times r_1^{**})) \\
&= \bigcap_{t \in \pi_{B_1}(r_2)} \pi_A(\sigma_{B_1=t}(r_1^*)) \\
&= r_1^* \div \pi_{B_1}(r_2) && \text{(Definition 4)}
\end{aligned}
$$

□

PROOF (LAW 9, page 21):

$$
\begin{aligned}
(r_1 \div r_2) \ltimes r_3 &= \pi_A((r_1 \div r_2) \bowtie r_3) && \text{(Definition of semi-join)} \\
&= \pi_A(\sigma_{A=A}((r_1 \div r_2) \times r_3)) && \text{(Definition of natural join)} \\
&= \pi_A\left(\bigcup_{t \in r_3} \sigma_{A=t.A}((r_1 \div r_2) \times (t))\right) && \text{(Definition of Cartesian product)} \\
&= \bigcup_{t \in r_3} \sigma_{A=t.A}(r_1 \div r_2) && \text{(Remove duplicate attribute set } A\text{)} \\
&= \bigcup_{t \in r_3} (\sigma_{A=t.A}(r_1) \div r_2) && \text{(Law 3)} \\
&= \left(\bigcup_{t \in r_3} \sigma_{A=t.A}(r_1)\right) \div r_2 && \text{(Law 2)} \\
&= (r_1 \ltimes r_3) \div r_2 && \text{(Definition of semi-join)}
\end{aligned}
$$

□

PROOF (LAW 10, page 22): As defined in Section 2.3.1, the schema of r_1 is $R_1(A \cup B)$ and the schema of r_2 is $R_2(B)$. We will show the three cases separately.

Case 1: $\sigma_{c=0}\left(\gamma_{\text{count}(B)\to c}(r_2)\right) \neq \emptyset$, i.e., $r_2 = \emptyset$.

It follows from the definition that $r_1 \div r_2 = r_1$.

Case 2: $\sigma_{c=1}\left(\gamma_{\text{count}(B)\to c}(r_2)\right) \neq \emptyset$, i.e., $|r_2| = 1$.

Assume that, w.l.o.g., $r_2 = \{t_2\}$. We have to show the following:

$$
\begin{aligned}
r_1 \div r_2 &= \pi_A(r_1 \ltimes r_2) \\
\bigcap_{t \in r_2} \pi_A(\sigma_{B=t}(r_1)) &= \pi_A(r_1 \ltimes r_2) && \text{(Definition 4)} \\
\pi_A(\sigma_{B=t_2}(r_1)) &= \pi_A(r_1 \ltimes r_2) && \text{(with } r_2 = \{t_2\}\text{)}
\end{aligned}
$$

Now, let us consider $\sigma_{B=t_2}(r)$:

$$
\begin{aligned}
\sigma_{B=t_2}(r_1) &= \{t \in r_1 \mid t.B = t_2\} \\
&= \{t \in r_1 \mid t.B \in r_2\} \\
&= r_1 \ltimes r_2.
\end{aligned}
$$

Hence, $\pi_A(\sigma_{B=t_2}(r_1)) = \pi_A(r_1 \ltimes r_2)$.

Case 3: $\sigma_{c>1}\left(\gamma_{\text{count}(B)\to c}(r_2)\right) \neq \emptyset$, i.e., $|r_2| > 1$.

From the construction of r_1 as $r_1 = {}_A\gamma_{f(X)\to B}(r_0)$ we know that $\forall t_1, t_2 \in r_1 : t_1.A \neq t_2.A$. From the precondition of Case 3 we also know that $\exists t_1, t_2 \in r_2 : t_1 \neq t_2$. With Definition 4 of the division operator the claim can be shown by a simple indirect proof. □

PROOF (LAW 11, page 22): Let $e = \pi_A(r_1 \ltimes r_2) = \{t_a \mid \exists t_b \in r_2 : (t_a, t_b) \in r_1\}$. We have to show three cases:

Case 1: $|e| > 1$:

In this case there exist $t_{a_1}, t_{a_2} \in e$ with $t_{a_1} \neq t_{a_2}$. Hence, there also exist $t_{b_1}, t_{b_2} \in r_2$ with $t_{b_1} \neq t_{b_2}$ and $(t_{a_1}, t_{b_1}) \in r_1$ and $(t_{a_2}, t_{b_2}) \in r_1$. Considering the precondition of the law $\forall (t_{a_1}, t_{b_1}), (t_{a_2}, t_{b_2}) \in r_1 : t_{b_1} \neq t_{b_2}$ this implies that

(a) $(t_{a_1}, t_{b_2}) \notin r_1$ and

(b) $\forall t_{a_i} \neq t_{a_1} : (t_{a_i}, t_{b_1}) \notin r_1$.

Hence, there is no t_a such that $r_2 \subseteq \{y \mid (t_a, y) \in r_1\}$. It directly follows that $r_1 \div r_2 = \{t_a \mid \exists t_b \in r_2 : (t_a, t_b) \in r_1 \wedge r_2 \subseteq \{y \mid (t_a, y) \in r_1\}\} = \emptyset$.

Case 2: $|e| < 1$:

Consider

$$\begin{aligned} r_1 \div r_2 &= \{t_a | \exists t_b \in r_2 : (t_a, t_b) \in r_1 \wedge r_2 \subseteq \{y | (t_a, y) \in r_1\}\} \\ &\subseteq \{t_a | \exists b \in r_2 : (t_a, t_b) \in r_1\} \\ &= \pi_A (r_1 \ltimes r_2) \\ &= \emptyset \end{aligned}$$

Case 3: $|e| = 1$:

Because of the precondition that the divisor attribute set $r_2.B$ is a foreign key referencing r_1 we have $\pi_B (r_1) \supseteq r_2$. This implies $\forall t_b \in r_2 \; \exists t_a : (t_a, t_b) \in r_1$.

With $|e| = |\{t_a | \exists t_b \in r_2 : (t_a, t_b) \in r_1\}| = 1$ we conclude that $\exists t_a : \forall t_b \in r_2 \, (t_a, t_b) \in r_1$, which implies that $|r_1 \div r_2| = |\{t_a | \exists t_b \in r_2 : (t_a, t_b) \in r_1 \wedge r_2 \subseteq \{y | (t_a, y) \in r_1\}\}| \geq 1$.

In Case 2 we have shown that $r_1 \div r_2 \subseteq \pi_A (r_1 \ltimes r_2)$. From this it follows that $r_1 \div r_2 = \pi_A (r_1 \ltimes r_2)$.

