Universal Quantification in Relational Databases: A Classification of Data and Algorithms

Ralf Rantzau[1], Leonard Shapiro[2], Bernhard Mitschang[1], and Quan Wang[3]

[1] Computer Science Department, University of Stuttgart,
Breitwiesenstr. 20-22, 70565 Stuttgart, Germany
{rantzau, mitsch}@informatik.uni-stuttgart.de

[2] Computer Science Department, Portland State University,
P.O. Box 751, Portland, OR 97201-0751, Oregon, U.S.A.
len@cs.pdx.edu

[3] Oracle Corporation
quan.wang@oracle.com

Abstract. Queries containing universal quantification are used in many applications, including business intelligence applications. Several algorithms have been proposed to implement universal quantification efficiently. These algorithms are presented in an isolated manner in the research literature – typically, no relationships are shown between them. Furthermore, each of these algorithms claims to be superior to others, but in fact each algorithm has optimal performance only for certain types of input data. In this paper, we present a comprehensive survey of the structure and performance of algorithms for universal quantification. We introduce a framework for classifying all possible kinds 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. For the first time, we are able to identify the optimal algorithm to use for any given input dataset. These two classifications of input data and optimal algorithms are important for query optimization. They allow a query optimizer to make the best selection when optimizing at intermediate steps for the quantification problem.

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 $\forall x\colon f(x)$ 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.

enrollment (dividend) ÷ **course (divisor)** = **result (quotient)**

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

course_id
Compilers
Databases
Theory

student_id
Bob

Fig. 1. Division operation representing the query "Which students have taken all courses?"

1.1 Overview of the Division Operator

To illustrate the division operator we will use a simple example throughout the paper, illustrated in Figure 1, representing data from one department at a university [7]. 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.

In the example of Figure 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 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 [2] [3] [7]. 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 can employ an algorithm that exploits this structure, i.e., the algorithm does not have to restructure the input before it can start generating output data. Of course, there is no guarantee that such an algorithm is always "better" than an algorithm that requires previous restructuring. However, the division operator offers a variety of alternative algorithms that can exploit such a structure for the sake of good performance and low memory consumption.

Suppose we are fortunate and the input data is highly structured. For example, suppose the data has the schema of Figure 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.

How could such 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.

1.3 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 criteria of optimality are: the time to deliver the first/last result row, the amount of memory for internal, temporary data structures, the number of scans over the input data, or the ability to be non-blocking, i.e., to return some result rows before the entire input data are consumed.

Which algorithm should we use to process the division operation, given the dividend and divisor tables shown in Figure 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 Figure 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 3.2.4.

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

1.4 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. Section 5 gives a brief overview of related work. Section 6 concludes this 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 optimal algorithms in Section 3 that exploit the specific data properties of a class.

2.1 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 Figure 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.

Instead of grouping, one could use a nested-loops approach to process this query: 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.

If the data is grouped, then the grouping algorithm is clearly more efficient. Even if the data is not grouped, the aggregation approach is in general more efficient. For large datasets, the (at most) $n \cdot \log(n)$ cost of grouping and subsequent linear aggregation is typically cheaper than the n^2 cost of nested-loops.

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

Table 1. A classification of dividend and divisor. Attributes are either grouped (G) or not grouped (N). We use the same (a different) index of G$_i$ 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).

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

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 $O(n \cdot \log(n))$ as opposed to the nearly linear time complexity for hash-based grouping. Though sort-based grouping algorithms do more than necessary, both hash-based and sort-based grouping perform well for large datasets [6] [7].

