PERGAMON

Information Systems 28 (2003) 3–32

www.elsevier.com/locate/infosys

Algorithms and applications for universal quantification in relational databases

Ralf Rantzau[a,*], Leonard D. Shapiro[b], Bernhard Mitschang[a], Quan Wang[c]

[a] *Computer Science Department, University of Stuttgart, Breitwiesenstr. 20-22, 70565 Stuttgart, Germany*
[b] *Computer Science Department, Portland State University, P.O. Box 751, Portland, OR 97201-0751, USA*
[c] *Oracle Corporation, Portland, OR, USA*

Abstract

Queries containing universal quantification are used in many applications, including business intelligence applications and in particular data mining. We present a comprehensive survey of the structure and performance of algorithms for universal quantification. We introduce a framework that results in a complete classification of input data for universal quantification. 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.

In addition to the classification, we show the relationship between relational division and the set containment join and we illustrate the usefulness of employing universal quantifications by presenting a novel approach for frequent itemset discovery.

© 2002 Elsevier Science Ltd. All rights reserved.

Keywords: Query operators; Grouping; Relational division; Set containment join; Frequent itemset discovery

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 the formula f, and we write $\forall x : f(x)$ in calculus.

In relational databases, universal quantification is implemented by the division operator (represented by $\div$) of the relational algebra. 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 paper, we will focus on the division operator exclusively.

Several algorithms have been proposed to implement relational division efficiently. These algorithms are presented in an isolated manner in

*Corresponding author. Tel.: +49-711-7816-433; fax: +49-711-7816-424.
E-mail address: rantzau@informatik.uni-stuttgart.de (R. Rantzau).

0306-4379/02/$ - see front matter © 2002 Elsevier Science Ltd. All rights reserved.
PII: S 0 3 0 6 - 4 3 7 9 (0 2) 0 0 0 4 7 - 9

the research literature—typically, no relationships are shown between them. Furthermore, each of these algorithms claims to be superior to others, but in fact each algorithm has optimal performance only for certain types of input data.

1.1. The division operator

To illustrate the division operator, we will use a simple example throughout the paper, illustrated in Fig. 1, representing data from a CS department at a university [1]. A *course* row represents a course that has been offered by the department and an *enrollment* row indicates that a student has taken a particular course. The following query can be represented by the division operator:

> Which students have taken *all* courses offered by the department?

As indicated in the table *result*, only Bob has taken all the courses. Bob is enrolled in another course (Graphics) but this does not affect the result. Both Alice and Chris are not enrolled in the Databases course. Therefore, they are not included in the result.

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

enrollment

student_id	*course_id*
Alice	Compilers
Alice	Theory
Bob	Compilers
Bob	Databases
Bob	Graphics
Bob	Theory
Chris	Compilers
Chris	Graphics
Chris	Theory

(a) Dividend

course

course_id
Compilers
Databases
Theory

(b) Divisor

result

student_id
Bob

(c) Quotient

Fig. 1. *enrollment* ÷ *course* = *result*, representing the query "Which students have taken all courses?"

In the example of Fig. 1, the divisor and quotient have only one attribute each, but in general, they may have an arbitrary number of attributes. 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.

1.2. Outline of the paper

The remainder of this paper is organized as follows. In Section 2, we present a classification of input data for algorithms that evaluate division within queries. Section 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 4, we evaluate the algorithms according to both applicability and effectiveness for different kinds of input data, based on a performance analysis. In Section 5, we discuss the relationship between relational division and the set containment join. Section 6 illustrates a new approach to exploit division and set containment join to discover frequent itemsets. Section 7 gives an overview of related work. Section 8 concludes the paper and comments on future work.

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 attributes. For some of these classes, we will present efficient algorithms in Section 3 that exploit the specific data properties of a class.

2.1. Input data characteristics

The goal of this paper 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 [1–3]. 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 Fig. 1 but is of much larger size, and suppose:

- *enrollment* is sorted by *student_id* and *course_id* and resides on disk, and
- *course* is sorted by *course_id* and resides in memory.

Then the example query can be executed with one scan of the *enrollment* table. This is accomplished by reading the *enrollment* table from disk. As each student appears, the *course_id* values associated with that student are merged with the course table. If all courses match, the *student_id* is copied to the result.

The single scan of the *enrollment* table is obviously the most efficient possible algorithm in this case. In the remainder of this paper, 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 Fig. 1, how could this careful structuring of input data, such as sorting by *student_id* and *course_id*, occur? It could happen by chance, or for two other more commonly encountered reasons:

1. The data might be stored in tables, which were sorted in that order for other purposes, for example, so that it is easy to list enrollments on a roster in ID order, or to find course information when a course ID number is given.
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.

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 Fig. 1? Several algorithms are applicable but they are not equally efficient. For example, since the dividend and divisor are both sorted on the attribute *course_id* in Fig. 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 *course_id* for each group of *student_id*? 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 3.3.4.

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 sort algorithms.

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 Fig. 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 Fig. 1 could be the input of an aggregation that counts the number of students. The meaning of the resulting aggregate is the number of students who have taken all courses of the department. Alternatively, the result in Fig. 1 could be an input of a join with a table $student(student\_id, name, address, \ldots)$ to retrieve a student'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.

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 course the number of enrolled students in the *enrollment* table of Fig. 1. One way to compute the aggregates involves *grouping*: after the table has been grouped on *course_id*, all rows of the table with the same value of *course_id* 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 course ID as the first group value and then search through the whole table to find the rows that match this ID and compute the sum. Then, we pick a second course ID, 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 attribute, the result is equal to grouping on a single attribute 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 attributes, we have to be aware of the *attribute 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 attributes with all of the divisor's attributes.

Sorted data appears frequently in query processing. Note that sorting is a special grouping operation. For example, grouping only requires that students enrolled in the same course 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. Though sort-based grouping algorithms do more than necessary, both hash and sort-based grouping perform well for large datasets [1,4].

2.4. Grouped input data for division

Relational division has two input tables, a dividend and a divisor, and it returns a quotient table. As a consequence of the definition of the division operator, we can partition the attributes of the dividend S into two sets, which we denote D and Q, because they correspond to the attributes of the divisor and the quotient, respectively. The divisor's attributes correspond to D, i.e., for each attribute in the divisor there is a different attribute in D of the same domain. As already mentioned, for simplicity, we assume that the names of attributes in the quotient R are the same as the corresponding attribute names in the dividend S and the divisor T. Thus, we write a division operation as $R(Q) = S(Q \cup D) \div T(D)$. In Fig. 1, $Q = \{student\_id\}$ and $D = \{course\_id\}$.

Our classification of division algorithms is based on whether certain attributes 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 can retrieve the rows in sorted order, specified by the index attribute list. Thus, in some situations 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 1, we show all possible classes of input data based on whether or not interesting attribute sets are grouped, i.e., grouped on one of Q, D, or the divisor. As we will see later in this paper, 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 attribute 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 attributes Q (denoted by G_1, the major group) and for each group, it is grouped on the divisor D (denoted by G_2, the minor group). The divisor is grouped in the same ordering (G_2) 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

Table 1
A classification of dividend and divisor

Class	Dividend		Divisor	Description of grouping
	Q	D		
0	N	N	N	
1	N	N	G	
2	N	G	N	
3	N	G_1	G_2	Arbitrary ordering of groups in D and divisor
4	N	G_1	G_1	Same ordering of groups in D and divisor
5	G	N	N	
6	G	N	G	
7	G_1	G_2	N	Q major, D minor
8	G_2	G_1	N	D major, Q minor
9	G_1	G_2	G_3	Q major, D minor; arbitrary ordering of groups in D and divisor
10	G_1	G_2	G_2	Q major, D minor; same ordering of groups in D and divisor
11	G_2	G_1	G_3	D major, Q minor; arbitrary ordering of groups in D and divisor
12	G_2	G_1	G_1	D major, Q minor; same ordering of groups in D and divisor

Note: Attributes are either grouped (G) or not grouped (N). We use the same (a different) subscript of G when D and the divisor have the same (a different) ordering of groups in classes 3, 4, 9–12. In addition, when the dividend is grouped on both Q and D in classes 7–12, then G_1 (G_2) denotes the attributes that the table is grouped on first (second).

property in Table 1, but the algorithms in Section 3 will refer to this distinction. We do not consider any data property other than grouping in this paper because our approach is complete and can easily and effectively be exploited by a query optimizer and query processor.

