Relational Division:
Four Algorithms and Their Performance

Goetz Graefe

Oregon Graduate Center
Department of Computer Science
and Engineering
19600 N.W. von Neumann Drive
Beaverton, OR 97006-1999 USA

Technical Report No. CS/E 88-022

Abstract

We outline three known algorithms for relational division, the algebra operator used to express universal quantification (for-all conditions), and a new algorithm called **Hash-Division**. By comparing the algorithms analytically and experimentally, we show that the new algorithm provides performance competitive or superior to techniques used to-date, namely techniques using sorting or aggregate functions. Furthermore, the new algorithm can eliminate duplicates in the divisor on the fly, ignores duplicates in the divided, and allows two kinds of partioning, either of which can be used to resolve hash table overflow or to efficiently implement the algorithm on a multi-processor system.

Relational Division:
Four Algorithms and Their Performance

Goetz Graefe
Oregon Graduate Center

Abstract

We outline three known algorithms for relational division, the algebra operator used to express universal quantification (for-all conditions), and a new algorithm called **Hash-Division**. By comparing the algorithms analytically and experimentally, we show that the new algorithm provides performance competitive or superior to techniques used to-date, namely techniques using sorting or aggregate functions. Furthermore, the new algorithm can eliminate duplicates in the divisor on the fly, ignores duplicates in the dividend, and allows two kinds of partitioning, either of which can be used to resolve hash table overflow or to efficiently implement the algorithm on a multi-processor system.

1. Introduction

Relational completeness [Codd1972a] includes universal quantification, i.e., the ability of a relational query system to evaluate "for all" predicates. For-all predicates can be expressed in relational algebra using the division operator. Due to the lack of efficient algorithms, however, this operator has not been implemented in most systems. A number of reasons seem to justify this omission. First, the division operator can be expressed in terms of other relational operators. If $R(r,s)$ and $S(s)$ are relations, then $R \div S = \pi_r(R) - \pi_r((\pi_r(R) \times S) - R)$. Second, it can be expressed using aggregate functions, as described in Section 2. Third, for-all predicates are not needed very frequently, so why bother?

The first reason is of merely theoretical validity since the equivalent expression contains a Cartesian product operator. The second reason is valid, but as we will see in Sections 4 and 5, implementing division by means of aggregate functions may result in the inferior algorithms. The third reason may be true for accounting databases; it probably does not hold for database systems that support logic programming or enforce complex integrity constraints on sets.

In this paper, we introduce a new algorithm for relational division called **Hash-Division**. This algorithm uses two hash tables, one for the divisor and one for the quotient. Both analytical and experimental results show that the performance of this new algorithm is competitive or superior to that of previously used methods.

The remainder of this paper is organized as follows. Section 2 gives an overview of three algorithms that have been used for division. In Section 3, we describe the new division algorithm in detail. Section 4 contains an analytical comparison of the four algorithms, and Section 5 shows our experimental results. Section 6 describes how hash-division can be used effectively in a multi-processor system. Section 7 contains our summary and conclusions.

2. Previous Work

In order to describe the algorithms most clearly, we introduce two examples that will be used throughout the paper. Assume a university database with two relations, *Courses (course-no, title)* and *Transcript (student-id, course-no, grade)* with the obvious key attributes. For the first example, we are interested in finding the students who have taken all courses offered by the university. In relational calculus, this is expressed as

> find the students (*student-id*'s) such that **for all** courses (*course-no*'s) in *Courses*, a tuple with this *student-id* and *course-no* appears in the *Transcript* relation.

In relational algebra, this query is

$$\pi_{student-id,course-no}(Transcript) \div \pi_{course-no}(Courses).$$

In this example, the projection of *Transcript* is the **dividend**, the projection of *Courses* is the **divisor**, and the division result is called the **quotient**. The attributes of the divisor are called **divisor attributes**, $course-no$ in the example. The **quotient attributes** are the attributes of the dividend that are not in the divisor, $student-id$ in the example.

It is important to notice that in this example both relations are projected on their key attributes. Thus, the problem of duplicate tuples does not arise. If the inputs of the division may contain duplicates, either the division algorithm employed must be able to deal with them,

or the inputs must be properly preprocessed. Duplicate elimination can be quite expensive, making an algorithm very desirable that is insensitive to duplicates in its inputs.

As our second example, we are interested in finding the students who have taken all database classes, i.e., courses for which the title attribute contains the string "database". For this example, the divisor is restricted by a prior selection. While these two examples seem almost identical, the difference has ramifications when division is implemented using aggregations, which is described below.

2.1. A Naive Sort-Based Algorithm

The first algorithm is the most naive one; it directly implements the calculus predicate. First, the dividend is sorted using the quotient attributes as major and the divisor attributes as minor sort keys. In the examples, *Transcript* is sorted on *student-id*'s and, for equal *student-id*'s, on *course-no*'s. Second, the divisor is sorted on all its attributes. Third, the two sorted relations are scanned in a fashion similar to nested loops join. The dividend serves as outer, the divisor as inner relation. Differently than in nested loops join, when an equality match has been found, both relation scans can be advanced. The dividend is scanned exactly once, whereas the divisor is scanned once entirely for each quotient tuple, and partially for each candidate quotient tuple which actually does not participate in the quotient. Note that the dividend relation can contain a tuple that does not match with any of the divisor tuples, e.g., a *Transcript* tuple of a physics course in the second example. Further details of the scan logic are left to the reader. Essentially this algorithm was proposed in [Smith1975a]. It is the first algorithm analyzed in Sections 4 and 5.