2.2 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 Figure 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

Class 0:

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

Class 2:

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

Class 5:

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

Class 10:

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

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

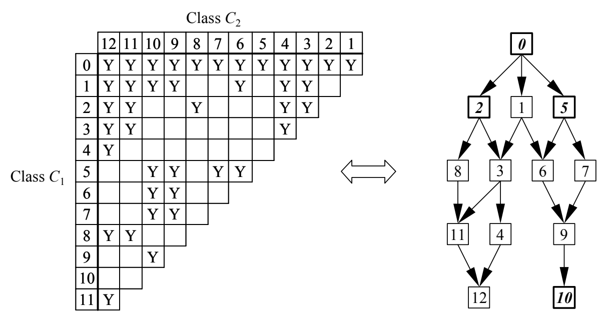
we will see later in this paper, some classes have no suitable algorithm that can exploit its specific combination of data properties. The classes that have at least one algorithm exploiting exactly its data properties are shown in italics. In class 0, for example, no table is grouped on an interesting 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 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.

Figure 2 illustrates four classes of input data for division, based on the example data of Figure 1. These classes, which are shown in italics in Table 1, are important for several algorithms that we present in the following section.

If we know that an algorithm can process data of a specific class, it is useful to know which other classes are also covered by the algorithm. This information can be represented, e.g., by a Boolean matrix like the one on the left in Figure 3. 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 in Figure 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 9–12.

For the subsequent discussion of division algorithms, we define two terms to refer to certain row subsets of the dividend. Suppose the dividend S is grouped on Q (D), i.e., suppose the dividend belongs to class 5 (2) and all its descendants in Figure 3. Furthermore, suppose v is one specific value a group in D (Q). 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



Class C_1 \ Class C_2	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											

$$\begin{tikzcd}
 & & & \boxed{\mathit{0}} \arrow[dl] \arrow[d] \arrow[drr] & & & \\
 & \boxed{\mathit{2}} \arrow[dl] \arrow[dr] & & \boxed{1} \arrow[dl] \arrow[dr] & & \boxed{\mathit{5}} \arrow[dl] \arrow[dr] & \\
\boxed{8} \arrow[d] & & \boxed{3} \arrow[dll] \arrow[d] & & \boxed{6} \arrow[dr] & & \boxed{7} \arrow[dl] \\
\boxed{11} \arrow[dr] & & \boxed{4} \arrow[dl] & & & \boxed{9} \arrow[d] & \\
 & \boxed{12} & & & & \boxed{\mathit{10}} &
\end{tikzcd}$$

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.

example, in the *enrollment* table of class 5 in Figure 2, the quotient group of Alice consists of the rows {(Alice, Theory), (Alice, Compilers)}. Similarly, the divisor group of Databases in class 2 in Figure 2 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. In Section 4, we will analyze and compare the effectiveness of each algorithm with respect to the data classification of Section 2.

Each division algorithm (analogous to other classes of algorithms, like joins, for example) has performance advantages for certain data characteristics. No algorithm is able to outperform the others for every input data conceivable.

The following algorithms assume that the division's input consists of a dividend table $S(Q \cup D)$ 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 2.2.

3.1 Algorithm Classification

In this section, we present a classification of algorithms based on what kind of data structures are employed. In addition, we illustrate how universal quantification is expressed in a query language.

There are two fundamental approaches to processing relational division. The first one relies on direct row matches between the dividend's divisor attributes D and the divisor table. We call this class of algorithms *scalar* to contrast them to the second class, called *aggregate* algorithms. 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.

Any query involving universal quantification can be replaced by a query that makes use of counting [7]. However, formulating division queries that involve aggregation is non-intuitive and error-prone because one has to take care of duplicates, NULL values, and referential integrity.

3.2 Scalar Algorithms

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

3.2.1 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 Figure 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 scan 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* and add the quotient value to *seen_quotients*.

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.

In the illustration in box (1) in Figure 4, we assume hash tables as the data structures used for matching. It shows the divisor/quotient hash tables that represent *seen_divisors* and *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.

3.2.2 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.

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. Due to space restrictions we refer to [12] for details on this algorithm.

Box (2) in Figure 4 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.