□

PROOF (LAW 12, page 24):

$$\begin{aligned} r_1 \div^* \left(r_2' \cup r_2''\right) &= \bigcup_{t \in \pi_C \left(r_2' \cup r_2''\right)} \left(r_1 \div \pi_B \left(\sigma_{C=t} \left(r_2' \cup r_2''\right)\right)\right) \times (t) \qquad \text{(Definition 8)} \\ &= \left(\bigcup_{t \in \pi_C \left(r_2'\right)} \left(r_1 \div \pi_B \left(\sigma_{C=t} \left(r_2' \cup r_2''\right)\right)\right) \times (t) \right) \cup \\ &\quad \left(\bigcup_{t \in \pi_C \left(r_2''\right)} \left(r_1 \div \pi_B \left(\sigma_{C=t} \left(r_2' \cup r_2''\right)\right)\right) \times (t) \right) \end{aligned}$$

From our assumption $\pi_C (r_2') \cap \pi_C (r_2'') = \emptyset$ it follows for all $t \in \pi_C (r_2')$ that $\sigma_{C=t} (r_2'') = \emptyset$ and hence $\sigma_{C=t} (r_2' \cup r_2'') = \sigma_{C=t} (r_2')$. Similarly,

$$\begin{aligned} \pi_C \left(r_2'\right) \cap \pi_C \left(r_2''\right) = \emptyset &\Rightarrow \forall t \in \pi_C \left(r_2''\right) : \sigma_{C=t} \left(r_2''\right) = \emptyset \\ &\Leftrightarrow \forall t \in \pi_C \left(r_2''\right) : \sigma_{C=t} \left(r_2' \cup r_2''\right) = \sigma_{C=t} \left(r_2'\right). \end{aligned}$$

Hence, we have

$$r_1 \div^* \left(r_2' \cup r_2''\right) = \left(\bigcup_{t \in \pi_C \left(r_2'\right)} \left(r_1 \div \pi_B \left(\sigma_{C=t} \left(r_2'\right)\right)\right) \times (t) \right) \cup$$

$$\left(\bigcup_{t \in \pi_C\left(r_2''\right)} \left(r_1 \div \pi_B\left(\sigma_{C=t}\left(r_2''\right)\right)\right) \times (t) \right)$$
$$= \left(r_1 \div^* r_2'\right) \cup \left(r_1 \div^* r_2''\right). \qquad \text{(Definition 8)}$$

□

PROOF (LAW 13, page 24):

$$\begin{aligned}
\sigma_{p(A)}\left(r_1 \div^* r_2\right) &= \sigma_{p(A)}\left(\bigcup_{t \in \pi_C(r_2)} \left(r_1 \div \pi_B\left(\sigma_{C=t}\left(r_2\right)\right)\right) \times (t) \right) \qquad \text{(Definition 8)} \\
&= \bigcup_{t \in \pi_C(r_2)} \sigma_{p(A)}\left(\left(r_1 \div \pi_B\left(\sigma_{C=t}\left(r_2\right)\right)\right) \times (t)\right) \\
&= \bigcup_{t \in \pi_C(r_2)} \left(\sigma_{p(A)}\left(r_1\right) \div \pi_B\left(\sigma_{C=t}\left(r_2\right)\right)\right) \times (t) \\
&= \sigma_{p(A)}\left(r_1\right) \div^* r_2 \qquad \text{(Definition 8)}
\end{aligned}$$

□

PROOF (LAW 14, page 24):

$$\begin{aligned}
\sigma_{p(C)}\left(r_1 \div^* r_2\right) &= \sigma_{p(C)}\left(\bigcup_{t \in \pi_C(r_2)} \left(r_1 \div \pi_B\left(\sigma_{C=t}\left(r_2\right)\right)\right) \times (t) \right) \qquad \text{(Definition 8)} \\
&= \bigcup_{t \in \sigma_{p(C)}(\pi_C(r_2))} \left(r_1 \div \pi_B\left(\sigma_{C=t}\left(r_2\right)\right)\right) \times (t) \\
&= \bigcup_{t \in \sigma_{p(C)}(\pi_C(r_2))} \left(r_1 \div \pi_B\left(\sigma_{C=t}\left(\sigma_{p(C)}\left(r_2\right)\right)\right)\right) \times (t) \\
&= \bigcup_{t \in \pi_C\left(\sigma_{p(C)}(r_2)\right)} \left(r_1 \div \pi_B\left(\sigma_{C=t}\left(\sigma_{p(C)}\left(r_2\right)\right)\right)\right) \times (t) \\
&= r_1 \div^* \sigma_{p(C)}\left(r_2\right) \qquad \text{(Definition 8)}
\end{aligned}$$

□

PROOF (LAW 15, page 24):

$$\begin{aligned}
\sigma_{p(B)}\left(r_1\right) \div^* \sigma_{p(B)}\left(r_2\right) &= \bigcup_{t \in \pi_C\left(\sigma_{p(B)}(r_2)\right)} \left(\sigma_{p(B)}\left(r_1\right) \div \pi_B\left(\sigma_{C=t}\left(\sigma_{p(B)}\left(r_2\right)\right)\right)\right) \times (t) \text{ (Def. 8)} \\
&= \bigcup_{t \in \pi_C\left(\sigma_{p(B)}(r_2)\right)} \left(\sigma_{p(B)}\left(r_1\right) \div \pi_B\left(\sigma_{p(B)}\left(\sigma_{C=t}\left(r_2\right)\right)\right)\right) \times (t)
\end{aligned}$$

$$
\begin{aligned}
&= \bigcup_{t \in \pi_C\left(\sigma_{p(B)}(r_2)\right)} \left(\sigma_{p(B)}(r_1) \div \sigma_{p(B)}\left(\pi_B\left(\sigma_{C=t}(r_2)\right)\right)\right) \times (t) \\
&= \bigcup_{t \in \pi_C\left(\sigma_{p(B)}(r_2)\right)} \left(r_1 \div \sigma_{p(B)}\left(\pi_B\left(\sigma_{C=t}(r_2)\right)\right)\right) \times (t) \qquad \text{(Law 4)} \\
&= \bigcup_{t \in \pi_C\left(\sigma_{p(B)}(r_2)\right)} \left(r_1 \div \pi_B\left(\sigma_{p(B)}\left(\sigma_{C=t}(r_2)\right)\right)\right) \times (t) \\
&= \bigcup_{t \in \pi_C\left(\sigma_{p(B)}(r_2)\right)} \left(r_1 \div \pi_B\left(\sigma_{C=t}\left(\sigma_{p(B)}(r_2)\right)\right)\right) \times (t) \\
&= r_1 \div^* \sigma_{p(B)}(r_2) \qquad \text{(Definition 8)}
\end{aligned}
$$

□

B

Pseudo Code

The following basic division algorithms (Algorithms 15–19) assume that the division's input consists of a dividend table r_1 and a divisor table r_2 with the schemas $R_1(a,b)$ and $R_2(b)$, respectively, where a and b are attributes, and the data types of $r_1.b$ and $r_2.b$ are compatible. The aim of these algorithms is to compute the rows in the quotient table $r_3 = r_1 \div r_2$, which has the schema $R_3(a)$.