Fig. 2 illustrates four classes of input data for division, based on the example data of Fig. 1. These classes, which are shown in bold font in Table 1, are important for several algorithms that we present in the following section. Note that for class 10 both tables are grouped in the same order on *course_id*. If the value "Graphics" is present in a quotient group then it always appears after "Theory" and before "Compilers." Fig. 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 and 3.3.3.

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 Fig. 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 Fig. 3. If a cell of the matrix is marked with "Y" (yes), 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 D, Q, or divisor. 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.

enrollment

student_id	*course_id*
Bob	Theory
Alice	Compilers
Chris	Theory
Chris	Graphics
Alice	Theory
Bob	Graphics
Chris	Compilers
Bob	Databases
Bob	Compilers
not grouped	**not grouped**

course

course_id
Databases
Theory
Compilers
not grouped

(a) Class 0

enrollment

student_id	*course_id*
Alice	Theory
Chris	Theory
Bob	Theory
Bob	Databases
Bob	Graphics
Chris	Graphics
Bob	Compilers
Chris	Compilers
Alice	Compilers
not grouped	**grouped**

course

course_id
Databases
Theory
Compilers
not grouped

(b) Class 2

enrollment

student_id	*course_id*
Chris	Graphics
Chris	Compilers
Chris	Theory
Alice	Theory
Alice	Compilers
Bob	Theory
Bob	Compilers
Bob	Databases
Bob	Graphics
grouped	**not grouped**

course

course_id
Databases
Theory
Compilers
not grouped

(c) Class 5

enrollment

student_id	*course_id*
Chris	Theory
Chris	Graphics
Chris	Compilers
Alice	Theory
Alice	Compilers
Bob	Databases
Bob	Theory
Bob	Graphics
Bob	Compilers
grouped	**grouped**

course

course_id
Databases
Theory
Compilers
grouped

(d) Class 10

Fig. 2. Four important classes of input data, based on the example of Fig. 1.

Class C_2

Class C_1	12	11	10	9	8	7	6	5	4	3	2	1
0	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
1	Y	Y	Y	Y			Y		Y	Y		
2	Y	Y			Y				Y	Y		
3	Y	Y							Y			
4	Y											
5			Y	Y		Y	Y					
6			Y	Y								
7			Y	Y								
8	Y	Y										
9			Y									
10												
11	Y											

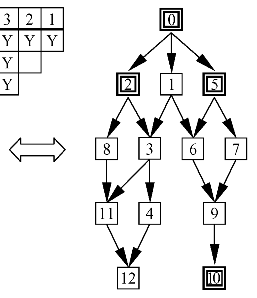


Fig. 3. A matrix and a directed acyclic graph representing the input data classification described in Table 1. All algorithms to be discussed in Section 3 assume data properties of either class 0, 2, 5, or 10.

For the subsequent discussion of division algorithms, we define two terms to refer to certain row subsets of the dividend. Let the dividend S be grouped on Q (D) as the first or the only set of group attributes, i.e., let the dividend belong to class 5 (2) and all its descendants in Fig. 3. Furthermore, let v be one specific value of such a group. Then, the set of rows defined by $\sigma_{Q=v}(S)$ ($\sigma_{D=v}(S)$) is called the *quotient group* (*divisor group*) of v. For example, in the *enrollment* table of class 5 in Fig. 2(c), the quotient group of Alice consists of the rows {(Alice, Theory), (Alice, Compilers)}. Similarly, the divisor group of Databases in class 2 in Fig. 2(b) consists of the single row (Bob, Databases).

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 2.

In Section 4, we will analyze and compare the effectiveness of each algorithm with respect to the data classification of Section 2.

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.

Table 2
Abbreviations for division algorithms

Division algorithm	Abbrev.
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-loops division	NLD
Nested-loops 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

The following algorithms assume that the division's input consists of a dividend table $S(Q, D)$ and a divisor table $T(D)$, where Q is a set of quotient attributes and D is the set of divisor attributes, as defined in Section 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) of the dividend $|S|$ and the size of the divisor $|T|$. We use $|Q|$, the number of distinct values of quotient attributes Q in the dividend, for some algorithms to derive a complexity formula. Note that always $|Q| \leqslant |S|$, and in the *worst* case $|Q| = |S|$, i.e., each single row of S is a potential (candidate) quotient. To derive formulas for the *typical* time and memory complexities, we use the assumption that $|S| \gg |T|$, i.e., there are many quotient candidates and/or the number of rows of an 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-world 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 [8]. However, since the focus of this paper 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.

3.2. Query language representation and algorithm classification

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.

The commonly used approach to express universal quantification uses two "NOT EXISTS" clauses, exploiting the mathematical equivalence $\forall x \exists y : f(x, y) \equiv \neg\exists x \neg\exists y : f(x, y)$ as follows:

```
SELECT DISTINCT student_id
FROM   enrollment AS e1
WHERE  NOT EXISTS (
  SELECT *
  FROM   course AS c
  WHERE  NOT EXISTS (
    SELECT *
    FROM   enrollment AS e2
    WHERE  e2.student_id = e1.student_id AND
           e2.course_id = c.course_id))
```

This query asks for each student, where there is no course that the student is not enrolled in.

The previous 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* [5], was planned to be included in the SQL:1999 standard [6] but it was finally excluded for reasons unknown to the authors. We can 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 student_id
FROM   enrollment AS e1
WHERE  FOR ALL (SELECT *
                FROM   course AS c)
       (EXISTS (
  SELECT *
  FROM   enrollment AS e2
  WHERE  e2.student_id = e1.student_id AND
         e2.course_id = c.course_id))
```

This query asks for each student, where for all courses there is an enrollment of this student.

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 non-trivial 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.

There is a third way mentioned in the literature that uses aggregation. The example query of Section 1.1 can be phrased in SQL using aggregation as follows:

```
SELECT   student_id
FROM     enrollment
GROUP BY student_id
HAVING   COUNT(DISTINCT course_id) = (
  SELECT COUNT(DISTINCT course_id)
  FROM   course)
```

Any query involving universal quantification can be replaced by a query that makes use of counting [1]. However, there is a problem with this approach to express division 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 *course_id* (*D*) value in *enrollment* (the dividend) is also contained in the *course* table (the divisor). Defining a foreign key *enrollment.course_id* that references *course* 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 all dividend rows whose *D* values are contained in the divisor. Fig. 4 illustrates the semi-join for our university example in Fig. 1.

The second condition of this approach requires that the *course_id* (*D*) 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 *course_id* 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 *Q* and *D*, as in the classes 7–12 in Table 1.

We have seen that the two approaches 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 *D* 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 [2], 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 *old_enrollment*

student_id	*course_id*
Chris	Compilers
Chris	Graphics
Chris	Theory
Alice	Compilers
Alice	Theory
Bob	Compilers
Bob	Databases
Bob	Graphics
Bob	Theory

(a) Original dividend

course

course_id
Databases
Compilers
Theory

(b) Divisor

new_enrollment

student_id	*course_id*
Chris	Compilers
Chris	Theory
Alice	Compilers
Alice	Theory
Bob	Compilers
Bob	Databases
Bob	Theory

(c) Resulting dividend

Fig. 4. Semi-join *old_enrollment* $\ltimes$ *course* = *new_enrollment*, representing the preprocessing of the *enrollment* table for aggregate division algorithms, based on the example in Fig. 1.

because one has to take care of duplicates, NULL values, and referential integrity, as already mentioned before.

Some query languages for non-relational data models also offer support to express quantification. For example, there is “work in progress” by the W3C on the Working Draft of *XQuery* [7], a query language for XML data. Universal quantification can be expressed in XQuery by an *every expression*.

3.3. Scalar algorithms

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

3.3.1. Nested-loops division

This algorithm is the most naïve way to implement division. However, like nested-loops join, an operator using *nested-loops 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 Fig. 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 (Q) 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 the most efficient algorithm for small ungrouped datasets.

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 $O(|S|^2 + |T|)$ and the typical time complexity is $O(|S|^2)$. The memory complexity is $O(|Q| + |T|)$. Since in the extreme case $|Q| = |S|$, the worst case memory complexity is $O(|S| + |T|)$ and the typical memory complexity is $O(|S|)$.

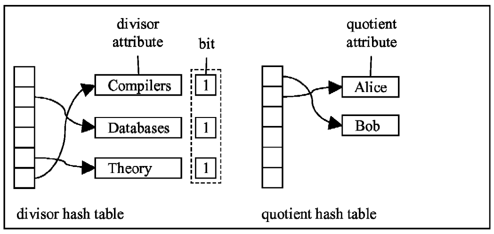
The pseudo-code of the nested-loops division algorithm is shown in the appendix. In that code, the *seen_divisors* and *seen_quotients* data structures are represented by the divisor hash table *dht* and the quotient hash table *qht*, respectively.

Fig. 5(g) 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 Alice and Bob (in this order) have been processed and we have not yet started to process any rows of Chris in the outer loop. We find that Bob is a quotient because all bits in the divisor hash table are equal to 1.

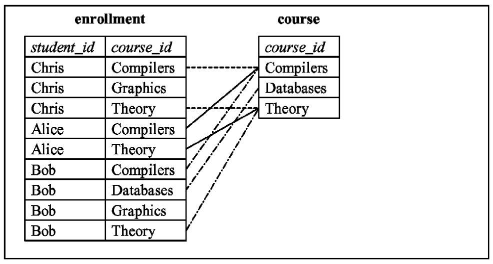
3.3.2. Merge-sort division

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

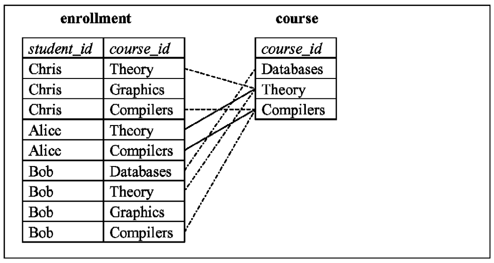
- the divisor T is sorted, and that
- the dividend S is grouped on Q, and for each group, it is sorted on D in the same order (ascending or descending) as T.



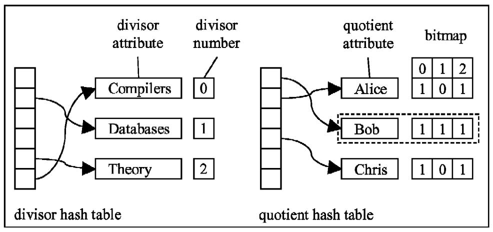
(a) Nested-Loops Division (NLD)



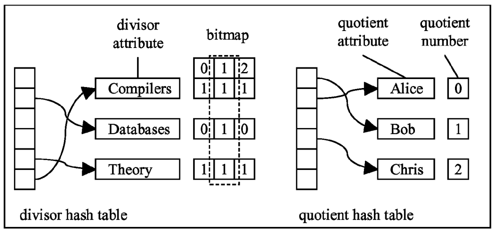
(b) Merge-Sort Division (MSD)



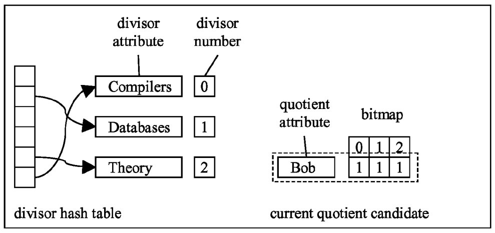
(c) Merge-Group Division (MGD)



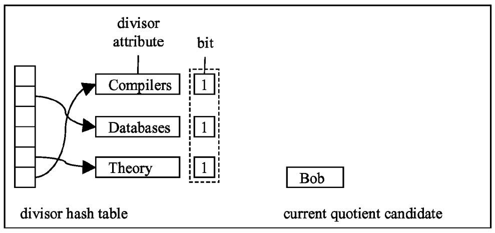
(d) Hash-Division (HD)



(e) Transposed Hash-Division (HDT)



(f) Hash-Division for Quotient Groups (HDQ)



(g) Transposed Hash-Division for Divisor Groups (HDTQ)

Fig. 5. Overview of the data structures and processing used in scalar algorithms. The value setting is based on the example from Fig. 1. Except for MSD and MGD, broken lined boxes indicate that a quotient is found.

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

The algorithm resembles merge-join for processing a single quotient group and is similar to nested-loops 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. If the divisor value D of the current dividend row and the divisor row match, we proceed with the next row in both tables. If D is greater than the current divisor row, we scan forward to the next quotient group. If D 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 MSD is similar to the approach called *naïve division*, presented in [1] and originating from [8]. 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 Q 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 MSD without duplicate removal logic in the appendix.

The worst case time complexity of this algorithm is $O(|S| + |Q||T|) = O(|S| + |S||T|) = O(|S||T|)$ 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 $O(|S||T|)$. 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.

Fig. 5(b) illustrates the matches between rows of dividend and divisor. Observe that the data is not sorted but only grouped on *student_id* in an arbitrary order.

3.3.3. Merge-group division

We can generalize MSD 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 attributes, but that
- the order of groups in each quotient group and the order of groups in the divisor are the same.

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 Q 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 attributes in MGD, we cannot skip a group on unequal values, as we do in MSD. For example, suppose that the divisor S has a single integer attribute and consists of the following rows in the given order: $S = (3, 1, 5)$ and the D values of the current quotient group G consists of the rows $G = (2, 5, 4, 6)$. We can be sure that G is *not* a valid quotient only after

- we have scanned the entire group G, where we find that the first element of S (3) is not contained in G, or
- we have scanned S 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 S (3 and 1) before element 5 appears.

The MGD approach makes use of a look-ahead of n divisor rows for some predefined value $n \geqslant 1$. 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 1 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. Fig. 5(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 $O(|S| + |Q||T|)$ 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 $O(|S| + |S||T|) = O(|S||T|)$. The typical case time complexity is also $O(|S||T|)$. The worst case memory complexity is $O(|T|)$ if we keep the entire divisor as a look-ahead in memory. The typical case memory complexity then becomes $O(1)$ since $|T| \ll |S|$.

3.3.4. *Classic hash-division*

In this section, we present the classic *hash-division* (HD) algorithm [1]. We call this algorithm "classic" to distinguish it from our variations of this approach in the following sections.

The two central data structures of HD are the divisor and quotient hash tables, sketched in Fig. 5(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 the appendix.

In a first phase, hash-division builds the divisor hash table while scanning the divisor. The hash function takes the divisor attributes 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 D 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 1 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 1, 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 Fig. 5(d), the contents of the hash tables are shown for the time when all dividend and divisor rows of Fig. 1 have been processed. We see that since Bob's bitmap contains no zeroes, Bob 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 $O(|S|+|T|)$ and $O(|S|)$, respectively. The memory complexity consists of $O(|T|)$ to store the divisor hash table plus $O(|Q||S|)$ for the quotient hash table. The size of a bitmap is proportional to $|S|$. Since the worst case scenario implies that $|Q|=|S|$, the total worst and typical case memory complexity is $O(|S||T|)$.

3.3.5. *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 1 at a certain bit position of a bitmap indicates which quotient candidate has the same values of D 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 D 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 at the position given by the quotient number to 1.

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 contained 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 1 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 the appendix.

Figs. 5(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 *enrollment* rows of Fig. 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.

3.3.6. *Hash-division for quotient groups*

Both, classic and transposed hash-division can be improved if the dividend is grouped on either D or Q. 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 3.4.3.

Let us first focus on classic hash-division. If the dividend is grouped on Q, 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 1, 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 Q. 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. In this paper, we do not elaborate on this variation of HD.

3.3.7. 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 Q, 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 attributes, 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 attribute 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 1. 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 Figs. 5(f) and (g) for the time when the group of dividend rows containing the quotient candidate Bob have been processed.

3.4. Aggregate algorithms

This class of algorithms compares the number of rows in each quotient candidate with the number of divisor rows. In case of equality, a quotient candidate becomes a quotient. All algorithms 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 Q or D.

3.4.1. Nested-loops counting division

Similar to scalar nested-loops division, *nested-loops 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.

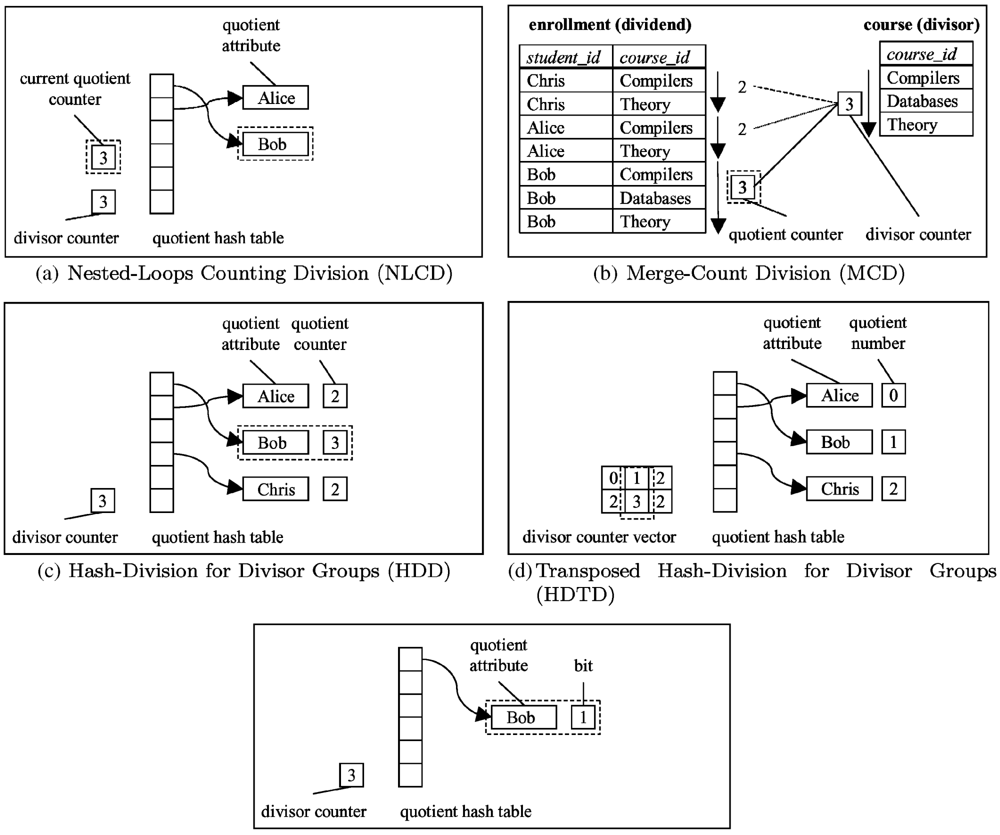
We have to keep track of which quotient candidates we have already checked, using a quotient hash table as shown in Fig. 6(a). A global counter is used to keep track of the number of dividend rows belonging to the same quotient candidate. We fully scan the dividend in an outer loop: We pick the first dividend row, insert its Q value into the quotient hash table, and set the counter to 1. 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 Q 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.

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

3.4.2. Merge-count division

Assuming that the dividend is grouped on Q, *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 and 3.3.3. Instead of comparing the elements of quotient groups with the divisor, MCD uses a representative (the row