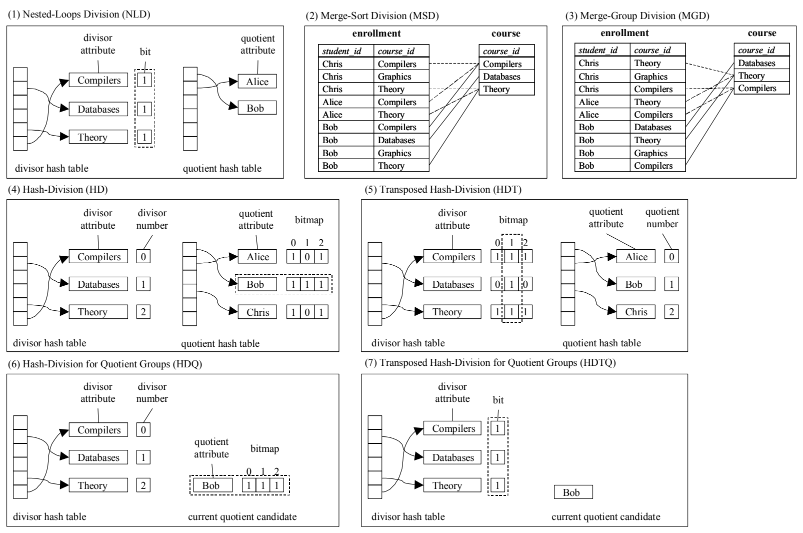


Fig. 4. Overview of the data structures and processing used in scalar algorithms. The value setting is based on the example from Figure 1. Except for boxes (2) and (3), broken lined boxes indicate that a quotient is found.

3.2.3 Merge-Group Division

We can generalize merge-sort division to an algorithm that we call *merge-group division* (MGD). In contrast to MSD, we assume that (1) both inputs are only grouped and not necessarily sorted on the divisor attributes, but that (2) 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. Due to space restrictions we refer to [12] for details on this algorithm.

3.2.4 Classic Hash-Division

In this section, we present the classic *hash-division* (HD) algorithm [7]. 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 box (4) in Figure 4. 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.

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 box (4) of Figure 4, the contents of the hash tables are shown for the time when all dividend and divisor rows of Figure 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.

3.2.5 Transposed Hash-Division

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

Same as the classic hash-division algorithm, HDT first builds the divisor hash table. However, we store a bitmap with each row of the *divisor*. A value of 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. All bits of the bitmap, whose size is the same as the bitmaps in the divisor hash table, 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.

Boxes (4) and (5) of Figure 4 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 Figure 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.

3.2.6 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 presents some optimizations for quotient groups. The optimizations of hash-division for divisor groups are presented in Section 3.3.3.

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.

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 sketch the data structures used in the boxes (6) and (7) of Figure 4 for the time when the group of dividend rows containing the quotient candidate Bob have been processed.

3.3 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*.

3.3.1 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 box (1) of Figure 5. 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.

3.3.2 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 3.2.2 and 3.2.3. Instead of comparing the elements of quotient groups with the divisor, MCD uses a representative (the row count) of each quotient group to compare it with the divisor's aggregate. Box (2) in Figure 5 illustrates the single scan required to compare the size of the each quotient group with the divisor size.

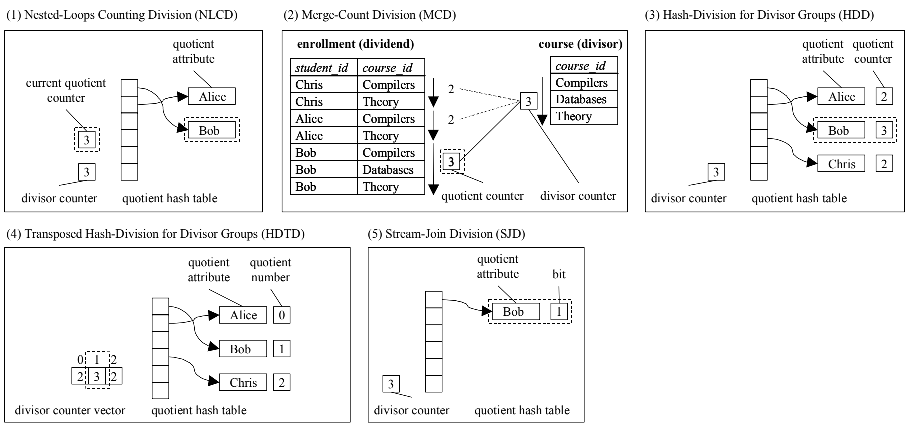


Fig. 5. Overview data structures used in aggregate algorithms. Broken lined boxes indicate that a quotient is found. Only Bob's group has as many dividend rows as the divisor.

