Fast Algorithms for Universal Quantification in Large Databases

GOETZ GRAEFE
Portland State University
and
RICHARD L. COLE
Redbrick Systems

Universal quantification is not supported directly in most database systems despite the fact that it adds significant power to a system's query processing and inference capabilities, in particular for the analysis of many-to-many relationships and of set-valued attributes. One of the main reasons for this omission has been that universal quantification algorithms and their performance have not been explored for large databases. In this article, we describe and compare three known algorithms and one recently proposed algorithm for relational division, the algebra operator that embodies universal quantification. For each algorithm, we investigate the performance effects of explicit duplicate removal and referential integrity enforcement, variants for inputs larger than memory, and parallel execution strategies. Analytical and experimental performance comparisons illustrate the substantial differences among the algorithms. Moreover, comparisons demonstrate that the recently proposed division algorithm evaluates a universal quantification predicate over two relations as fast as hash (semi-) join evaluates an existential quantification predicate over the same relations. Thus, existential and universal quantification can be supported with equal efficiency by adding the recently proposed algorithm to a query evaluation system. A second result of our study is that universal quantification should be expressed directly in a database query language because most query optimizers do not recognize the rather indirect formulations available in SQL as relational division, and therefore produce very poor evaluation plans for many universal quantification queries.

Categories and Subject Descriptors: E.5 [**Data**]: Files; H.2.3 [**Database Management**]: Languages; H.2.4 [**Database Management**]: Systems—*query processing*

General Terms: Algorithms, Experimentation

This research has been partially supported by the National Science Foundation with grants IRI-8996270, IRI-8912618, and IRI-9116547, the Advanced Research Projects Agency (ARPA order number 18, monitored by the US Army Research Laboratory under contract DAAB-07-91-C-Q518), Texas Instruments, Digital Equipment Corp., Intel Supercomputer Systems Division, Sequent Computer Systems, ADP, and the Oregon Advanced Computing Institute (OACIS).
Contact author's current address: G. Graefe, Microsoft Corp., One Microsoft Way, Redmond, WA 98052-6399 ⟨goetzg@microsoft.com⟩.

1. INTRODUCTION

Quantification is a powerful concept for querying sets and databases, and database query languages such as SQL support existential quantification, which can easily be implemented using well-known semi-join algorithms. However, relational completeness also requires universal quantification, i.e., the ability of a database system to evaluate complex "for-all" predicates [Codd 1972]. Similarly, a relationally complete algebra includes relational division, the operator required for most complex universal quantification queries. Nonetheless, most query evaluation systems do not provide the relational division operation or algorithms to implement it.

As an example of a universal quantification query, consider a market research request to find customers who satisfy *all* of a given list of criteria, using a relation R that indicates which customer meets which criteria. This query is to find those customers A such that for each criterion B in the given list, there exists a tuple in R matching A and B. In relational algebra, this query is a division of R by the list of interesting criteria. In fact, the query is a typical application of universal quantification and relational division, which are not supported directly or efficiently in any commercial relational database system.

Other examples can easily be imagined. Which students have taken all computer science courses required to graduate? Which courses have been taken by all students of a research group? Which suppliers can provide all parts for a certain assembly? Which parts are available from all regional suppliers? Which applicants possess all skills required for a new job opening? Which skills are common to all recent hires? In order to illustrate the ubiquity of useful universal quantification and relational division queries, these questions are intentionally based on the relational schemas frequently used for simple database examples. In fact, any many-to-many relationship induces a pair of universal quantification queries, not only the two many-to-many relationships underlying the first four questions above. Similarly, any set-valued attribute suggests a pair of universal quantification queries, not only the "skills"-attribute underlying the last two questions. Many of these queries answer very useful questions in the real world, in particular when combined with a restriction on the divisor, e.g., the restrictions to "computer science" and to "recent hires."

Given the power of universal quantification and relational division to analyze many-to-many relationships and set-valued attributes, why have they been neglected in the past? We believe that this omission has largely been due to the lack of efficient implementation algorithms. This article attempts to remedy this problem by surveying algorithms for relational division and comparing their suitability for small data volumes (all data fit in memory), for large data volumes, and for parallel execution. Our primary results is that a recently proposed algorithm called *hash-division* is superior to all prior algorithms by its generality and its performance in simple and complex cases as well as in parallel query execution systems. Our experiments demonstrate that relational division can be computed as fast as a

hash-based join or semi-join of the same pair of input relations. However, this performance can be achieved only if query language and query formulation enable the optimizer to detect universal quantifications that can be implemented by relational division, and if the fastest relation division algorithm is used.

Because universal quantification and relational division have been neglected in past research, let us name and address four traditional rationales for not supporting universal quantification and relational division explicitly in database query languages and in query evaluation systems. First, relational division can be expressed in terms of other relational operators. Second, universal quantification can be expressed using negated existential quantifications. Third, it can be expressed using aggregate functions. Fourth, for-all predicates are not used very frequently, so why bother?