2.2. Implementing Division by Aggregation

Since repeated scans in the above algorithm suggest that it may be rather slow, it seems worthwhile to search for alternatives. One such alternative uses aggregations. In some relational database management systems, aggregation is the only way to express for-all predicates.

The first example query can be expressed as

> find the students who have taken as many courses as there are courses offered by the university[1].

This query is evaluated in three steps. First, the courses offered by the university are counted using a scalar aggregate operator. Second, for each student, the courses taken are counted using an aggregate function operator. Third, only those students whose number of courses taken is equal to the number of courses offered are selected to be included in the quotient.

The second example query can be expressed as

> find the students who have taken as many database courses as there are database courses offered by the university.

While the first and the third step in the algorithm above remain virtually the same, the second one becomes significantly more complex. Since it is important to count only those tuples from the *Transcript* relation which refer to database courses, the aggregate function must be preceded by a semi-join or simply a join of *Transcript* and *Courses* restricted to database courses.

The scalar aggregate operator can be implemented quite easily, e.g., using a file scan, and similarly the final selection. The aggregate function and the join require more effort; in the remainder of this section, we will only concern ourselves with these operators.

2.2.1. Division Using Sort-Based Aggregation

The traditional way of implementing aggregate functions relies on sorting [Epstein1979a]. In the examples, *Transcript* is sorted on attribute *student-id*. Afterwards, the count of courses taken by each student can be determined in a single file scan. The exact logic of this scan is left to the reader. An obvious optimization of this algorithm is to perform aggregation during

---

[1] Typically, it is necessary to explicitly request uniqueness of the *student-id*'s and *course-no*'s counted. In most systems, the default for aggregation is not to eliminate duplicates. The reason is that many applications must consider duplicates, e.g., *sum of salaries by department* is different than *sum of distinct salaries by department*. Thus, in cases where aggregation is used to express for-all predicates, a duplicate elimination step is explicitly requested and inserted into the query evaluation plan.

sorting, i.e., whenever two tuples with equal sort keys are found, they are aggregated into one tuple, thus reducing the number of tuples written to temporary files.

If the query requires a join prior to the aggregation as in the second example, any of the join algorithms available in the system can be used, typically merge join, index join, or their semi-join versions if they exist. If merge join is used, notice that the relation must be sorted on different than the grouping attributes. In the example, it must be sorted first on *course-no*'s for the join and then on *student-id*'s for aggregation.

Sort-based aggregation has been used in a number of systems, e.g., INGRES [Epstein1979a]. It is the second algorithm analyzed in Sections 4 and 5.

2.2.2. Division Using Hash-Based Aggregation

It was an interesting insight that sorting actually results in more order than necessary for a number of relational algebra operations. In the examples, it is not important that the *Transcript* tuples be rearranged in ascending *student-id* order. It is only necessary that *Transcript* tuples with equal *student-id* attribute are brought together. The fastest way to achieve this uses hashing. Thus, a number of hash-based algorithms have been proposed for join, duplicate elimination, and aggregate functions, e.g. [Bratbergsengen1984a, DeWitt1984a, DeWitt1985a].

Hash-based aggregate functions keep the tuples of the output relation in a main memory hash-table. The output relation contains the grouping attributes, *student-id* in the examples, and one or more aggregation values, e.g., a sum, a count, or both in the case of average computation. Each input tuple is either aggregated into an existing output tuple with matching grouping attributes, or it is used to create a new output tuple. When the entire input in consumed, the result of the aggregate function is stored in the hash table. Note that, since the hash table contains only the aggregation output, it is not necessary that the aggregation input fit into main memory. In the examples, if there are 500 students with a total of 10,000 *Transcript* tuples, the hash table needs to hold only 500 tuples. Thus, hash aggregation performs well, i.e., without I/O for temporary files, for much larger files than sort-based aggregation.

If the aggregate function is preceded by a join as in the second example, the join can also be implemented using hashing. The hash table used for the join is a different one than the one used for aggregation, just as sort-based join and aggregation require two sorting steps on different attributes. The hash table in the semi-join is built by hashing on *course-no*'s, whereas the hash table for the aggregation is built on *student-id*'s.

Hash-based aggregation has been used only in a small number of systems, e.g. in GAMMA [DeWitt1986a]. Division using hash-based aggregation is the third algorithm analyzed in Sections 4 and 5.