3.3.3 Hash-Division for Divisor Groups

In Section 3.2.6, 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.

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 sketch the data structures used in the boxes (3) and (4) of Figure 5 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. Bob is the only student who is contained in all three divisor groups.

3.3.4 Stream-Join Division

The new algorithm *stream-join division* (SJD) [11] 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.

Box (5) of Figure 5 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 Figure 2 with the exception that the Graphics group {(Bob, Graphics), (Chris, Graphics)} is missing, due to referential integrity.

The advantage of SJD lies in the fact that the amount of memory can decrease but will never increase after the divisors 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 [11]) 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 2 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 Figure 3. The overview of division algorithms in Table 2 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.

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 Figure 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 2. 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 2 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.

Table 2. Overview of division algorithms showing for each algorithm the class of required input data, its algorithm class, and its time and memory complexities. 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.

Division Algorithm	Abbre-viation	Algorithm Class	Data Class	Dividend S		Divisor T	Complexity in O-Notation			
							Time		Memory	
				Q	D		worst	avg.	worst	avg.
Nested-Loops Counting Division	NLCD	aggregate	0	N	N	N	$\|S\|^2 + \|T\|$	$\|S\|^2$	1	1
Nested-Loops Division	NLD	scalar					$\|S\|^2 + \|T\|$	$\|S\|^2$	$\|S\| + \|T\|$	$\|S\|$
Hash-Division	HD	scalar					$\|S\| + \|T\|$	$\|S\|$	$\|S\|\times\|T\|$	$\|S\|\times\|T\|$
Transposed Hash-Division	HDT	scalar					$\|S\| + \|T\|$	$\|S\|$	$\|S\|\times\|T\|$	$\|S\|\times\|T\|$
Hash-Division for Divisor Groups	HDD	aggregate	2	N	G	N	$\|S\| + \|T\|$	$\|S\|$	$\|S\|$	$\|S\|$
Transp. Hash-Div. for Divisor Groups	HDTD	aggregate					$\|S\| + \|T\|$	$\|S\|$	$\|S\|$	$\|S\|$
Stream-Join Division	SJD	aggregate					$\|S\| + \|T\|$	$\|S\|$	$\|S\|$	$\|S\|$
Merge-Count Division	MCD	aggregate	5	G	N	N	$\|S\| + \|T\|$	$\|S\|$	1	1
Hash-Division for Quotient Groups	HDQ	scalar					$\|S\| + \|T\|$	$\|S\|$	$\|T\|$	1
Transp. Hash-Div. for Quotient Groups	HDTQ	scalar					$\|S\| + \|T\|$	$\|S\|$	$\|T\|$	1
Merge-Group Division	MGD	scalar	10	G_1	G_2	G_2	$\|S\|\times\|T\|$	$\|S\|\times\|T\|$	$\|T\|$	1
Merge-Sort Division	MSD	scalar			S_2	S_2	$\|S\| + \|T\|$	$\|S\|$	1	1

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 [7].

5 Related Work

Quantifiers in queries can be expressed by relational algebra. Due to the lack of efficient division algorithms in the past, early work has recommended avoiding the relational division operator to express universal quantification in queries [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 [4] [5]. Universal quantification is expressed by new algebra operators and is optimized based on query graphs in a non-relational data model [5]. 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] [6] [7]. Some of the algorithms reviewed in this paper have been compared both analytically and experimentally [7]. 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 [7] 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 3.2.2 falls into the category "direct algorithm based on sorting," while the hash-division for divisor groups algorithm of Section 3.3.3 belongs to the combination "indirect algorithm based on hashing." Our classification details these four approaches and focuses on the fact that data properties should be exploited as much as possible by employing "slim" algorithms that are separated from preprocessing algorithms, like grouping and sorting. Our analysis is more sensitive to the properties of input data. For example, if the input data is in class 2 (where the data is grouped on the dividend's divisor attributes), then as shown in Section 3.3.4, the stream-join division algorithm is at least as efficient as the hash-division algorithm of [7] but requires less memory.