Some of the algorithms employ hash table data structures, where *dht* and *qht* represent a divisor hash table and a quotient hash table, respectively.

```
//
// Preconditions:
//   none
//
Table table11 = new Table ("r1");
Table table12 = new Table ("r1");
Table table2  = new Table ("r2");
while (table2.hasNext())
  dht.insert(table2.next());
while (table11.hasNext()) {
  Row row11 = table11.next();
  if (!qht.contains(row11.a)) {
    while (table12.hasNext()) {
      Row row12 = table12.next();
      if ((row11.a == row12.a) && (dht.contains(row12.b))
        set bit of row12.b in dht to one;
    }
    if (no bit in dht is equal to zero)
      output row (row11.a);
    reset all bits in dht to zero;
    qht.insert(row11.a);
    table12.reset();
  }
}
```

Algorithm 15: Nested-loop division

```
//
// Preconditions:
//   none
//
Table table1 = new Table ("r1");
Table table2 = new Table ("r2");
int divisorCount = 0;
// Build the divisor hash table dht
while (table2.hasNext()) {
  Row row2 = table2.next();
  row2.divisorNumber = divisorCount;
  dht.insert(row2);
  divisorCount++;
}
// Build the quotient hash table qht
while (table1.hasNext()) do
  Row row1 = table1.next();
  if (dht.contains(row1.b)) {
    if (!qht.contains(row1.a)) {
      Row q = new quotient candidate row created from quotient
              attributes of dividend row row1 including a bitmap
              initialized with zeroes;
      qht.insert(q);
    }
    set bit in q's bitmap corresponding t.divisorNumber;
  }
// Find result in the quotient hash table qht
foreach bucket in qht do
  foreach row q in bucket do
    if (the associated bitmap of q contains no zero)
      output row (q);
```

Algorithm 16: Classic hash-division

```
//
// Preconditions:
//   (1) The dividend table r1 is grouped on column a.
//
Table table1 = new Table ("r1");
Table table2 = new Table ("r2");
int divisorCount = 0;
while (table2.hasNext()) {
  divisorCount++;
  table2.next();
}
if (table1.hasNext()) {
  // Dividend is not empty
  Row row1 = table1.next();
  currentQuotient = row1.a;
}
while (table1.hasNext()) {
  quotientCount = 0;
  // Look at current row without proceeding to the next one
  Row row1 = table1.peek();
  while (table1.hasNext() && (row1.a == currentQuotient)) {
    quotientCount++;
    row1 = table1.next();
  }
  if (quotientCount == divisorCount)
    output row (currentQuotient);
  if (table1.hasNext()) {
    row1 = table1.next();
    currentQuotient = row1.a;
  }
}
```

Algorithm 17: Merge-count division

```
//
// Preconditions:
//   (1) The divisor table r2 is sorted in ascending order.
//   (2) The dividend is grouped on column a.
//   (3) For each dividend group, the dividend is sorted on column b in
//       ascending order.
//
// It is straightforward to modify the algorithm for a descending sort
// order.
//
Table table1 = new Table ("r1");
Table table2 = new Table ("r2");
boolean isFirstRow = true;
Row row1 = null;
Row row2 = null;
while (table1.hasNext()) {
  if (isFirstRow && table2.hasNext()) {
    // This is the first time that we fetch a dividend row
    row1 = table1.next();
    row2 = table2.next();
    isFirstRow = false;
  }
  currentQuotient = row1.a;
  while (table1.hasNext() && (row1.a == currentQuotient) &&
         table2.hasNext() && (row1.b <= row2.b)) {
    while (table1.hasNext() && (row1.a == currentQuotient) &&
           (row1.divisor < row2.divisor))
      row1 = table1.next();
    while (table1.hasNext() && (row1.a == currentQuotient) &&
           table2.hasNext() && (row1.b == row2.b)) {
      row1 = table1.next();
      row2 = table2.next();
    }
  }
  if (!table2.hasNext())
    // All divisor values of the divisor table have been matched
    output row (currentQuotient);
  // Reopen the sorted divisor table
  row2.reset();
  if (table2.hasNext())
    // Fetch the first divisor row
    row2 = table2.next();
  while (table1.hasNext() && (row1.a == currentQuotient))
    row1 = table1.next();
}
```

Algorithm 18: Merge-sort division

```
//
// Preconditions:
//   none
//
Table table1 = new Table ("r1");
Table table2 = new Table ("r2");
// Build the divisor hash table dht
while (table2.hasNext()) {
  Row row2 = table2.next();
  // Insert row with a new bitmap initialized with zeroes
  dht.insert(row2);
}
// Build the quotient hash table qht
int quotientCount = 0;
while (table1.hasNext()) {
  Row row1 = table1.next();
  int index;
  if (!qht.contains(row1.a)) {
    qht.insert(row1.a);
    // Assign a new quotient number to the quotient candidate row1.a
    index = qht.setQuotientNumber(row1.a, quotientCount);
    quotientCount++;
  }
  else
    index = qht.getQuotientNumber(row1.a);
  Row d = dht.get(row1.a);
  d.bitmap[index] = 1;
}
// Find result in the divisor hash table dht
if (!dht.isEmpty()) {
  // Bitmap initialized with ones
  Bitmap bitmap = new Bitmap();
  foreach bucket in dht do
    foreach row d in bucket do
      // Bit-wise AND operation
      bitmap = bitmap & d.bitmap;
  int index;
  foreach index value in bitmap with bit == 1 do {
    // Quotient row in qht associated with index
    Row q = qht.get(index);
    output row (q);
  }
}
```