As an aside, let us briefly consider duplicates again. In the naive division algorithm and in sort-based aggregation, duplicates can be conveniently eliminated during the initial sort phase. Hash-based aggregation, however, cannot include duplicate elimination, since only one tuple is kept in the hash table for each group. While efficient duplicate elimination schemes based on hashing exist [Gerber1986a], they require that the entire input must be kept in main memory hash tables or in overflow files. Thus, duplicate elimination based on hashing may be impractical for a very large dividend relation.

3. Hash-Division

This section contains a description of the new algorithm followed by an example, a discussion of the algorithm, and some considerations on hash table overflow.

3.1. Description

In this section, we describe the new algorithm for relational division. A pseudo-code version of the hash-division algorithm is given in Figure 1. It uses two hash tables, one for the divisor and one for the quotient. The first hash table is called the *divisor table*, the second the *quotient table*. With each tuple in the divisor table, an integer value is kept, called the *divisor number*. With each tuple in the quotient table, a bit map is kept with one bit for each divisor tuple.

```
/* step 1: building the divisor table */
assign divisor count ← 0
initialize empty divisor table
open divisor input
for each divisor tuple
    calculate hash bucket in divisor table
    insert divisor tuple into hash bucket
    assign tuple's divisor number ← divisor count
    assign divisor count ← divisor count + 1
close divisor input

/* step 2: building the quotient table */
initialize empty quotient table
open dividend input
for each dividend tuple
    calculate hash bucket in divisor table
    scan hash bucket for a matching divisor tuple
    if match is found in divisor table
        calculate hash bucket in quotient table
        scan hash bucket for a matching quotient tuple
        if no match is found in quotient table
            create new quotient tuple and bit map
            insert into hash bucket of quotient table
            project dividend tuple into quotient tuple
            clear bit map
        set bit corresponding to divisor tuple's divisor number
close dividend input
free divisor table

/* step 3: finding result in the quotient table */
for each bucket in quotient table
    for each tuple in bucket
        test bit map for a zero bit
        if no zero is found
            print quotient tuple
free quotient table
```

Figure 1. The Hash-Division Algorithm.

The algorithm proceeds in three steps. First, it inserts all divisor tuples into the divisor table. The bucket is determined by hashing on all attributes. In the process, the algorithm counts the divisor tuples, and stores the current count to the divisor number when the tuple is inserted. Thus, all divisor tuples receive a unique number.

Second, the algorithm consumes the dividend relation. For each dividend tuple, it first checks whether it corresponds to a divisor tuple in the divisor table by hashing and matching the dividend tuple on the divisor attributes. If no such divisor tuple exists, the dividend tuple is immediately discarded. In the second example, a student's *Transcript* tuple for a physics course does not pass this test and is not considered further as a candidate for the quotient. If a matching divisor tuple is found, its divisor number is kept and the dividend tuple is considered a quotient candidate. Next, the algorithm determines whether a matching quotient candidate already exists in the quotient table by hashing and matching the dividend tuple on the quotient attributes. If no such quotient candidate exists, e.g., because the quotient table is empty at the beginning, a new quotient candidate tuple is created by projecting the dividend tuple on the quotient attributes, and the new quotient tuple is inserted into the quotient table. Together with the new tuple, a bit map is created with one bit for each divisor tuple in the divisor table. This bit map is initialized with zero's, except for the bit that corresponds to the divisor number kept earlier. If, however, a matching quotient candidate tuple already exists, all that needs to be done is to set one bit in the quotient candidate's bit map.

Finally, after all dividend tuples have been consumed, the quotient of the relational division consists exactly of those tuples in the quotient table for which the bit map contains no zero, which can be determined by a simple algorithm scanning all buckets in the quotient table.

3.2. An Example

Let us consider a concrete example. Assume that we have, after suitable selections and projections, the relations shown in Figure 2. We need to find the students who have taken all database courses, i.e. the quotient of the two relations.

First, the *Courses* relation is read and its tuples are inserted into the divisor table. Divisor number 0 is assigned to tuple (*Database*1), and 1 to (*Database*2). Second, the *Transcript* relation is read. For its first tuple, (*Ann*,*Database*1), a matching divisor tuple, (*Database*1), is located in the divisor table, but no matching quotient tuple can be found since the quotient

Transcript

student–id	*course–no*
Ann	Database 1
Barb	Database 2
Ann	Database 2
Barb	Optics

Courses

course–no
Database 1
Database 2

Figure 2. Example Relations.

table is still empty at this point. Thus, a new quotient tuple, (*Ann*), is created and a bit map with two bits is initialized with zero's. The first bit (indexed by 0) in the bit map is then set to one since it corresponds to the divisor tuple, (*Database*1). For the second dividend tuple, (*Barb*,*Database*2), another quotient tuple and a bit map are created in the same way. For the third dividend tuple, (*Ann*,*Database*2), both a matching divisor tuple, (*Database*2), and a matching quotient tuple, (*Ann*), can be found in the two hash tables, and the second bit (indexed by 1) in the bit map of (*Ann*) is set to one. The last dividend tuple, (*Barb*,*Optics*), does not have a matching divisor tuple in the divisor table, i.e., (*Optics*), and this dividend tuple is discarded.