The first rationale is correct: if $R(A, B)$ and $S(B)$ are relations, then $R \div S = \pi_A(R) - \pi_A((\pi_A(R) \times S) - R)$ [Maier 1983]. However, Cartesian products tend to be very large, and this one must be matched against the dividend R in a difference operation, which is as expensive as a join. Thus, a query evaluation plan based on the equivalent expression using a Cartesian product operator is very inefficient. We will present more efficient query evaluation plans that do not require Cartesian product operations, and will therefore ignore this alternative to relational division in our performance comparisons.

The second rationale is also correct, although universal quantification queries expressed using negated existential quantifications are very complex, as we will see in Section 2. In SQL, for example, two nested "NOT EXISTS" clauses and subqueries are required for our simple example queries, even in the absence of additional query predicates. Given this complexity, universal quantification queries are hard to specify correctly in SQL. Furthermore, query optimizers typically will not recognize such queries as universal quantifications and relational divisions, and therefore will choose query evaluation plans much slower than the ones proposed in this article. In fact, many commercial relational optimizers do not unnest any subqueries; in those systems, formulating a universal quantification query using two "NOT EXISTS" clauses results in a query evaluation plan similar to the relational algebra expression above. At best, if both inputs are indexed with clustering B-trees, the query plan for a universal quantification query with two "NOT EXISTS" clauses will be similar to the naive division algorithm discussed in Section 2.

The third rationale is also valid, and SQL formulations for relational division queries based on aggregation and their query evaluation plans will be discussed in Section 2. Unfortunately, formulations of universal quantification and relational division queries based on aggregation are also very complex. Moreover, as we will see in the performance evaluation, implementing division by means of aggregate functions results in poor performance except in rare circumstances—competitive performance requires that both inputs be duplicate-free, that referential integrity between the inputs hold,

and that the query processor use hash-based, not sort-based, aggregation. Whenever the divisor is the result of a selection, referential integrity between the relational division inputs will not hold and must be enforced explicitly in a separate step, which is particularly expensive in parallel query evaluation systems. Thus, direct algorithms that avoid aggregation are even more important for modern, parallel database systems than they have been in the past.

The fourth rationale seems to be the most important one: universal quantification and relational division are rarely taught and used. However, it is not at all clear what is cause and what is effect. Is it possible that universal quantification queries are shunned because they are complex to express in SQL and, if used, tend to run a long time? While universal quantification may not be useful in transaction-processing environments, universal quantification and relational division are very valuable in analyzing many-to-many relationships and set-valued attributes, as discussed earlier. Moreover, in next-generation database systems that enforce complex integrity constraints on sets or support on-line decision support, data mining, knowledge bases, or logic programming, the importance of quantification in general and universal quantification in particular must be expected to grow.

The following sections examine algorithms for universal quantification by relational division. We chose the context of the relational data model and its algebra because we assume that the reader is most familiar with that model. However, the algorithms considered here and the conclusions about the algorithms' performance are also applicable to other data models that require universal quantification. Furthermore, the algorithms can serve as a basis for more sophisticated quantification operators such as those explored on a conceptual (as opposed to algorithmic) level by Carlis [1986] and by Whang et al. [1990].

Section 2 gives an overview of algorithms that have been proposed for relational division. A recently proposed algorithm called *hash-division* is discussed in Section 3. In Section 4, we consider modifications of these algorithms using temporary files for inputs and intermediate results larger than main memory. Section 5 describes adaptations of the algorithms to parallel machines, both shared-memory and distributed-memory systems. In Section 6, we illustrate complete query evaluation plans and derive analytical cost formulas for the four algorithms. Section 7 uses the cost formulas for any analytical performance comparison. An experimental comparison follows in Section 8. The final section contains a summary and our conclusions from this research.

2. EXISTING ALGORITHMS

In order to describe the algorithms most clearly, we introduce two example queries that will be used throughout this paper. Assume a university database with two relations, *Course* (*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. As an English query over the relations in the database, this is

> find the students (*student-id's*) in *Transcript* 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

$$\pi_{student\text{-}id,course\text{-}no}(Transcript) \div \pi_{course\text{-}no}(Courses). \quad (1)$$

In this example, the projection of *Transcript* is the **dividend** and the projection of *Courses* is the **divisor**. 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. The set of quotient attribute values in the dividend, $\pi_{student\text{-}id}(Transcript)$ in this example, is called the set of **quotient candidates**. The set of all dividend tuples with the same quotient attribute value is called a **quotient candidate group.**

It is important to notice that both relations are projected on their key attributes in this example. Thus, the problems of duplicate tuples in the dividend or the divisor do not arise. However, if students were permitted to take the same course multiple times, the *Transcript* relation would require an additional key attribute *term* and the projection of *Transcript* on *student-id* and *course-no* would require an explicit duplicate removal step. As we will see, if the inputs of relational division may contain duplicates, either the division algorithm employed must be able to handle them or the inputs must be properly preprocessed. Performing duplicate removal can be quite expensive, making very desirable an algorithm that is insensitive to duplicates in its inputs.