Algorithm 19: Transposed hash-division

```
public class SubsetIndexNode implements Comparable, Comparator {
  private Integer set;         // Set identifier.
  private TreeSet elements;    // Sorted list of (Integer) elements.
  private TreeSet subsets;     // Sorted list of nodes that are subsets of this node.
  private TreeSet supersets;   // Sorted list of nodes that are supersets of this node.
  private boolean isMarked;    // Has the node been visited before?
  private boolean isInResult; // Is this part of the result of a subset search?
  private SubsetIndex subsetIndex; // "Callback anchor" for resetting marked nodes.

  SubsetIndexNode(SubsetIndex subsetIndex, Integer set) {...}
  public Integer getSet() {...}
  public TreeSet getElements() {...}
  public int cardinality() {...}
  public boolean isMarked() {...}
  public boolean isInResult() {...}
  public void addToResult() {...}
  public void resetIsInResult() {...}
  public void mark() {...}
  public void unmark() {...}
  public void markSupersetsSubtree() {...}
  private void markSubsetsSubtree(boolean mark) {...}
  public void addElement(Integer element) {...}
  public void addSubset(SubsetIndexNode node) {...}
  public void addSuperset(SubsetIndexNode node) {...}
  public boolean equals(Object other) {...}
  public boolean isSubsetOf(SubsetIndexNode other) {...}
  public boolean isSupersetOf(SubsetIndexNode other) {
  public int compareTo(Object other) {
  public int compare(Object object1, Object object2) {
  public void findSupersets(TreeSet resultNodes, SubsetIndexNode node) {...}
  public void removeSupersetsTreeElement(TreeSet otherElements) {...}

  // Add all index nodes that are a subset of the given node to the result set.
  public void findSubsets(TreeSet resultNodes, SubsetIndexNode node) {
    // We check a node only once.
    if (!isMarked()) {
      mark();
      if (this.isSubsetOf(node)) {
        resultNodes.add(this);
        Iterator iterator = supersets.iterator();
        while (iterator.hasNext()) {
          SubsetIndexNode indexNode = (SubsetIndexNode) iterator.next();
          indexNode.findSubsets(resultNodes, node);
        }
      }
    }
  }

  // Add all index nodes that are a subset of the given node to the result set.
  public void findCompressedSubsets(TreeSet resultNodes, SubsetIndexNode node) {
    // We add a node only once to the result.
    if (!isInResult) {
      Iterator subsetsIterator = subsets.iterator();
      boolean allSubsetsInResult = true;
      while (allSubsetsInResult && subsetsIterator.hasNext())
        allSubsetsInResult = ((SubsetIndexNode) subsetsIterator.next()).isInResult();

      if (allSubsetsInResult && this.reconstruct().isSubsetOf(node.reconstruct())) {
        addToResult();
        resultNodes.add(this);
        Iterator supersetsIterator = supersets.iterator();
        while (supersetsIterator.hasNext()) {
          SubsetIndexNode indexNode = (SubsetIndexNode) supersetsIterator.next();
          indexNode.findCompressedSubsets(resultNodes, node);
        }
      }
    }
  }
}
```

Algorithm 20: The SubsetIndexNode class as a Java code sample

```
public class SubsetIndex {
  private boolean isCompressed; // Compressed index?
  private static int MAX_CARDINALITY;
  private TreeSet cardinalitySiblings[];
  // Sorted list of nodes with same #elements.
  private TreeSet lowBorder; // Sorted list of nodes that have no subsets.
  private TreeSet highBorder; // Sorted list of nodes that have no supersets.
  private TreeSet markedNodes; // For garbage collection.
  private TreeSet resultNodes; // For garbage collection.

  SubsetIndex(boolean isCompressed, int maxCardinality) {...}

  // Add a node to the index.
  // Precondition: We start adding nodes with nodes of highest cardinality
  // in a given table and continue with nodes of same or decreasing cardinality.
  public void add(SubsetIndexNode node) {
    unmark();
    lowBorder.add(node);
    highBorder.add(node);
    // Check the nodes of the index by growing cardinality.
    // Start with supersets of size + 1.
    for (int c = node.cardinality() + 1; c <= MAX_CARDINALITY; c++) {
      Iterator iterator = cardinalitySiblings[c].iterator();
      while (iterator.hasNext()) {
        SubsetIndexNode indexNode = (SubsetIndexNode) iterator.next();
        if (node.isSubsetOf(indexNode) && !indexNode.isMarked()) {
          if (isCompressed)
            // Remove all elements from index node that are already in node.
            indexNode.getElements().removeAll(node.getElements());
          // Link both nodes to each other.
          indexNode.addSubset(node);
          node.addSuperset(indexNode);
          // Make sure that these nodes do not belong to the respectitve border.
          lowBorder.remove(indexNode);
          highBorder.remove(node);
          // We want to avoid to visit the subsets again.
          indexNode.markSupersetsSubtree();
        }
      }
    }
    cardinalitySiblings[node.cardinality()].add(node);
  }

  // Returns the nodes that are a subset of the given node.
  public Iterator subsetIterator(SubsetIndexNode node) {
    TreeSet resultNodes = new TreeSet();
    // Initially, all nodes of the index are unmarked.
    unmark();
    // Add all matching rows to the result set starting with the smallest sets and then
    // test bottom up.
    Iterator iterator = lowBorder.iterator();
    while (iterator.hasNext()) {
      SubsetIndexNode indexNode = (SubsetIndexNode) iterator.next();
      if (isCompressed)
        indexNode.findCompressedSubsets(resultNodes, node);
      else
        indexNode.findSubsets(resultNodes, node);
    }
    return resultNodes.iterator();
  }

  public Iterator supersetIterator(SubsetIndexNode node) {...}
  public void unmark() {...}
  public void resetIsInResult() {...}
  public boolean isCompressed() {...}
  public void addMarkedNode(SubsetIndexNode node) {...}
  public void addResultNode(SubsetIndexNode node) {...}
}
```

Algorithm 21: The SubsetIndex class as a Java code sample

C

Datasets

Sample type	Scale factor	Unnested layout			Nested layout	
		Transactions	Rows	Disk (bytes)	Rows	Disk (bytes)
original	1.0000	1 000 000	10 208 647	122 503 764	1 000 000	93 669 176
	0.5000	500 000	5 104 867	61 258 404	500 000	46 838 936
	0.1000	100 000	1 021 623	12 259 476	100 000	9 372 984
	0.0500	50 000	509 986	6 119 832	50 000	4 679 888
	0.0100	10 000	102 466	1 229 592	10 000	939 728
	0.0050	5 000	50 777	609 324	5 000	466 216
	0.0010	1 000	10 116	121 392	1 000	92 928
	0.0005	500	5 119	61 428	500	46 952
	0.0001	100	1 105	13 260	100	10 040
query	1.0000	989 981	5 612 509	67 350 108	989 981	56 779 844
	0.5000	494 976	2 807 910	33 694 920	494 976	28 402 992
	0.1000	99 061	562 253	6 747 036	99 061	5 686 756
	0.0500	49 489	279 909	3 358 908	49 489	2 833 140
	0.0100	9 913	56 384	676 608	9 913	570 028
	0.0050	4 945	27 941	335 292	4 945	282 868
	0.0010	990	5 606	67 272	990	56 728
	0.0005	495	2 844	34 128	495	28 692
	0.0001	99	602	7 224	99	6 004