Finally, the quotient table is scanned for tuples and bit maps with no remaining zero's. The only such tuple and bit map is (*Ann*), and this tuple is printed as the quotient table. In fact, this is the only student who has taken both database courses.

3.3. Discussion

A number of observations can be made. First, the algorithm does not require a *stop-and-go* operator on its input, i.e., an operator that has to consume its entire input before producing its first output such as sort. Thus, it can smoothly receive its inputs from a dataflow query processing system.

Second, the algorithm is a stop-and-go operator itself; only after both inputs are consumed does it produce the quotient relation by scanning the quotient table. Fortunately, this problem can be alleviated by associating an additional counter with each quotient tuple and some minor modifications of the algorithm. When a quotient tuple is first created, the counter is set to zero. Before setting a bit in the quotient table, the modified algorithm tests whether or not this bit position is set already. If it is, the dividend tuple is discarded. If it is not, the bit in the bit map is set to one, and the counter is incremented and compared with the divisor count. If the counters are equal, the quotient tuple is produced immediately. With these modifications, the algorithm can also be used as a producer in a dataflow query processing system.

Third, if one or both of the inputs of the division are stored in disk files, the algorithm may outperform other division algorithms simple because it does not require random I/O and thus allows efficient read-ahead of physically clustered or contiguous files. Since this is common to most hash-based database processing algorithms, we will not explore this issue further.

Fourth, the algorithm requires efficient handling of bit maps, including a scan over a possibly large bit map. Since many computer architectures provide special instructions for bit maps, we do not consider this to be a problem. Note that initializing a bit map and searching for a single zero in a bit map can be done by inspecting a word at a time.

Fifth, duplicates in the divisor can be eliminated while building the divisor table. Duplicates in the dividend are ignored automatically since they map to the same bit in the same bit map.

Sixth, it is interesting to compare hash-division with division using on hash-based aggregation with prior semi-join. In either case, there are two hash tables. The first contains the divisor relation for the semi-join or as divisor table, respectively. The second hash table is used to develop the quotient relation. The counter used for aggregation serves the same function as the bit map in hash-division. Using bit maps, however, provides the additional functionality that duplicates in the dividend are ignored. If duplicates are known not to be a problem, hash-

division could be modified to employ counters instead of divisor numbers and bit maps.

Finally, hash-division depends on sufficient main memory to hold both hash tables. Recall, however, that the divisor and the quotient are the smaller relations involved; the big relation is the dividend as it is a superset of the Cartesian product of divisor and quotient. Even though the quotient table contains actually some more tuples than the quotient, we expect that the memory requirements do not pose a major problem in most cases.

If, however, the divisor table or the quotient table are larger than the available main memory, *hash table overflow* occurs and portions of one or both tables must be temporarily spooled to secondary storage. Alternatively, hash-division can be modified to take advantage of a multi-processor system. In the next section, we describe techniques for handling hash table overflow in a single processor database system. Adaptations of the hash-division algorithm for multi-processor systems are discussed in Section 6.

3.4. Hash Table Overflow

If the available memory is not sufficient for divisor table and quotient table, the input data must be partitioned into disjoint subsets called *clusters* that can be processed in multiple *phases*. The clusters are processed one at a time. The first cluster in kept in main memory while the other clusters are spooled to temporary files, one for each cluster, in a way similar to hybrid hash-join [DeWitt1984a]. For hash-division, there are two partitioning strategies which can be used alone or together.

In the first strategy, called *quotient partitioning*, the dividend relation is partitioned on the quotient attributes using a partitioning strategy such as range-partitioning or hash-partitioning. Each phase produces a *quotient cluster*, which is the quotient of one dividend cluster and the divisor. The quotient of the entire division is the union (concatenation) of all quotients clusters. Since all dividend clusters are divided with the entire divisor, the divisor table must be kept in main memory during all phases. While this may be a problem for large divisors, it certainly is not a problem if the divisor is rather small.

The second strategy, called *divisor partitioning*, partitions both the divisor and the dividend relations using the same partitioning function applied to the divisor attributes. Each phase performs the division algorithm, producing one quotient cluster. Notice that the quotient clusters are different for quotient partitioning and divisor partitioning. For quotient partitioning, the quotient clusters must be gathered in a final *collection phase*. Only the quotient tuples that were produced by all single-phase divisions, i.e., the tuples that appear in all quotient clusters, participate in the final result. This set can easily be determined since this problem is exactly the division problem again. Thus, in order to obtain the final result, each quotient tuple produced by a the single-phase division is tagged with the phase number. The collection phase divides the union (concatenation) of all quotient clusters over the set of phase numbers. However, instead of using a divisor table to determine which bit to set in the bit maps, the phase number can be used. Thus, the collection phase can skip the first step of hash-division.

4. Analytical Comparison of the Algorithms

In this section, we first develop the cost formulas for the algorithms discussed above and then apply these formulas to some relation sizes.