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 Figure 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 paper has a broader definition of queries involving universal quantification than the classic definition that involves the division operator. However, the anti-semi-join approach requires a considerable overhead for preprocessing the dividend. An equivalent definition of the division operator using anti-semi-join ($\overline{\ltimes}$) as well as semi-join ($\ltimes$) and left outer join ($⟕$), is: $S \div T = ((S \ltimes T)\ ⟕\ T)\ \overline{\ltimes}\ T$.

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 [9]. Such quantifiers can be processed by algorithms that employ multi-dimensional matrix data structures [13]. In that paper, however, the implementation of an operator called *all* is presented that is similar but different form 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 [10]. 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.

6 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 [1]. No keyword similar to "FOR ALL" [8] is available and division has to be expressed indirectly, for example by using two negated "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 [6]. However, we believe that as soon as a dedicated keyword for universal quantification is supported by a 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.

Note that division requires a "constant" divisor. It is also common that queries involve a correlated divisor, e.g. "Which students have taken all courses of their major?" Unfortunately, such queries cannot be translated into a simple division query. However, it may be possible to employ several divisions for such a single query, each division having a single constant divisor. This could be a worthwhile strategy if the number of divisors is low and if they can be easily computed in advance.

Our future work includes the analysis of further data properties that have an influence on 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 [7] on parallelizing hash-division and aggregate algorithms. Finally, we plan to investigate the potential of using universal quantification in queries of business intelligence applications.

References

1. ANSI/ISO/IEC 9075-2: Information Technology – Database Language – SQL – Part 2: Foundation (SQL/Foundation). (1999)
2. Bry, F.: Towards an Efficient Evaluation of General Queries: Quantifier and Disjunction Processing Revisited. SIGMOD 1989: 193–204
3. Claußen, J., Kemper, A., Moerkotte, G., Peithner, K: Optimizing Queries with Universal Quantification in Object-Oriented and Object-Relational Databases. VLDB 1997: 286–295
4. Dayal, U.: Processing Queries with Quantifiers: A Horticultural Approach. PODS 1983: 125–136
5. Dayal, U.: Of Nests and Trees: A Unified Approach to Processing Queries that Contain Nested Subqueries, Aggregates, and Quantifiers. VLDB 1987: 197–208
6. Graefe, G.: Query Evaluation Techniques for Large Databases. ACM Computing Surveys 25(2): 73–170 (1993)
7. Graefe, G., Cole, R.: Fast Algorithms for Universal Quantification in Large Databases. ACM Transactions on Database Systems 20(2): 187–236 (1995)
8. Gulutzan, P., Pelzer, T.: SQL-99 Complete, Really: An Example-Based Reference Manual of the New Standard. R&D Books, Lawrence, Kansas, U.S.A., 1999
9. Hsu, P.-Y., Parker, D.: Improving SQL with Generalized Quantifiers. ICDE 1995: 298–305
10. Jarke, M., Koch, J.: Range Nesting: A Fast Method to Evaluate Quantified Queries. SIGMOD 1983: 196–206
11. Nippl, C., Rantzau, R., Mitschang, B.: StreamJoin: A Generic Database Approach to Support the Class of Stream-Oriented Applications. IDEAS 2000: 83–91
12. Rantzau, R., Shapiro, L., Mitschang, B., Wang, Q.: Universal Quantification in Relational Databases: A Classification of Data and Algorithms. Technical Report, Computer Science Department, University of Stuttgart, 2002 (to appear)
13. Rao, S., Badia, A., van Gucht, D.: Providing Better Support for a Class of Decision Support Queries. SIGMOD 1996: 217–227