Table C.1: Sizes of samples from dataset T1E1.D1E6

Sample type	Scale factor	Unnested layout			Nested layout	
		Transactions	Rows	Disk (bytes)	Rows	Disk (bytes)
original	1.0000	100 000	9 886 943	118 643 316	100 000	80 295 544
	0.5000	50 000	4 942 956	59 315 472	50 000	40 143 648
	0.1000	10 000	989 278	11 871 336	10 000	8 034 224
	0.0500	5 000	494 693	5 936 316	5 000	4 017 544
	0.0100	1 000	99 270	1 191 240	1 000	806 160
	0.0050	500	49 253	591 036	500	400 024
	0.0010	100	9 835	118 020	100	79 880
	0.0005	50	4 875	58 500	50	39 600
	0.0001	10	955	11 460	10	7 760
query	1.0000	95 353	1 254 606	15 055 272	95 353	11 181 084
	0.5000	47 657	625 226	7 502 712	47 657	5 573 692
	0.1000	9 472	124 453	1 493 436	9 472	1 109 288
	0.0500	4 788	64 544	774 528	4 788	573 808
	0.0100	957	13 004	156 048	957	115 516
	0.0050	479	6 084	73 008	479	54 420
	0.0010	97	1 300	15 600	97	11 564
	0.0005	47	544	6 528	47	4 916
	0.0001	9	19	228	9	260

Table C.2: Sizes of samples from dataset T1E2.D1E5

Sample type	Scale factor	Unnested layout			Nested layout	
		Transactions	Rows	Disk (bytes)	Rows	Disk (bytes)
original	1.0000	10 000	9 959 457	119 513 484	10 000	79 795 656
	0.5000	5 000	4 979 807	59 757 684	5 000	39 898 456
	0.1000	1 000	995 551	11 946 612	1 000	7 976 408
	0.0500	500	497 999	5 975 988	500	3 989 992
	0.0100	100	99 427	1 193 124	100	796 616
	0.0050	50	49 584	595 008	50	397 272
	0.0010	10	9 888	118 656	10	79 224
	0.0005	5	5 000	60 000	5	40 060
	0.0001	1	971	11 652	1	7 780
query	1.0000	9 546	1 020 893	12 250 716	9 546	8 281 696
	0.5000	4 778	510 325	6 123 900	4 778	4 139 936
	0.1000	948	87 402	1 048 824	948	710 592
	0.0500	470	61 096	733 152	470	494 408
	0.0100	95	7 197	86 364	95	58 716
	0.0050	48	5 151	61 812	48	41 784
	0.0010	9	34	408	9	380
	0.0005	5	17	204	5	196
	0.0001	1	1	12	1	20

Table C.3: Sizes of samples from dataset T1E3.D1E4

Sample type	Scale factor	Unnested layout			Nested layout	
		Transactions	Rows	Disk (bytes)	Rows	Disk (bytes)
original	1.0000	515 597	3 367 020	40 404 240	515 597	33 123 324
	0.5000	257 798	1 681 224	20 174 688	257 798	16 543 368
	0.1000	51 560	336 702	4 040 424	51 560	3 312 336
	0.0500	25 780	169 259	2 031 108	25 780	1 663 432
	0.0100	5 156	33 006	396 072	5 156	325 920
	0.0050	2 578	16 798	201 576	2 578	165 320
	0.0010	516	3 189	38 268	516	31 704
	0.0005	258	1 804	21 648	258	17 528
	0.0001	52	319	3 828	52	3 176

Table C.4: Sizes of samples from dataset BMS3

List of Algorithms

List of Figures

List of Tables

Bibliography

[AHV95] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[AIS93] Rakesh Agrawal, Tomasz Imielinski, and Arun Swami. Mining Association Rules between Sets of Items in Large Databases. In *Proceedings SIGMOD, Washington DC, USA*, pages 207–216, May 1993.

[AS94] Rakesh Agrawal and Ramakrishnan Srikant. Fast Algorithms for Mining Association Rules. In *Proceedings VLDB, Santiago, Chile*, pages 487–499, September 1994.

[Bay96] Rudolf Bayer. The Universal B-Tree for Multidimensional Indexing. Technical Report, Technische Universität München, Department of Computer Science, November 1996. 14 pages.

[Bay97] Rudolf Bayer. UB-Trees and UB-Cache: A New Processing Paradigm for Database Systems. Technical Report, Technische Universität München, Department of Computer Science, March 1997. 15 pages.

[BBD+01] Jochen Van den Bercken, Björn Blohsfeld, Jens-Peter Dittrich, Jürgen Krämer, Tobias Schäfer, Martin Schneider, and Bernhard Seeger. XXL—A Library Approach to Supporting Efficient Implementations of Advanced Database Queries. In *Proceedings VLDB, Rome, Italy*, pages 39–48, September 2001.

[BDPP97] Patrick Bosc, Didier Dubois, Olivier Pivert, and Henri Prade. Flexible Queries in Relational Databases—The Example of the Division Operator. *Theoretical Computer Science*, 171(1–2):281–302, 1997.

[BM72] Rudolf Bayer and Edward M. McCreight. Organization and Maintenance of Large Ordered Indices. *Acta Informatica*, 1(3):173–189, 1972.

[Bos97] Patrick Bosc. On the Primitivity of the Division of Fuzzy Relations. In *Proceedings SAC, San Jose, California, USA*, pages 197–201, February–March 1997.

[BP82] Bill P. Buckles and Frederick E. Petry. A Fuzzy Representation of Data for Relational Databases. *Fuzzy Sets and Systems*, 7(3), May 1982. 213–226.

[Bry89] François Bry. Towards an Efficient Evaluation of General Queries: Quantifier and Disjunction Processing Revisited. In *Proceedings SIGMOD, Portland, Oregon, USA*, pages 193–204, May–June 1989.