For the analytical comparisons, we assume a dividend relation R (tuple cardinality $|R|$, page cardinality r) and a divisor relation S (tuple cardinality $|S|$, page cardinality s) with quotient relation Q (tuple cardinality $|Q|$, page cardinality q). Further, we assume m main memory pages, with $s+q < m < r$. Our cost measure consists of both CPU and I/O costs, both measured as delay times without overlap of CPU and I/O activity. The cost units, their values (in milliseconds), and their description are given in Table 1.

We will give the cost formula for the easy case of $R = Q \times S$, i.e., all tuples of R participate in the quotient. The reason for chosing this case is the difficulty of defining an *average* case. For the division algorithms based on aggregation, the cost formulas for the versions with and without join are given. None of the cost formulas includes the cost of projecting the quotient tuples from dividend tuples and writing the quotient as these costs are common to all

Unit	ms	Description
RIO	30	random I/O, one page from or to disk
SIO	15	sequential I/O, one page from or to disk
Comp	0.03	comparison of two tuples
Hash	0.03	calculation of a hash value from a tuple
Move	0.4	memory to memory copy of one page
Bit	0.003	setting a bit in a bit map, and clearing and scanning a bit in a bit map

Table 1. Cost Units.

algorithms. Furthermore, we restrict our analysis to duplicate free inputs because we feel that it is the more frequent case. Notice, though, that duplicate elimination may be very expensive, in particular for large relations.

4.1. Sorting

Since several of the algorithms require sorting, we first give the formulas for the cost of sorting. We distinguish for relations that fit in main memory and those that do not. For the former we assume quicksort with an approximate cost function of

$$2\,|S|\,\log_2(|S|)\;Comp \tag{1}$$

for relation S since it fits in main memory. For relations larger than main memory, we assume a disk-based merge-sort algorithm. Its cost is

$$\log_m(r/m)\left(r\,(2\;RIO + Move) + |R|\,\log_2(m)\;Comp\right) + 2\,|R|\,\log_2(|R|\;m/r)\;Comp \tag{2}$$

for relation R since it does not fit in main memory. The first portion of the formula is the product of the number of merge passes and the cost of each merge, the second portion calculates the cost of sorting the initial runs using quicksort.

4.2. Naive Division Algorithm

For sorted inputs, the analysis of the naive algorithm is as straightforward as the algorithm itself. In the assumed case, the cost for the actual division step is

$$(r + s)\ SIO + |R|\ Comp \tag{3}$$

as the outer relation is scanned once and the inner is assumed to be kept in buffer memory.

4.3. Division Using Sort-Based Aggregation

If we assume for simplicity that the aggregation step is performed in the output procedure of the final merge step, the cost of a sort-based aggregate function consists of sorting plus the cost of comparing grouping attributes, i.e.,

$$|R|\ Comp. \tag{4}$$

Since the aggregation consist of simply incrementing a counter, we ignore its cost. The cost of the scalar aggregate, i.e., counting the cardinality of the divisor, is

$$s\ SIO. \tag{5}$$

If a merge-join is required before aggregation, the cost for an additional sort step and the cost of merging must be added to the actual aggregation cost. In the assumed case of $R = Q \times S$, the merge step uses the same logic as the naive algorithm, except that more comparisons are performed. Thus, the merge-join cost is

$$(r + s)\ SIO + |R|\,|S|\ Comp. \tag{6}$$

We assume that the entire S relation is kept in buffer memory during join processing. We ignore the cost of memory to memory copying since the join is actually a semi-join, which can be implemented without copying.

4.4. Division Using Hash-Based Aggregation

For hash-based aggregation, the cost is

$$r\ SIO + |R|\ (Hash + hbs\ Comp) + s\ SIO, \tag{7}$$

where *hbs* is the average size of each hash bucket, and the last part is the cost of the scalar aggregate. If a semi-join is required, its cost is

$$(s+r)\ SIO + |S|\ Hash + |R|\ (Hash + hbs\ Comp). \quad (8)$$

4.5. Hash-Division

Recall that we assumed that $s+q < m < r$, meaning that divisor and quotient tables fit in main memory and no hash table overflow occurs. When hash-division is used, a join is never necessary (just as in the naive algorithm), and the cost for the division is

$$(r + s)\ SIO + |S|\ Hash + |R|\ (2\ (Hash + hbs\ Comp) + Bit). \quad (9)$$

Both input relations are read sequentially. For each divisor and each dividend tuple, a hash bucket is calculated and the tuple is compared with all tuples in this bucket, on the average two tuples.

4.6. Some Example Relation Sizes

Let us use the formulas derived above to compare the four algorithms for relational division. We consider three sizes for S and Q, 25, 100, or 400 tuples. We assume that 10 tuples of either S or Q fit on one page, which implies that 5 tuples of R fit on one page. The memory used for sorting or hash tables is 100 pages, the average hash bucket size *hbs* is 2. Furthermore, we assume that neither R nor S are sorted originally.

Table 2 shows the run-times of the algorithms calculated with the formulas provided above. We realize that the formulas are not precise and that the assumption of $R = Q \times S$ does not represent all cases or the "typical" case. Nevertheless, we believe that Table 2 allows some interesting insights.