(a) Nested-Loops 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)

Fig. 6. Overview of data structures used in aggregate algorithms. Broken lined boxes indicate that a quotient is found. Only Bob's group has as many dividend rows as the divisor.

count) of each quotient group to compare it with the divisor's aggregate. Fig. 6(b) illustrates the single scan required to compare the size of the each quotient group with the divisor size.

MCD has a worst case time complexity of $O(|S| + |T|)$ and an typical case time complexity of $O(|S|)$. 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 $O(1)$.

3.4.3. Hash-division for divisor groups

In Section 3.3.6, we have analyzed optimizations of hash-division that require a dividend that is grouped on Q. We now show some optimizations of hash-division for a dividend that is grouped on D. Unlike the hash-division-like algorithms based on quotient groups, the following two algorithms are blocking.

This 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 1. 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.

3.4.4. *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 Figs. 6(c) and (d) for the time when the entire dividend has been processed. Note that the dividend contains only three divisor groups (no Graphics rows), because we require that referential integrity between *enrollment* and *course* is preserved, e.g., by applying a semi-join of the two tables before division, as in Fig. 4. Bob is the only student 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 $O(|S| + |T|)$ and $O(|S|)$, respectively. The worst and typical case memory complexity is $O(|S|)$.

3.4.5. *Stream-join division*

The new algorithm *stream-join division* (SJD) [9] is an improvement of hash-division for divisor groups (HDD). As all other algorithms assuming a dividend that is grouped on D as the only or the major set of group attributes, 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 paper, 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., Q 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 1. Otherwise, i.e., when the Q 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 1 are added to the output.

Fig. 6(e) illustrates the use of the quotient hash table in SJD. We assume that the dividend is equal to the *enrollment* table of class 2 in Fig. 2(b) with the exception that the Graphics group {(Bob, Graphics), (Chris, Graphics)} is missing, due to referential integrity. We show the contents of the hash table for the time when the entire *enrollment* table has been processed. We see that Chris and Alice are not contained in the hash table because both have already been eliminated after the second group (Databases). Only Bob's bit is set to 1 and he is a quotient row because the number of groups (3, without Graphics) 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 [9]) with each other on the attributes Q.

4. Evaluation of algorithms

In this section, we briefly compare the division algorithms discussed in Section 3 with each other and show which algorithm is optimal, with respect to time and memory complexities, for each class of input data discussed in Section 2.

Table 3 characterizes the algorithms presented so far and shows the time and memory complexities involved. 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 Fig. 3. The overview of division algorithms in Table 3 shows that, despite the detailed classification in Table 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 D,
- class 5, which covers dividends that are grouped only or first on Q, and finally
- class 10, which specializes class 5 (and class 0, of course) by requiring that for each quotient group, the rows of D and the divisor appear in the same order. Hence, the dividend is grouped on Q as major and D as minor.

Table 3
Overview of division algorithms showing for each algorithm the class of required input data, its algorithm class, and its time and memory complexities

Division Algorithm	Algorithm class	Data class	Dividend S		Divisor T	Complexity in O-notation			
			Q	D			Time		Memory
						Worst	Typical	Worst	Typical
NLCD	Aggregate	0	N	N	N	$\lvert S\rvert^2+\lvert T\rvert$	$\lvert S\rvert^2$	1	1
NLD	Scalar					$\lvert S\rvert^2+\lvert T\rvert$	$\lvert S\rvert^2$	$\lvert S\rvert+\lvert T\rvert$	$\lvert S\rvert$
HD	Scalar					$\lvert S\rvert+\lvert T\rvert$	$\lvert S\rvert$	$\lvert S\rvert\lvert T\rvert$	$\lvert S\rvert\lvert T\rvert$
HDT	Scalar					$\lvert S\rvert+\lvert T\rvert$	$\lvert S\rvert$	$\lvert S\rvert\lvert T\rvert$	$\lvert S\rvert\lvert T\rvert$
HDD	Aggregate	2	N	G	N	$\lvert S\rvert+\lvert T\rvert$	$\lvert S\rvert$	$\lvert S\rvert$	$\lvert S\rvert$
HDTD	Aggregate					$\lvert S\rvert+\lvert T\rvert$	$\lvert S\rvert$	$\lvert S\rvert$	$\lvert S\rvert$
SJD	Aggregate					$\lvert S\rvert+\lvert T\rvert$	$\lvert S\rvert$	$\lvert S\rvert$	$\lvert S\rvert$
MCD	Aggregate	5	G	N	N	$\lvert S\rvert+\lvert T\rvert$	$\lvert S\rvert$	1	1
HDQ	Scalar					$\lvert S\rvert+\lvert T\rvert$	$\lvert S\rvert$	$\lvert T\rvert$	1
HDTQ	Scalar					$\lvert S\rvert+\lvert T\rvert$	$\lvert S\rvert$	$\lvert T\rvert$	1
MGD	Scalar	10	G_1	G_2	G_2	$\lvert S\rvert\lvert T\rvert$	$\lvert S\rvert\lvert T\rvert$	$\lvert T\rvert$	1
MSD	Scalar			S_2	S_2	$\lvert S\rvert\lvert T\rvert$	$\lvert S\rvert\lvert T\rvert$	1	1

Note: Input data are either not grouped (N), grouped (G), or sorted (S). Class 10 is first grouped on Q, indicated by G_1. For each quotient group, it is grouped (G_2) or sorted (S_2) on D in the same order as the divisor. The algorithm names corresponding to the abbreviations in the first column are given in Table 2.

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 *D*. 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 Fig. 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. 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 HDQ and HDTQ 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 is important to observe that one should not directly compare complexities of scalar and aggregate algorithms in Table 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 1 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 [1]. However, to simplify our presentation, the pseudo-code of MSD in the appendix 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 to HD and 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 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 Figs. 5(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 to 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.

5. The set containment problem

Universal quantification checks if all elements of a given set fulfill a given condition. In many applications, this condition is a set element membership test, i.e., the quantification problem becomes a set containment problem. For example, the problem stated in Section 1.1 can be rephrased as follows: "Find the students whose associated set of enrolled courses contains the given set of courses offered by the department."

5.1. Set storage representations

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 float or string. A more rigorous extension of the relational model is the *nested relational model* [10,11], where attributes can be relations themselves.

There are basically two orthogonal classifications for the storage representation of sets: nesting and location [12]. The attribute values are stored as multiple values: the nested representation stores the values as a variable length attribute and 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 Fig. 7, according to [12]. In this figure, we show as an example a single tuple of the relation *enrollment*(*student_id*, *courses*), where *student_id* is an atomic attribute and *courses* is a set-valued attribute. Here, we represent the fact that the student Chris is enrolled in the courses Compilers, Graphics, and Theory. Only the unnested internal representation conforms to the 1NF.

5.2. Set containment join and relational division

The set containment problem has been studied in great detail in the past [12–17]. In particular, several efficient set containment test algorithms have been developed and storage data structures to represent sets in relational, object-relational, and object-oriented databases are discussed.

It is interesting to observe that the division operator is closely related to set containment join, which can be implemented efficiently [12,14]. *Set containment join* (SCJ), denoted by $\bowtie_{\subseteq}$, is a join

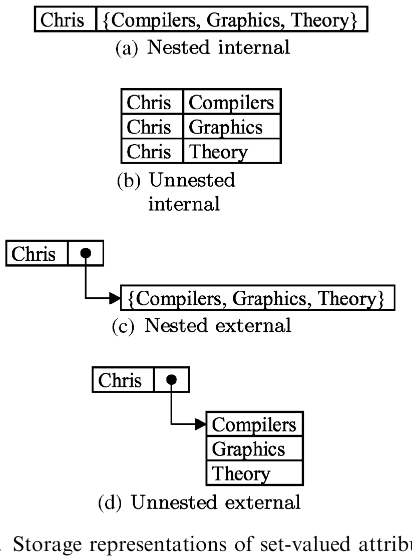


Fig. 7. Storage representations of set-valued attributes.

between the set-valued attributes b and c of two relations $R(a,b)$ and $S(c,d)$:

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

Fig. 8 illustrates an example computation of the set containment join based on the scenario introduced in Section 1.1. Only the table *course* has been changed by adding an additional attribute *program* that indicates which combination of CS courses are required for a certain advanced program. We find that Bob has all prerequisites to specialize in systems and applications while Chris is only allowed to specialize in applications.

Suppose, the tables *course* and *enrollment* are defined as before and that the layout of the set-valued attribute *courses* is *unnested internal* for both tables, as sketched in Fig. 9. We have not found a definition of such a result table in a nested internal representation in the literature. Since all join attributes are preserved, it is unclear how the rows belonging to a set on the one side are combined with tuples of another set on the other side. 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 compute the Cartesian product between the two groups of tuples that fulfill the set containment.

Because of this problem, we devised an extension of the division operator, called *set containment division* ($\div_{\supseteq}$) that returns the same rows as the set containment join but that delivers only the columns of the non-join attributes. Fig. 9 illustrates the behavior of set containment division based on the same input data as in Fig. 8 but using a 1NF data layout.

enrollment

student_id	*courses*
Alice	{C, T}
Bob	{C, D, G, T}
Chris	{C, G, T}

(a) $R(a,b)$

course

courses	*program*
{C, D, T}	Systems
{C, G}	Applications

(b) $S(c,d)$

course $\bowtie_{courses \subseteq courses}$ *enrollment*

student_id	*courses*	*courses*	*program*
Bob	{C, D, G, T}	{C, D, T}	Systems
Bob	{C, D, G, T}	{C, G}	Applications
Chris	{C, G, T}	{C, G}	Applications

(c) $S \bowtie_{c \subseteq b} R = T(a,b,c,d)$

Fig. 8. An example computation of the set containment join operator ($\bowtie_{\subseteq}$) based on relations in non-first normal form employing a set-valued attribute.

enrollment

student_id	*course_id*
Alice	Compilers
Alice	Theory
Bob	Compilers
Bob	Databases
Bob	Graphics
Bob	Theory
Chris	Compilers
Chris	Graphics
Chris	Theory

(a) $R(a,b)$

course

course_id	*program*
Compilers	Systems
Databases	Systems
Theory	Systems
Compilers	Applications
Graphics	Applications

(b) $S(c,d)$

enrollment $\div_{course\_id \supseteq course\_id}$ *course*

student_id	*program*
Bob	Systems
Bob	Applications
Chris	Applications

(c) $R \div_{b \supseteq c} S = T(a,d)$