[Car86] John V. Carlis. HAS, a Relational Algebra Operator or Divide is not Enough to Conquer. In *Proceedings ICDE, Los Angeles, California, USA*, pages 254–261, February 1986.

[CD02] Jianjun Chen and David DeWitt. Dynamic Re-grouping of Continuous Queries. In *Proceedings VLDB, Hong Kong, China*, pages 430–441, August 2002.

[CDH+99] John Clear, Debbie Dunn, Brad Harvey, Michael L. Heytens, Peter Lohman, Abhay Mehta, Mark Melton, Lars Rohrberg, Ashok Savasere, Robert M. Wehrmeister, and Melody Xu. NonStop SQL/MX Primitives for Knowledge Discovery. In *Proceedings KDD, San Diego, California, USA*, pages 425–429, August 1999.

[CHK+03] Michael Cammert, Christoph Heinz, Jürgen Krämer, Martin Schneider, and Bernhard Seeger. A Status Report on XXL—A Software Infrastructure for Efficient Query Processing. *BTCDE*, 26(2):12–18, June 2003.

[CKMP97] Jens Claußen, Alfons Kemper, Guido Moerkotte, and Klaus Peithner. Optimizing Queries with Universal Quantification in Object-Oriented and Object-Relational Databases. In *Proceedings VLDB, Athens, Greece*, pages 286–295, August 1997.

[Cod70] Edgar Codd. A Relational Model for Large Shared Data Banks. *CACM*, 13(6):377–387, June 1970.

[Cod72] Edgar Codd. Relational Completeness of Database Sub-Languages. In Randall Rustin, editor, *Courant Computer Science Symposium 6: Database Systems*, pages 65–98. Prentice-Hall, 1972.

[Dat94] Chris Date. *An Introduction to Database Systems*. Addison-Wesley, sixth edition, 1994.

[Day83] Umeshwar Dayal. Queries with Quantifiers: A Horticultural Approach. In *Proceedings PODS, Atlanta, Georgia, USA*, pages 125–136, March 1983.

[Day87] Umeshwar Dayal. Of Nests and Trees: A Unified Approach to Processing Queries that Contain Nested Subqueries, Aggregates, and Quantifiers. In *Proceedings VLDB, Brighton, England*, pages 197–208, September 1987.

[DD92] Hugh Darwen and Chris Date. Into the Great Divide. In Chris Date and Hugh Darwen, editors, *Relational Database: Writings 1989–1991*, pages 155–168. Addison-Wesley, Reading, Massachusetts, USA, 1992.

[DD95] Hugh Darwen and Chris Date. The Third Manifesto. *SIGMOD Record*, 24(1):39–49, March 1995.

[Dem82] Robert Demolombe. Generalized Division for Relational Algebraic Language. *Information Processing Letters*, 14(4):174–178, 1982.

[Dep86] Uwe Deppisch. S-Tree: A Dynamic Balanced Signature Index for Office Retrieval. In *Proceedings SIGIR, Pisa, Italy*, pages 77–87, September 1986.

[FT82] Patrick C. Fischer and Stan J. Thomas. Operators for Non-First-Normal-Form Relations. In *Proceedings COMPSAC, New York, USA*, November 1982.

[GC95] Goetz Graefe and Richard Cole. Fast Algorithms for Universal Quantification in Large Databases. *TODS*, 20(2):187–236, 1995.

[GK98] Timothy Griffin and Bharat Kumar. Algebraic Change Propagation for Semijoin and Outerjoin Queries. *SIGMOD Record*, 27(3):22–27, March 1998.

[GMUW02] Hector Garcia-Molina, Jeffrey Ullman, and Jennifer Widom. *Database Systems—The Complete Book*. Prentice-Hall, 2002.

[GP99] Peter Gulutzan and Trudy Pelzer. *SQL-99 Complete, Really: An Example-Based Reference Manual of the New Standard*. R&D Books, Lawrence, Kansas, USA, 1999.

[Gra89] Goetz Graefe. Relational Division: Four Algorithms and Their Performance. In *Proceedings ICDE, Los Angeles, California, USA*, pages 94–101, February 1989.

[Gra93] Goetz Graefe. Query Evaluation Techniques for Large Databases. *ACM Computing Surveys*, 25(2):73–170, June 1993.

[Gra95] Goetz Graefe. The Cascades Framework for Query Optimization. *BTCDE*, 18(3):19–29, September 1995.

[HB99] Seth Hettich and Stephen Bay. The UCI KDD Archive, 1999. http://kdd.ics.uci.edu.

[Hel00] Sven Helmer. *Performance Enhancements for Advanced Database Management Systems*. PhD thesis, University of Mannheim, Germany, December 2000.

[HFKZ96] Jiawei Han, Yongjian Fu, Krzysztof Koperski, and Osmar Zaiane. DMQL: A Data Mining Query Language for Relational Databases. In *Proceedings DMKD, Montreal, Canada*, June 1996.

[HGG01] Jochen Hipp, Ulrich Günzer, and Udo Grimmer. Integrating Association Rule Mining Algorithms with Relational Database Systems. In *Proceedings ICEIS, Setubal, Portugal*, pages 130–137, July 2001.

[HIL00] Tok Wee Hyong, A. Indriyati, and Low Wai Lup. Towards Ad Hoc Mining of Association Rules with Database Management Systems. Research Report, School of Computing, National University of Singapore, October 2000.

[HK01] Jiawei Han and Micheline Kamber. *Data Mining: Concepts and Techniques*. Morgan Kaufmann Publishers, 2001.

[HKMT95] Marcel Holsheimer, Martin Kersten, Heikki Mannila, and Hannu Toivonen. A Perspective on Databases and Data Mining. In *Proceedings KDD, Montreal, Quebec, Canada*, pages 150–155, August 1995.

[HM97] Sven Helmer and Guido Moerkotte. Evaluation of Main Memory Join Algorithms for Joins with Set Comparison Join Predicates. In *Proceedings VLDB, Athens, Greece*, pages 386–395, August 1997.

[HM02] Sven Helmer and Guido Moerkotte. Compiling Away Set Containment and Intersection Joins. Technical Report, University of Mannheim, Germany, April 2002.

[HP95] Ping-Yu Hsu and D. Stott Parker. Improving SQL with Generalized Quantifiers. In *Proceedings ICDE, Taipei, Taiwan*, pages 298–305, March 1995.

[HPY00] Jiawei Han, Jian Pei, and Yiwen Yin. Mining Frequent Patterns without Candidate Generation. In *Proceedings SIGMOD, Dallas, Texas, USA*, pages 1–12, May 2000.