First, division by sort-based aggregation does not perform much better than the naive algorithm. In fact, if a semi-join is necessary to ensure that only valid tuples are counted, the additional sort cost makes division by aggregation significantly more expensive.

Second, if hash-based algorithms are used for the aggregation, the division can be performed much faster. The reason is that the cost for sorting the large dividend relation with multiple passes strongly dominates the cost for aggregation.

$\|S\|$	$\|Q\|$	Naive Div.	Sort-Aggregation		Hash-Aggregation		Hash-Div.
			no join	with join	no join	with join	
			[Times in milliseconds]				
25	25	9949	8074	18529	1969	3938	2028
25	100	39663	32163	73738	7763	15526	7996
25	400	158517	128517	294572	30938	61876	31868
100	25	39808	32308	79766	7875	15753	8111
100	100	158662	128662	317475	31050	62103	31983
100	400	634080	514080	1268311	123750	247503	127473
400	25	159280	129280	409160	31500	63012	32442
400	100	634698	514698	1629996	124200	248412	127932
400	400	2536369	2056369	6513339	495000	990012	509892

Table 2. Analytical Cost of Division.

Third, the new hash-division algorithm performs competitively with division by aggregation based on hashing. However, if a semi-join is needed, hash-division outperforms division by hash-based aggregation.

At this point, it may be allowed to do some speculations. If we drop the assumption that $R = Q \times S$, i.e., the dividend relation contains tuples that either do not match with any divisor tuple or do not participate in the quotient, we expect that hash-division always outperforms all other algorithms because tuples that do not match with any divisor tuple are eliminated early, even though dividend tuples that do not participate in the quotient are inserted in the quotient table.

In order to verify this analysis, we implemented the algorithms and obtained experimental results.

5. Experimental Comparison of the Algorithms

Before we report on the experiments, we give an overview of how we implemented the algorithms.

5.1. Implementation

All experiments were run on a Digital Equipment Corp. MicroVax II on top of a record-oriented file system developed at the Oregon Graduate Center using experiences from WiSS [Chou1985a] and GAMMA [DeWitt1986a]. It simulates a disk using a UNIX file or main memory. Its main services are extent-based files, records, B^+-trees, scans, a fast buffer manager, and a main memory manager. Copying is avoided as scans give memory addresses to records fixed in the buffer pool. When all buffer slots are fixed and a new request cannot be satisfied, the buffer pool grows dynamically until the main memory pool is exhausted, and shrinks as buffer slots are unfixed. An unfix call indicates whether the page can be replaced immediately or should be inserted into an LRU list. The initial buffer size is 256 KB, 100 KB of which can be used as sort buffer. For intermediate query results, the buffer manager also supports virtual devices, i.e., records can have a record identifier and can be fixed in the buffer pool but disappear when unfixed. Thus, all operators are programmed as if input and output were permanent files.

All relational algebra operators are implemented as iterators, i.e., they support a simple *open-next-close* protocol[2]. A tree-structured query evaluation plan is used to execute queries by demand-driven dataflow passing record identifiers and record addresses in the buffer pool between operators. All functions on data records, e.g., comparison and hashing, are compiled prior to execution and passed to the processing algorithms by means of pointers to the function entry points.

The naive division algorithm was implemented in such a way that if first consumes the entire divisor relation, building a linked list of divisor tuples fixed in the buffer pool. It then consumes the dividend relation, advancing in the linked list of divisor tuples as matching divi-

[2] Opening a sort operator prepares sorted runs and merges them until only one merge step is left. The final merge is performed on demand by the *next* function.

dend tuples are produced by the dividend input, and producing a quotient tuple each time the end of the divisor list is reached.

Sort-based aggregation is implemented by the sort procedure. Our implementation of sort performs aggregation and duplicate elimination as early as possible, i.e., no intermediate run contains duplicate sort keys. Merge join consists of a merging scan of both inputs, in which tuples from the inner relation with equal key values are kept in a linked list of tuples pinned in the buffer pool. For semi-joins in which the outer relation produces the result, no linked lists are used.

In our implementation of hash-based algorithms, we use bucket chaining as conflict resolution in hash tables. The hash algorithms use the file system's memory manager to allocate space for hash tables, bit maps, and chain elements. Chain elements are auxiliary data structures that contain a pointer to the next tuple in the bucket, a tuple's record identifier and main memory address in the buffer pool, and the divisor count or the pointer to the bit map respectively.

Since we wanted to shield our measurements as much as possible from the operating system's file system, we used only the operating system measure for CPU cost (milli-seconds in user mode, from *getrusage* UNIX system call). The I/O cost was calculated based on statistics collected by our file system. The transfer size, i.e., the unit of I/O, was chosen to be 8 KB, except for sort runs where it was 1 KB to allow high fan-in. Table 3 shows the statistics gathered and their weight as used in reporting experimental results. Unfortunately, we could not use very much disk space, so we had to restrict our record sizes to 8 bytes for the divisor and the quotient, and to 16 bytes for the dividend.