Fig. 9. An example computation of the set containment division operator ($\div_{\supseteq}$) based on relations in first normal form.

Formally, the set containment division can be expressed with the help of (basic) relational division as follows, again based on the two relations $R(a,b)$ and $S(c,d)$:

$$\begin{aligned} T(a,d) &= R \div_{b \supseteq c} S \\ &= \bigcup_{x \in \pi_d(S)} ((R \div \pi_c(\sigma_{d=x}(S))) \times (x)). \end{aligned}$$

The idea of this expression is to merge the result of several divisions. In each division, the entire dividend R is divided by those tuples of the divisor S, which belong to the same group. There are as many divisions as the number of distinct values of $S.d$. We append the value of the current group to all result tuples of each division, specified by the Cartesian product.

We have seen that set containment join and relational division are very similar. We have demonstrated the similarity by defining an operator that operates on 1NF data like division but which can process many sets on both sides of the input like set containment join. The characteristics of the three operators discussed before are summarized in Table 4.

Table 4
Summary of operator characteristics

	Division	Set containment division	Set containment join
Operator and input relations	$R(a,b) \div S(c)$	$R(a,b) \div_{b \supseteq c} R(c,d)$	$S(c,d) \bowtie_{c \subseteq b} R(a,b)$
Left input/dividend	Many groups	Many groups	Many sets
Right input/divisor	Single group	Many groups	Many sets
Result/quotient attributes	$T(a)$	$T(a,d)$	$T(a,b,c,d)$
Data layout	1NF	1NF	Non-1NF

5.3. Overview of set containment join algorithms

The set containment join algorithms that have been proposed in the literature are based on signatures [13] and partitioning [17]. Enhanced approaches combining both techniques have been developed, which significantly outperform all known previous approaches [14–16]. All these algorithms assume that the data is managed by the database system in a non-1NF way, i.e., the data can be everything but unnested internal, which is the layout assumed for the set containment division and basic division problems. However, in [16] the new approaches are compared to SQL-based approaches based on *counting* the number of elements in the join result of both sets and comparing it to the set cardinality of the candidate subset. Such a comparison is incomplete because other SQL-based approaches using NOT EXISTS (as for division) have not been taken into consideration, as described in Section 3.2.

A recent study compared set containment joins based on a nested internal and an unnested internal set representation [18], also based on the counting approach, only. In particular, 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 is forced to apply 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 ID 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 IDs and set cardinalities. Finally, a filter condition appends only those set ID 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-loops approach. Unfortunately, the results have not been compared to more sophisticated approaches as the ones proposed, for example, in [14].

The research results on relational division should be applied to set containment in future work since the division problem can be considered a sub-problem of set containment join under the assumption that the sets are stored using an unnested internal representation. The main difference between both operations is that division is applied to a single set of dividend set elements, whereas set containment join compares possibly multiple sets from both sides of the join with each other. To the best of our knowledge, the strong commonality between relational division and set containment join has not been identified and investigated before.

6. New applications for universal quantification

In this section, we first argue why business intelligence problems are likely to benefit from being expressed using SQL. Then, we suggest a novel approach to compute frequent itemsets—a popular data mining task—which employs universal quantifications in the SQL queries.

6.1. Database mining

In business intelligence applications, several data mining and OLAP techniques are employed to extract novel and useful information from huge corporate data sets. Typically, the data sets are managed by a data warehouse that is based on relational database technology. Although the term 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 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 (replicated) data copies have to be maintained.
- *Scalability*: If extremely large data sets are to be mined then 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, it may be better to employ a different query.

The latter three arguments motivated our research on SQL-based algorithms for several data mining methods. One of these methods is discussed in the following section.

6.2. Frequent itemset discovery with SQL

In this section, we first briefly introduce the frequent itemset discovery problem and explain the relationship between frequent itemset discovery and relational division and set containment join. Then, we present a new approach for this problem, which makes use of universal quantifications.

6.2.1. The frequent itemset discovery problem

The computation of frequent itemsets is a computationally expensive preprocessing step for association rule discovery, which finds rules in large transactional data sets [19]. Frequent itemsets are combinations of items that appear frequently together in a given set of transactions. Association rules characterize, for example, the purchasing 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.

Given a set of transactions, the *frequent itemset discovery problem* is to find itemsets within the transactions that appear at least as frequently as a given threshold, called *minimum support*. For example, a user can define that an itemset is frequent if it appears in at least 2% of all transactions.

Almost all frequent 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 consists of two phases:

1. The *candidate generation phase* computes a set of potentially 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 stores them in F_k.

The key problem of frequent itemset discovery is: "How many transactions contain a certain given itemset?" This question can be answered in relational algebra using the division operator. Suppose that we have a relation *Transaction*(*tid*, *item*) containing a set of transactions and a relation *Itemset*(*item*) containing a single itemset, each row containing one item. We want to collect those *tid* values in a relation *Contains*(*tid*), where *for all* tuples in *Itemset* there is a corresponding tuple in *Transaction* that has a matching *item* value together with that *tid*. In relational algebra, this problem can be stated as

$$\mathit{Transaction}(\mathit{tid}, \mathit{item}) \div \mathit{Itemset}(\mathit{item}) = \mathit{Contains}(\mathit{tid}).$$

The example in Fig. 10 illustrates the division operation. The *Transaction* table consists of three

Transaction

tid	*item*
1001	diapers
1001	beer
1001	chips
1002	chips
1002	diapers
1003	beer
1003	avocados
1003	chips
1003	diapers

(a) Dividend

Itemset

item
chips
beer
diapers

(b) Divisor

Contains

tid
1001
1003

(c) Quotient

Fig. 10. Relationship between the frequent itemset discovery problem and relational division: $\mathit{Transaction} \div \mathit{Itemset} = \mathit{Contains}$.

transactions and two of them contain all items of *Itemset*. We simply have to count the values in *Contains* to decide if the itemset is frequent. For example, if the minimum support is set to 60% then the given itemset is considered a frequent itemset because the support is $\frac{2}{3}$, which is greater than 60%. Using division terminology, *Transaction* plays the role of the dividend, *Itemset* represents the divisor, and *Contains* is the quotient.

Unfortunately, frequent itemset discovery poses the additional problem that we have to check *many* (candidate) itemsets if they are frequent, i.e., unlike Fig. 10(b), we usually do not have a constant divisor relation but we need many divisor relations. However, we can employ efficient algorithms for this problem. We could arrange the itemsets in a table *Itemset (itemset, item)* and apply the division operation to each *itemset* group, separately. As shown in Section 5.2, this problem can also be expressed by set containment division:

$$\mathit{Itemset}(\mathit{itemset}, \mathit{item}) \div_{\mathit{item} \supseteq \mathit{item}} \mathit{Transaction}(\mathit{tid}, \mathit{item}) = \mathit{Contains}(\mathit{itemset}, \mathit{tid})$$

Another approach is to use the standard set containment join, which requires switching from the 1NF data representation to a non-1NF representation that uses set-valued attributes. We would have to preprocess the tables by transforming the *item* values of each group, defined by the *itemset* and *tid* values, respectively, into a set. Instead of the above tables in 1NF, the non-1NF tables would have a schema like: *Itemset (itemset, itemvalues)* and *Transaction (tid, itemvalues)*, each having a set-valued attribute *itemvalues*.

6.2.2. Support counting in SQL

In this paper, we focus on the support counting phase of frequent itemset discovery. For typical data sets, this phase is much more computationally expensive than the candidate generation phase.

There are several approaches to express the support counting phase in SQL. Most of them are based on SQL-92. The *SETM* algorithm is the first SQL-based approach described in the literature [20]. Several researchers have suggested improvements of SETM. It has been shown that SETM does not perform well on large data sets and new approaches have been devised, like for example *Three-Way-Join*, *Subquery*, and *Two-Group-Bys* [21]. The algorithms presented in that paper perform differently for different data characteristics. Subquery is reported to be the best approach 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.

More recently, an approach called *Set-Oriented Apriori* has been proposed [22]. The authors argue that too much redundant computations are involved in each support counting phase. Their performance results have shown that set-oriented apriori performs better than subquery, especially for high values of k.

We contrast our novel approach to previous approaches based on SQL-92 where the data is stored in 1NF, i.e., we do not investigate set-valued attributes, for example. One of the approaches based on SQL-92 is *K-Way-Join* [21], illustrated in Fig. 11. The K-Way-Join approach, which is based on SQL-92, uses k instances of the transaction table and joins it k times with itself and with a single instance of C_k. Same as all other known approaches based on SQL-92 that use a 1NF representation of itemsets, K-Way-Join assumes that the frequent and candidate k-itemsets are stored in a single row: ($itemset, item_1, \dots, item_k$). However, the given transactions are stored as multiple rows using the schema ($tid, item$). As we will show in the following section, our novel approach uses a data layout where itemsets are stored as multiple rows, same as the transactions.