[HS95] Maurice Houtsma and Arun Swami. Set-oriented Data Mining in Relational Databases. *DKE*, 17(3):245–262, December 1995.

[ISO02] ISO/IEC. *Information Technology—Database Language—SQL—Part 2: Foundation (SQL/Foundation), Working Draft 9075-2:2003*, December 2002.

[IV99] Tomasz Imielinski and Aashu Virmani. MSQL: A Query Language for Database Mining. *DMKD*, 3(4):373–408, December 1999.

[JK83] Matthias Jarke and Jürgen Koch. Range Nesting: A Fast Method to Evaluate Quantified Queries. In *Proceedings SIGMOD, San Jose, California, USA*, pages 196–206, May 1983.

[JK84] Matthias Jarke and Jürgen Koch. Query Optimization in Database Systems. *ACM Computing Surveys*, 16(2):111–152, June 1984.

[JS82] G. Jaeschke and Hans-Jörg Schek. Remarks on the Algebra of Non First Normal Form Relations. In *Proceedings PODS, Los Angeles, California, USA*, pages 124–138, March 1982.

[KSRM03] Tobias Kraft, Holger Schwarz, Ralf Rantzau, and Bernhard Mitschang. Coarse-Grained Optimization: Techniques for Rewriting SQL Statement Sequences. In *VLDB, Berlin, Germany*, September 2003.

[Loh88] Guy M. Lohman. Grammar-like Functional Rules for Representing Query Optimization Alternatives. In *Proceedings SIGMOD, Chicago, Illinois, USA*, pages 18–27, June 1988.

[Mai83] David Maier. *The Theory of Relational Databases*. Computer Science Press, 1983.

[Mak77] Akifumi Makinouchi. A Consideration on Normal Form of Not-Necessarily-Normalized Relation in the Relational Data Model. In *Proceedings VLDB, Tokyo, Japan*, pages 447–453, October 1977.

[Mam03] Nikos Mamoulis. Efficient Processing of Joins on Set-valued Attributes. In *Proceedings SIGMOD, San Diego, California, USA*, June 2003.

[MC03] Pratyush Mishra and Sharma Chakravarthy. Performance Evaluation of SQL-OR Variants for Association Rule Mining. In *Proceedings DaWaK, Prague, Czech Republic*, September 2003.

[MGM02a] Sergey Melnik and Hector Garcia-Molina. Divide-and-Conquer Algorithm for Computing Set Containment Joins. In *Proceedings EDBT, Prague, Czech Republic*, pages 427–444, March 2002.

[MGM02b] Sergey Melnik and Hector Garcia-Molina. Divide-and-Conquer Algorithm for Computing Set Containment Joins. Extended Technical Report, Stanford University, California, USA, 2002.

[MGM03] Sergey Melnik and Hector Garcia-Molina. Adaptive Algorithms for Set Containment Joins. *TODS*, 28(1):56–99, March 2003.

[MHM03] Norman May, Sven Helmer, and Guido Moerkotte. Nested Queries and Quantifiers in an Ordered Context. Technical Report, Fakultät für Mathematik und Informatik, University of Mannheim, Germany, February 2003.

[Mis02] Pratyush Mishra. Performance Evaluation and Analysis of SQL Based Approaches for Association Rule Mining. Master's thesis, University of Texas at Arlington, Texas, USA, December 2002.

[MPC96] Rosa Meo, Guiseppe Psaila, and Stefano Ceri. A New SQL-like Operator for Mining Association Rules. In *Proceedings VLDB, Bombay, India*, pages 122–133, September 1996.

[MTV94] Heikki Mannila, Hannu Toivonen, and A. Inkeri Verkamo. Efficient Algorithms for Discovering Association Rules. In *AAAI Workshop on Knowledge and Discovery in Databases, Seattle, Washington, USA*, pages 181–192, July 1994.

[MZB99] Volker Markl, Martin Zirkel, and Rudolf Bayer. Processing Operations with Restrictions in RDBMS without External Sorting: The Tetris Algorithm. In *Proceedings ICDE, Sydney, Australia*, pages 562–571, March 1999.

[NCFB01] Amir Netz, Surajit Chaudhuri, Usuma Fayyad, and J. Bernhardt. Integrating Data Mining with SQL Databases: OLE DB for Data Mining. In *Proceedings ICDE, Heidelberg, Germany*, pages 379–387, March 2001.

[NM02] Alexandros Nanopoulos and Yannis Manolopoulos. Efficient Similarity Search for Market Basket Data. *VLDB Journal*, 11(2):138–152, 2002.

[NRM00] Clara Nippl, Ralf Rantzau, and Bernhard Mitschang. StreamJoin: A Generic Database Approach to Support the Class of Stream-Oriented Applications. In *Proceedings IDEAS, Yokohama, Japan*, pages 83–91, September 2000.

[ÖÖM87] Gultekin Özsoyoğlu, Z. Meral Özsoyoğlu, and Victor Matos. Extending Relational Algebra and Relational Calculus with Set-Valued Attributes and Aggregate Functions. *TODS*, 12(4):566–592, December 1987.

[PHH92] Hamid Pirahesh, Joseph M. Hellerstein, and Waqar Hasan. Extensible/Rule Based Query Rewrite Optimization in Starburst. In *Proceedings SIGMOD, San Diego, California, USA*, pages 39–48, June 1992.

[PS03] Sriram Pemmaraju and Steven Skiena. *Computational Discrete Mathematics: Combinatorics and Graph Theory with Mathematica*. Cambridge University Press, 2003.

[PSTK99] Iko Pramudiono, Takahiko Shintani, Takayuki Tamura, and Masaru Kitsuregawa. Parallel SQL Based Association Rule Mining on Large Scale PC Cluster: Performance Comparison with Directly Coded C Implementation. In *Proceedings PAKDD, Beijing, China*, pages 94–98, April 1999.

[Ram02] Karthikeyan Ramasamy. *Efficient Storage and Query Processing of Set-valued Attributes*. PhD thesis, University of Wisconsin, Madison, Wisconsin, USA, 2002. 144 pages.

[Ran02] Ralf Rantzau. Frequent Itemset Discovery with SQL Using Universal Quantification. In *Proceedings DTDM, Prague, Czech Republic*, pages 51–66, March 2002.

[Ran03] Ralf Rantzau. Processing Frequent Itemset Discovery Queries by Division and Set Containment Join Operators. In *Proceedings DMKD, San Diego, California, USA*, June 2003.