5.2. Results

Table 4 shows the run-time of the algorithms as observed in our experiments. The experimental results verify all observations made in the analytical comparisons. In particular, the

ms	Cost
20	Physical seek on device
8	Rotational latency per transfer
0.5	Transfer time per KByte
2	CPU cost per transfer

Table 3. Experimental Cost of Division.

$\|S\|$	$\|Q\|$	Naive Div.	Sort-Aggregation		Hash-Aggregation		Hash-Div.
			no join	with join	no join	with join	
			[Times in milliseconds]				
25	25	978	648	1288	438	578	428
25	100	4230	2650	5000	1130	1530	1160
25	400	24356	10175	27987	3850	5640	4240
100	25	3710	2590	5120	1100	1510	1180
100	100	25305	10847	28393	3750	5470	4160
100	400	108049	42643	115678	14226	20976	16056
400	25	25686	12286	29573	3920	5540	4220
400	100	108279	47937	120412	14376	20946	16046
400	400	448470	190745	490765	56094	82414	63574

Table 4. Experimental Cost of Division.

ranking of the algorithms is as it were in the analytical comparison. The sort-based algorithms perform significantly poorer than the hash-based algorithms, and a preceding semi-join makes division expressed with aggregate functions inferior to direct implementations of division. The differences are marked for small relation sizes, but the factor of difference grows as the relations grow. Even for small relation sizes, $|R| = 625$, $|Q| = 25$, and $|S| = 25$, we observed a factor of 3 difference between the fastest and slowest division algorithms, 1288ms vs. 428ms. Thus, the implementation of division is unimportant only for very small dividend and divisor relations. If the dividend or the divisor are results of other database operations, e.g., selection or projection, the possible error in the selectivity estimate makes it imperative to choose the division algorithm very carefully.

The differences in the relative performance of the algorithms between the analytical and the experimental results come from the fact that the buffer size and management is different than assumed in the analytical comparison. In several cases in the experiment, the entire dividend relation fits into the buffer, so that no I/O costs occur due to sorting. Furthermore, only 40% of the entire buffer is used as sort space (100 KB of 256 KB), so that temporary file pages remain in the buffer pool from run creation to merging and deletion. Only when a large dividend relation has to be sorted twice, i.e., in the case of sort-based aggregation with preceding join, this effect cannot be observed, and the preceding join and additional sort more than double the cost of sort-based aggregation (e.g., $|S| = |Q| = 400$, 490,765ms vs. 190,745ms).

It is interesting to note that if a universal quantification is expressed in terms of an aggregate function with preceding join and the query optimizer does not rewrite the query to use relational division, the query may be evaluated using an inferior strategy. This is true for both sort-based query evaluation systems such as System R or Ingres (sort-based aggregation vs. naive division) and for hash-based systems such as GAMMA (hash-based aggregation vs. hash-division). Since it is much easier to implement a query optimizer that rewrites a division operator into an aggregation operator than vice versa, universial quantification should be included as a language construct in database query languages, e.g., as a "contains" clause.

In summary, hash-division is only about 10% slower than the fastest algorithm considered (hash-based aggregation without preceding semi-join), but it is more powerful as it never requires a preceding semi-join or duplicate elimination on its inputs.

6. Multi-Processor Implementations

In this section, we want to consider three questions. First, if a multi-processor system of the shared-nothing type such as GAMMA [DeWitt1986a] is available, can the hash-division algorithm be adapted to make good use of this "horse power?" Second, if hash-division's divisor table is larger than main memory, can a multi-processor configuration alleviate the problem? Third, if hash-division's quotient table is larger than main memory, can a multi-processor configuration

alleviate the problem? We will explore the first question is some detail; answers to the other two questions will then be easy to find. Two more issues will be addressed towards the end of the section.

The key for efficient parallel algorithms seems to be that the input data for the problem at hand can be partitioned into disjoint subsets that can be processed distributedly without requiring too much synchronization. Fortunately, both partitioning strategies discussed above for hash table overflow, i.e., quotient partitioning and divisor partitioning, can be employed.

When quotient partitioning is used, the divisor table must be *replicated* in the main memory of all participating processors. After replication, all local hash-division operators work completely independently of each other.

If divisor partitioning is chosen, the resulting clusters are processed in parallel instead of in phases as discussed for hash table overflow. If the basic hash-division algorithm is modified such that it produces quotient tuples as soon as possible (as described in Section 3.3), the collection phase can be overlapped with producing the clusters. Instead of tagging the quotient tuples with phase numbers, processor network addresses are attached to the tuples, and the collection site divides the set of all incoming tuples over the set of processor network addresses. In the unlikely case that the central collection site becomes a bottleneck, it is possible to decentralize the collection step using quotient partitioning.

Now let us return to the second and third question posed at the beginning of this section. If the divisor table is too large to fit in each local main memory, the divisor can be partitioned over all local main memories available in the database machine. If the quotient is larger than each local main memory, it is possible to partition the quotient after replicating the divisor.

The obvious fourth question is: what happens if neither one of these partitioning strategies work because both divisor and quotient are too large? In this case it will be necessary to resort to combinations of the techniques discussed above, i.e., divisor partitioning, quotient partitioning, and hash table overflow management. The optimal mix of partitioning strategies may be

interesting but too specialized to be discussed here.