```
SELECT    c.itemset, COUNT(*) AS support
FROM      Ck AS c, T AS t1, T AS t2, ..., T AS tk
WHERE     c.item1 = t1.item AND
          c.item2 = t2.item AND
          ...
          c.itemk = tk.item AND
          t1.tid = t2.tid AND
          t1.tid = t3.tid AND
          ...
          t1.tid = tk.tid
GROUP BY  c.itemset
HAVING    support >= @minimum_support
```

Fig. 11. Support counting phase according to the K-Way-Join algorithm.

6.2.3. *Support counting and universal quantification*

Based on the idea of using division to specify the itemset containment problem, we devised a complete algorithm, called *Quiver* (QUantified Itemset discovery using a VERtical table layout) [23], that employs SQL queries containing universal quantifications for both phases of the discovery task. The reason for devising a new approach is twofold:

1. We want to formulate intuitive queries that naturally express the universal quantification problem: "Count the number of transactions where for each transaction, *all* items of a given itemset are contained in the transaction." Previous approaches for SQL-based frequent itemset discovery are mostly "hardwired" queries, i.e., the quantification is circumvented by using many join conditions between individual items of candidates and transactions (as shown for the K-Way-Join approach in the previous section).
2. We want to employ a flexible itemset representation that is similar to the way transactions are stored in a database: *Transaction* ($tid, item$). In all previous approaches that use a 1NF representation, k-itemsets are stored as a single row: ($itemset, item_1, \dots, item_k$). Instead of this "horizontal" layout, Quiver uses a "vertical" layout, where a k-itemset is represented as k rows in the three-column table *(itemset, position,item)*. One benefit of this vertical layout is its ability to store even very large itemsets because in commercial database systems the maximum number of columns in a table is significantly lower than the number of rows.

In the following, we describe only the support counting phase of Quiver and we focus on the core problem, universal quantification. The entire approach, including the candidate generation phase using universal quantification, is described in detail in [23].

The query for support counting is first presented with help of tuple relational calculus since the calculus offers a universal quantifier to conveniently express the quantification. After this, we show how to derive an equivalent SQL query. As explained in Section 3.2, SQL does not offer a universal quantifier, therefore the query is expressed with the help of negated existential quantifiers.

Since Quiver follows the classical iterative two-phase approach, suppose that we have computed the set of candidate k-itemsets $C_k(\mathit{itemset}, \mathit{position}, \mathit{item})$ based on the set of frequent $(k-1)$-itemsets $F_k(\mathit{itemset}, \mathit{position}, \mathit{item})$ during the first phase of the kth iteration, with $k \geqslant 2$. The set of transactions is given by table $T(\mathit{tid}, \mathit{item})$.

We express the query Q in tuple relational calculus to derive combinations of transactions and candidates as

$$Q = \{(c_1 \in C, t_1 \in T) | \mathit{Contains}\}.$$

The query can be applied to candidate itemsets of any size. Therefore, the parameter k of the particular candidate set C_k is omitted for brevity. The *Contains* expression of this query is defined as

$$\begin{aligned} \mathit{Contains} = \forall c_2 \in C \exists t_2 \in T \\ &(c_2.\mathit{itemset} = c_1.\mathit{itemset}) \rightarrow \\ &(t_2.\mathit{tid} = t_1.\mathit{tid} \wedge \\ &t_2.\mathit{item} = c_2.\mathit{item}). \end{aligned}$$

The expression 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 c_1. The quantified (bound) tuple variables c_2 and t_2 represent the items belonging to c_1 and t_1, respectively. The universal quantification lies in the condition that for each item c_2 belonging to itemset c_1, there must be an item t_2 belonging to transaction t_1 that matches with c_2.

A combination (c_1, t_1) fulfilling the calculus query Q indicates that the itemset $c_1.\mathit{itemset}$ is contained in the transaction $t_1.\mathit{tid}$. We can find the support of each candidate by counting the number of distinct values $t_1.\mathit{tid}$ that appear in a combination $c_1.\mathit{itemset}$. We do not show the actual counting because the basic calculus does not include aggregate functions.

Since we are interested in an SQL representation of the given calculus query, we translate it into SQL in a straightforward manner by applying the following transformations:

- *Quantifiers*: As already explained before, there is no universal quantifier available in SQL. Therefore, we translate $\forall x \in R : f(x) \equiv \neg \exists x \in R : f(x)$ into "`NOT EXISTS (SELECT * FROM R AS x WHERE NOT f(x))`."
- *Implications*: We replace an implication by a disjunction, i.e., we transform $f \rightarrow g \equiv \neg f \vee g$ into "`NOT f OR g`."
- *Negations*: We use De Morgan's rules $\neg(f \wedge g) \equiv \neg f \vee \neg g$ and $\neg(f \vee g) \equiv \neg f \wedge \neg g$ for pushing a negation into a conjunction or a disjunction.

The resulting SQL query for support counting, shown in Fig. 12, contains two nested "NOT EXISTS" expressions analogous to the example SQL query used to express the student's enrollment problem in Section 3.2. Note that the query in Fig. 12 has to apply the aggregation on the set of *unique* transaction IDs because duplicates can occur as a result of the query processing.

To conclude this section, we point out that the Quiver approach shows how an important data mining task can be expressed in a natural way using universal quantification. If a database system were able to recognize the quantification problem inside queries like the one in Fig. 12, it could employ the most efficient algorithm that realizes the division operator, set containment division operator, or set containment join operator (discussed in Section 5), taking into account the current data characteristics, as explained in the previous sections. This is especially important if the data mining problem is a part of a larger, more complex query, involving several additional predicates. For example, consider a supermarket