[RBG96] Sudhir Rao, Antonio Badia, and Dirk van Gucht. Providing Better Support for a Class of Decision Support Queries. In *Proceedings SIGMOD, Montreal, Canada*, pages 217–227, June 1996.

[RCIC99] Karthik Rajamani, Alan L. Cox, Balakrishna R. Iyer, and Atul Chadha. Efficient Mining for Association Rules with Relational Database Systems. In *Proceedings IDEAS, Montreal, Canada*, pages 148–155, August 1999.

[RG00] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. McGraw-Hill, second edition, 2000.

[RPNK00] Karthikeyan Ramasamy, Jignesh M. Patel, Jeffrey F. Naughton, and Raghav Kaushik. Set Containment Joins: The Good, The Bad and The Ugly. In *Proceedings VLDB, Cairo, Egypt*, pages 351–362, September 2000.

[RS99] Ralf Rantzau and Holger Schwarz. A Multi-Tier Architecture for High-Performance Data Mining. In *Proceedings BTW, Freiburg, Germany*, pages 151–163, March 1999.

[RSMW02] Ralf Rantzau, Leonard Shapiro, Bernhard Mitschang, and Quan Wang. Universal Quantification in Relational Databases: A Classification of Data and Algorithms. In *Proceedings EDBT, Prague, Czech Republic*, pages 445–463, March 2002.

[RSMW03] Ralf Rantzau, Leonard Shapiro, Bernhard Mitschang, and Quan Wang. Algorithms and Applications for Universal Quantification in Relational Databases. *Information Systems*, 28(1):3–32, January 2003.

[SC75] John Miles Smith and Philip Yen-Tang Chang. Optimizing the Performance of a Relational Algebra Data Base Interface. *CACM*, 18(10):568–579, October 1975.

[SDS02] SDSU Java Library, 2002.

[SKS01] Abraham Silberschatz, Henry F. Korth, and S. Sudarshan. *Database System Concepts*. McGraw-Hill, fourth edition, 2001.

[STA98] Sunita Sarawagi, Shiby Thomas, and Rakesh Agrawal. Integrating Association Rule Mining with Relational Database Systems: Alternatives and Implications. Research Report RJ 10107 (91923), IBM Almaden Research Center, San Jose, California, USA, March 1998.

[TBM02] Eleni Tousidou, Panayiotis Bozanis, and Yannis Manolopoulos. Signature-based Structures for Objects with Set-valued Attributes. *Information Systems Journal, Elsevier*, 27(2):93–121, April 2002.

[TC99] Shiby Thomas and Sharma Chakravarthy. Performance Evaluation and Optimization of Join Queries for Association Rule Mining. In *Proceedings DaWaK, Florence, Italy*, pages 241–250, August–September 1999.

[Tra02] Transaction Processing Council. *TPC Benchmark H (Decision Support), Standard Specification, Revision 2.1.0*, 2002.

[W3C03] W3C. XQuery 1.0: An XML Query Language. W3C Working Draft 02 May 2003, W3C, May 2003.

[Win02] Marianne Winslett. Hector Garcia-Molina Speaks Out. *SIGMOD Record*, 31(3):47–54, September 2002.

[Win03a] Marianne Winslett. Jim Gray Speaks Out. *SIGMOD Record*, 32(1):54–62, March 2003.

[Win03b] Marianne Winslett. Rakesh Agrawal Speaks Out. *SIGMOD Record*, 32(3):83–90, September 2003.

[WZ02] Haixun Wang and Carlo Zaniolo. ATLaS: A Native Extension of SQL for Data Mining and Stream Computations. Technical Report, Computer Science Department, UCLA, California, USA, 2002.

[Yag91] Ronald R. Yager. Fuzzy Quotient Operators for Fuzzy Relational Databases. In *Proceedings IFES, Yokohama, Japan*, pages 289–296, November 1991.

[YPK00] Takeshi Yoshizawa, Iko Pramudiono, and Masaru Kitsuregawa. SQL Based Association Rule Mining Using Commercial RDBMS (IBM DB2 UDB EEE). In *Proceedings DaWaK, London, UK*, pages 301–306, September 2000.

[Zan02] Carlo Zaniolo. Extending SQL for Decision Support Applications. In *Presentation slides of keynote address at DMDW, Toronto, Ontario, Canada*, May 2002.

[ZKM01] Zijian Zheng, Ron Kohavi, and Llew Mason. Real World Performance of Association Rule Algorithms. In *Proceedings SIGKDD, San Francisco, California, USA*, pages 401–406, August 2001.

[ZND+01] Chun Zhang, Jeffrey Naughton, David DeWitt, Qiong Luo, and Guy Lohman. On Supporting Containment Queries in Relational Database Management Systems. In *Proceedings SIGMOD, Santa Barbara, California, USA*, May 2001.

Bibliographic Abbreviations

ACM	Association for Computing Machinery
BTCDE	Bulletin of the Technical Committee on Data Engineering
BTW	Conference Datenbanksysteme für Business, Technologie und Web (formerly Conference Datenbanksysteme in Büro, Technik und Wissenschaft)
CACM	Communications of the ACM
CAISE	International Conference on Advanced Information Systems Engineering
COMPSAC	IEEE International Conference on Computer Software and Applications
DaWaK	International Conference on Data Warehousing and Knowledge Discovery
DKE	Data and Knowledge Engineering
DMDW	International Workshop on Design and Management of Data Warehouses
DMKD	International Journal on Data Mining and Knowledge Discovery
DMKD	SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery
DTDM	International Workshop on Database Technologies for Data Mining
EDBT	International Conference on Extending Database Technology
ICDE	International Conference on Data Engineering
IDEAS	International Database Engineering and Applications Symposium
IEEE	Institute of Electrical and Electronics Engineers
IFES	International Fuzzy Engineering Symposium
KDD	ACM SIGKDD International Conference on Knowledge Discovery and Data Mining
PAKDD	Pacific-Asia Conference on Knowledge Discovery and Data Mining
PODS	ACM Symposium on Principals of Database Systems
SAC	ACM Symposium on Applied Computing
SIGIR	ACM SIGIR Conference on Research and Development in Information Retrieval

SIGKDD	ACM Special Interest Group on Knowledge Discovery and Data Mining
SIGMOD	ACM Special Interest Group on Management of Data, SIGMOD International Conference on Management of Data
TODS	ACM Transactions on Database Systems
VLDB	International Conference on Very Large Databases