Finally, since network activity can become a bottleneck in a shared-nothing database machine, it may be worthwhile to reduce network activity by bit vector filtering [Babb1979a]. The bit vector can be used to avoid shipping tuples for which no divisor record exists, e.g., *Transcript* tuples for an optics course in the second example. As with bit vector filters for join, the selection of tuples is only a heuristic, e.g., in the second example, a *Transcript* tuple for an agriculture course will erroneously pass the bit vector filter if it maps to the same bit as one of the database courses. Nevertheless, bit vector filters may reduce significantly the network cost for the dividend relation, which is the larger of the division operands.

Let us briefly consider how well the other division algorithms are suited for multi-processor dataflow implementations. We believe that sort-based methods have two inherent problems, namely that sorting is a stop-and-go operator, and that the final merge pass of the sort operation either introduces a bottleneck or requires very sophisticated scheduling and partitioning. Division using aggregate functions based on hashing are definitely competitive if no semi-join is required (first example above). However, if a semi-join is required (second example above), the dividend relation must be partitioned and shipped across the interconnection network twice, thus increasing the cost significantly in most environments.

7. Summary and Conclusions

In this paper, we have described and compared three known algorithms for relational division, naive division and sort- and hash-based aggregation (counting), and a new algorithm called **hash-division**.

Hash-division uses two hash tables. The *divisor table* is used to match dividend tuples with divisor tuples and to determine a unique divisor number. The *quotient table* contains all quotient candidates; a bit map with each quotient candidate (indexed by divisor numbers) is used to keep track of which matching dividend tuples have been encountered so far. Hash table overflow can be resolved efficiently using *divisor-partitioning*, *quotient-partitioning*, or both.

Hash-division can be used very effectively in a dataflow scheme or in a multi-processor system.

Both analytical and experimental comparisons reported here demonstrate the competitiveness of hash-division. Naive division is slow because it requires sorted inputs. Division by sort-based aggregate functions is almost as expensive, even more expensive if a semi-join and an additional sort are required to ensure that only proper dividend tuples are counted. Division by hash-based aggregate functions is slightly faster than hash-division, but only if no preceding semi-join is required. All algorithms except hash-division require uniqueness in their inputs, which may require further expensive preprocessing.

An additional result of our comparisons is that direct algorithms can outperform aggregate functions, both in sort-based and in hash-based query evaluation systems. Thus, it is desirable either to include *for-all* predicates in the query language, or to detect them automatically in a complex aggregate expression.

Universal quantification and relational division have been neglected in database systems, not because they are useless (in fact, they are very powerful), but because efficiency posed a major problem. Hash-division is an interesting new algorithm as it is both fast and general.

References

Babb1979a.
E. Babb, "Implementing a Relational Database by Means of Specialized Hardware," *ACM Transactions on Database Systems* **4**(1) pp. 1-29 (March 1979).

Bratbergsengen1984a.
K. Bratbergsengen, "Hashing Methods and Relational Algebra Operations," *Proceedings of the Conference on Very Large Data Bases*, pp. 323-333 (August 1984).

Chou1985a.
H.T. Chou, D.J. DeWitt, R.H. Katz, and A.C. Klug, "Design and Implementation of the Wisconsin Storage System," *Software - Practice and Experience* **15**(10) pp. 943-962 (October 1985).

Codd1972a.
E.F. Codd, "Relational Completeness of Database Sublanguages," pp. 65-98 in *Data Base Systems*, ed. R. Rustin,Prentice-Hall, New York (1972).

DeWitt1984a.
D.J. DeWitt, R. Katz, F. Olken, L. Shapiro, M. Stonebraker, and D. Wood, "Implementation Techniques for Main Memory Database Systems," *Proceedings of the ACM SIGMOD*

Conference, pp. 1-8 (June 1984).

DeWitt1985a.
D.J. DeWitt and R.H. Gerber, "Multiprocessor Hash-Based Join Algorithms," *Proceedings of the Conference on Very Large Data Bases*, pp. 151-164 (August 1985).

DeWitt1986a.
D.J. DeWitt, R.H. Gerber, G. Graefe, M.L. Heytens, K.B. Kumar, and M. Muralikrishna, "GAMMA - A High Performance Dataflow Database Machine," *Proceedings of the Conference on Very Large Data Bases*, pp. 228-237 (August 1986).

Epstein1979a.
R. Epstein, "Techniques for Processing of Aggregates in Relational Database Systems," *UCB/ERL Memorandum*, (M79/8)University of California, (February 1979).

Gerber1986a.
R. Gerber, "Dataflow Query Processing using Multiprocessor Hash-Partitioned Algorithms," *Ph.D. Thesis*, University of Wisconsin, (October 1986).

Smith1975a.
J.M. Smith and P.Y.T. Chang, "Optimizing the Performance of a Relational Algebra Database Interface," *Communications of the ACM* **18**(10) pp. 568-579 (October 1975).