```
SELECT itemset, COUNT(DISTINCT tid) AS support
FROM (
  SELECT c1.itemset, t1.tid
  FROM   Ck AS c1, T AS t1
  WHERE  NOT EXISTS (
    SELECT *
    FROM   Ck AS c2
    WHERE  NOT EXISTS (
      SELECT *
      FROM   T AS t2
      WHERE  NOT (c1.itemset = c2.itemset) OR
                 (t2.tid = t1.tid AND
                  t2.item = c2.item)))
) AS Contains
GROUP BY itemset
HAVING   support >= @minimum_support
```

Fig. 12. Support counting phase according to the Quiver algorithm.

scenario, where we restrict our analysis to transactions of the years 1999–2001, and we are only interested in items of the product category "soft drinks." Such additional predicates can significantly influence the choice on the most efficient algorithm for the quantification problem.

7. 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 [2]. 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 [24,25]. Universal quantification is expressed by new algebra operators and is optimized based on query graphs in a non-relational data model [25]. 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 [3,4,7]. Some of the algorithms reviewed in this paper have been compared both analytically and experimentally [1]. 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 [1] 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 falls into the category "direct algorithm based on sorting," while the hash-division for divisor groups algorithm of Section 3.4.3 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 algorithms, like grouping and sorting.

Based on a classification of queries that contain universal quantification, several query evaluation techniques have been analyzed [3]. The input data of this algorithm analysis is stored in an object-oriented or object-relational database, where set-valued attributes are available. Hence, the algorithms they examine can presuppose that the input data is grouped on certain attributes. For example, the table *enrollment* in Fig. 1 could be represented by a set-valued *enrolled_courses* attribute of a *student* class. The authors conclude that universal quantification based on anti-semi-join is superior to all other approaches, similar to the conclusion of [2]. 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 ($\bar{\ltimes}$) as well as semi-join ($\ltimes$) and left outer join ($\bowtie_{lo}$), is: $S \div T = ((S \ltimes T) \bowtie_{lo} T) \bar{\ltimes} T$.

In this paper, 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 [26]. Such quantifiers can be processed by algorithms that employ multi-dimensional matrix data structures [27]. 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 [28]. 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.

8. Conclusion and future work

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 SQL:1999 [6]. No keyword similar to "FOR ALL" [5] is available and division has to be expressed indirectly, for example by using nested "NOT EXISTS" clauses or by using the "division by counting" approach on the query language level. To the best of our knowledge, there is no database system that has an implementation of hash-division (or any of its improvements), although this efficient algorithm has been known for many years [4]. However, we believe that as soon as a dedicated keyword for universal quantification is supported by the SQL standard and its benefit is recognized and exploited by applications, many options and strategies are available today for database system vendors to implement an efficient division operator.

The similarity between relational division and the set containment join has been discussed for the first time. This may lead to more research that investigates the possibility of representing sets in an unnested storage layout because efficient algorithms for division can be exploited. We have proposed a new operator, called set containment division, that realizes set containment joins for data in first normal form.

We have discussed an important application of the division (and hence set containment) problem, namely frequent itemset discovery. We plan to investigate the potential of using universal quantification in queries in further data mining methods of business intelligence applications.

Our future work includes the analysis of further data properties that have an influence on the optimization of division queries, like the current data distribution or the availability of certain indexes. Furthermore, we will study the potential of parallelizing division algorithms, based on the detailed studies in [1] on parallelizing hash-division and aggregate algorithms. In addition, the comparison between division and set containment join algorithms deserves more attention. In particular, further investigations of both operators need to take into account the cost of nesting and unnesting between the 1NF and the non-1NF storage representations of sets in order to provide fair performance comparisons.

Acknowledgements

We would like to thank Sergey Melnik whose work on set containment joins led us think about the relationship between set containment joins and the classic relational division problem. Our work was facilitated by financial support from a DAAD/NSF cooperation. We acknowledge the helpful comments of the anonymous reviewers.

Appendix. Pseudo-code of division algorithms

The following algorithms in Figs 13–17 assume that the division's input consists of a dividend table $S(quotient, divisor)$ and a divisor table $T(divisor)$. Furthermore, we use the variables s and t to refer to a single row within S and T, respectively. The data structures *dht* and *qht* represent a divisor hash table and a quotient hash table, respectively.

```
s1 = s2 = s;
while not t.isEmpty() do
  insert t into dht;
while not s1.isEmpty() do
  if s1.quotient not in qht then begin
    while not s2.isEmpty() do
      if s1.quotient == s2.quotient and s2.divisor in dht then
        set bit of s2.divisor in dht;
    if no bit in dht is equal to zero then
      output row (s1.quotient);
    reset all bits in dht to zero;
    insert s1.quotient into qht;
  end;
```

Fig. 13. Nested-loops division (class 0).

```
// build the divisor hash table
divisor_count = 0;
while not t.isEmpty() do begin
  insert t into dht;
  t.divisor_number = divisor_count;
  divisor_count++;
end;
// build the quotient hash table
while not s.isEmpty() do
  if a matching divisor row t in dht is found then begin
    if no matching candidate quotient row q is found in qht then begin
      q = new quotient candidate row created from quotient attributes of
          dividend row s including a bitmap initialized with zeroes;
      insert q into qht;
    end;
    set bit in q's bitmap corresponding t.divisor_number;
  end;
// find result in the quotient table
foreach bucket in the quotient table do
  foreach row q in bucket do
    if the associated bitmap of q contains no zero then
      output row (q);
```

Fig. 14. Classic hash-division (class 0).

```
t_count = 0;
while not t.isEmpty() do begin
  t_count++;
  t.next();
end;
if not s.isEmpty() then begin
  s.next();
  current_quotient = s.quotient;
end;
while not s.isEmpty() do begin
  s_count = 0;
  while not s.isEmpty() and s.quotient == current_quotient do begin
    s_count++;
    s.next();
  end;
  if s_count == t_count then
    output row (current_quotient);
  if not s.isEmpty() then
    current_quotient = s.quotient;
end;
```

Fig. 15. Merge-count division (class 5).

```
is_first_row = true;
while not s.isEmpty() do begin
  if is_first_row and not t.isEmpty() then begin
    // this is the first time that we fetch a row from S
    s.next();
    t.next();
    is_first_row = false;
  end;
  current_quotient = s.quotient;
  while not s.isEmpty() and s.quotient == current_quotient and
        not t.isEmpty() and s.divisor <= t.divisor do begin
    while not s.isEmpty() and s.quotient == current_quotient and
          s.divisor < t.divisor do
      s.next();
    while not s.isEmpty() and s.quotient == current_quotient and
          not t.isEmpty() and s.divisor == t.divisor do begin
      s.next();
      t.next();
    end;
  end;
  if t.isEmpty() then
    // all divisor values of the divisor table have been matched
    output row (current_quotient);
  t.initialize();  // reopen the sorted divisor table
  if not t.isEmpty() then
    t.next();  // fetch the first divisor row
  while not s.isEmpty() and s.quotient == current_quotient do
    s.next();
end;
```

Fig. 16. Merge-sort division (class 10). Without loss of generality, the pseudo code assumes an ascending sort order.

```
// build the divisor hash table
while not t.isEmpty() do
  insert t into dht with a new bitmap initialized with zeroes;
// build the quotient hash table
quotient_count = 0;
while not s.isEmpty() do begin
  if not s.quotient is in qht then begin
    insert (s.quotient) into qht;
    index = (s.quotient).quotient_number = quotient_count;
    quotient_count++;
  else
    index = value of (s.quotient).quotient_number in qht;
  end;
  d = result of lookup of s.divisor in dht;
  d.bitmap[index] = 1;
end;
// find result in the divisor hash table
if number of rows in dht > 0 then begin
  bitmap = new bitmap initialized with ones;
  foreach bucket in the dht do
    foreach row d in bucket do
      bitmap = bitmap & d.bitmap;   // bit-wise AND operation
  foreach index value in bitmap == 1 do begin
    q = quotient row in qht associated with index;
    output row (q);
  end;
end;
```

Fig. 17. Transposed hash-division (class 0).

References

[1] G. Graefe, R. Cole, Fast algorithms for universal quantification in large databases, TODS 20 (2) (1995) 187–236.

[2] F. Bry, Towards an efficient evaluation of general queries: quantifier and disjunction processing revisited, in: Proceedings SIGMOD, Portland, OR, USA, May–June 1989, pp. 193–204.

[3] J. Claußen, A. Kemper, G. Moerkotte, K. Peithner, Optimizing queries with universal quantification in object-oriented and object-relational databases, in: Proceedings VLDB, Athens, Greece, August 1997, pp. 286–295.

[4] G. Graefe, Query evaluation techniques for large databases, ACM Comput. Surveys 25 (2) (1993) 73–170.

[5] P. Gulutzan, T. Pelzer, SQL-99 Complete, Really: An Example-Based Reference Manual of the New Standard, R&D Books, Lawrence, Kansas, USA, 1999.

[6] ANSI/ISO/IEC 9075-2, Information Technology, Database Language, SQL—Part 2: Foundation (SQL/Foundation), 1999.

[7] W3C, XQuery 1.0: An XML Query Language, Working Draft 7, W3C, 2001.

[8] J. Smith, P. Chang, Optimizing the performance of a relational algebra data base interface, CACM 18 (10) (1975) 568–579.

[9] C. Nippl, R. Rantzau, B. Mitschang, StreamJoin: a generic database approach to support the class of stream-oriented applications, in: Proceedings IDEAS, Yokohama, Japan, September 2000, pp. 83–91.

[10] G. Jaeschke, H.-J. Schek, Remarks on the algebra of non first normal form relations, in: Proceedings PODS, Los Angeles, California, USA, March 1982, pp. 124–138.

[11] A. Makinouchi, A consideration on normal form of not-necessarily-normalized relation in the relational data model, in: Proceedings VLDB, Tokyo, Japan, October 1977, pp. 447–453.

[12] K. Ramasamy, Efficient storage and query processing of set-valued attributes, Ph.D. Thesis, University of Wisconsin, Madison, WI, USA, 2002, 144pp.

[13] S. Helmer, G. Moerkotte, Evaluation of main memory join algorithms for joins with set comparison join predicates, in: Proceedings VLDB, Athens, Greece, August 1997, pp. 386–395.

[14] S. Melnik, H. Garcia-Molina, Adaptive algorithms for set containment joins, Department of Computer Science, Stanford University, CA, USA, Technical Report, November 2001.

[15] S. Melnik, H. Garcia-Molina, Divide-and-conquer algorithm for computing set containment joins, in: Proceedings EDBT, Prague, Czech Republic, March 2002, pp. 427–444.

[16] S. Melnik, H. Garcia-Molina, Divide-and-conquer algorithm for computing set containment joins, Stanford University, Extended Technical Report, CA, USA, 2002.

[17] K. Ramasamy, J. Patel, J. Naughton, R. Kaushik, Set containment joins: the good, the bad and the ugly, in: Proceedings VLDB, Cairo, Egypt, September 2000, pp. 351–362.

[18] S. Helmer, G. Moerkotte, Compiling away set containment and intersection joins, in: Technical Report 04/02, University of Mannheim, Germany, April 2002.

[19] R. Agrawal, R. Srikant, Fast algorithms for mining association rules, in: Proceedings VLDB, Santiago, Chile, September 1994, pp. 487–499.

[20] M. Houtsma, A. Swami, Set-oriented data mining in relational databases, DKE 17 (3) (1995) 245–262.

[21] S. Sarawagi, S. Thomas, R. Agrawal, Integrating association rule mining with relational database systems: alternatives and implications, in: Proceedings SIGMOD, Seattle, WA, USA, June 1998, pp. 343–354.

[22] S. Thomas, S. Chakravarthy, Performance evaluation and optimization of join queries for association rule mining, in: Proceedings DaWaK, Florence, Italy, August–September 1999, pp. 241–250.

[23] R. Rantzau, Frequent itemset discovery with SQL using universal quantification, in: Proceedings Workshop on Database Technology for Data Mining (DTDM), Prague, Czech Republic, March 2002, pp. 51–66.

[24] U. Dayal, Queries with quantifiers: a horticultural approach, in: Proceedings PODS, Atlanta, Georgia, USA, March 1983, pp. 125–136.

[25] U. Dayal, Of nests and trees: a unified approach to processing queries that contain nested subqueries, aggregates, and quantifiers, in: Proceedings VLDB, Brighton, England, September 1987, pp. 197–208.

[26] P. Hsu, D. Parker, Improving SQL with generalized quantifiers, in: Proceedings ICDE, Taipei, Taiwan, March 1995, pp. 298–305.

[27] S. Rao, A. Badia, D.V. Gucht, Providing better support for a class of decision support queries, in: Proceedings SIGMOD, Montreal, Canada, June 1996, pp. 217–227.

[28] M. Jarke, J. Koch, Range nesting: a fast method to evaluate quantified queries, in: Proceedings SIGMOD, San Jose, CA, USA, May 1983, pp. 196–206.