We also assume, for this example, that the *Transcript* relation represents a many-to-many relationship between students and courses. Thus, a referential integrity constraint holds between dividend and divisor, namely that all *course-no* values in the *Transcript* relation also appear in the *Course* relation. It is important that the validity of the referential integrity constraint be known a priori; otherwise, the division algorithm or a preprocessing step (a semi-join) must explicitly enforce it.

As our second example, we are interested in finding the students who have taken all database courses, i.e., courses for which the title attribute contains the string "Database." For this example, the divisor is restricted by a prior selection. Other restrictive operations on the divisor, e.g., a semi-join, would have the same effect as the selection considered here. While the two example queries seem almost identical, the referential integrity constraint assumed above does not hold for the restricted *Courses* relation. In the example, there may very well be transcript entries that do not refer to database courses. This difference has ramifications when division is implemented using aggregations, which will be described shortly.

Table I summarizes the cases that must be considered separately in a complete analysis of relational division algorithms. In order to shorten the discussion, we only consider cases in which either both or none of the two inputs may contain duplicates. Thus, in the analytical and experimental

Table I. Cases to Consider for Relational Division Algorithms

Case	*Violations of referential integrity*	*Duplicates in the dividend*	*Duplicates in the divisor*
1	No	No	No
2	No	No	Yes
3	No	Yes	No
4	No	Yes	Yes
5	Yes	No	No
6	Yes	No	Yes
7	Yes	Yes	No
8	Yes	Yes	Yes

performance comparisons, we only consider four cases, namely cases 1, 4, 5, and 8.

Let us briefly speculate which of these cases will arise most frequently. In practice, a dividend relation will often be a relation representing a many-to-many relationship between two entity types. In other words, many universal quantification queries will find the set of instances of one entity type, e.g., students, that are related to all or a selected subset of instances of another entity type, e.g., courses or database courses. We believe that queries using an entire set or stored relation as divisor are not often useful, e.g., our first example query. Instead, the subset alternative will be used much more frequently, e.g., our second example query. Translated from the E-R-model into the relational model, where relationships are represented with keys and foreign keys, and into the classification of Table I, duplicates in the inputs will often not be a problem, while referential integrity between dividend and divisor can typically not be assumed. On the other hand, the problem of duplicate removal does sometimes arise, e.g., in the modified example above where students are permitted to take the same course twice, requiring an additional key attribute *term* that is not present in the dividend.

Before discussing specific relational division algorithms, we would like to point out that there are some universal quantification queries that seem to require relational division but actually do not. Consider two additional relations *Student* (*student-id*, *name*, *major*) and *Requirement* (*major*, *course-no*) and a query for the students who have taken all courses required for their major. The crucial difference here is that the requirements are different for each student, depending on his or her major. This query can be answered with a sequence of binary matching operations. A join of the *Student* and *Requirement* relations projected on the *student-id* and *course-no* minus the *Transcript* relation an be projected on *student-id*'s to obtain a set of students who have not taken all their requirements. The difference of the *Student* relation with this finds the students who have satisfied all their requirements. The relational algebra expression that describes this plan is

$$\pi_{student\text{-}id}(Student) - \pi_{student\text{-}id}(\pi_{student\text{-}id,course\text{-}no}(Student \bowtie Requirment) - \pi_{student\text{-}id,course\text{-}no}(Transcript)).$$

This sequence of operations will have acceptance performance because it does not contain a Cartesian product operation and its required set matching algorithms (i.e., join and difference) all belong to the family of one-to-one match operations that can be implemented efficiently by any suitably parameterized join algorithm, e.g., hybrid hash join [DeWitt et al. 1986; Shapiro 1986]. In fact, since the difference could be executed by hashing on the *student-id* attributes, these two operations could be executed more efficiently by a single matching operation for three inputs [Graefe 1993].

The following division algorithms do not depend on the existence of indices or other associative access structures that might provide a mechanism for fast or ordered access to tuples. Sort-based indices could be used instead of explicit sort operations in the sort-based algorithms, in particular clustering indices, and hash-based indices could be used to the hash-based algorithms. When suitable index structures are available, they can speed up the affected algorithms, e.g., by rendering a sort operation unnecessary. However, it is not always possible to choose a suitable clustering. For example, keeping the *Transcript* table in our example sorted and clustered on *student-id*'s is useful, but as for any table representing a many-to-many relationship, the alternative sort order on the other part of the table's key (*course-no* in the example) is equally likely to be chosen to enhance join performance for other queries. Any table representing a many-to-many relationship implies the same problem.

Similarly, multi-table clustering (sometimes called "master-detail" clustering) would be useful and could improve the performance of some relational division algorithms. Unfortunately, it is available in only a very few database systems, and it has the same problem for many-to-many relationships as clustering by sort order: while a master-detail relationship is inherently a one-to-many relationship, a table representing a many-to-many relationship can be clustered in two ways. For generality, we therefore do not presume clustered relations.

Moreover, and more importantly, indices typically do not exist on intermediate query results. In order to ensure that our algorithms and analyses apply to both permanent, stored relations and to intermediate query processing results, we discuss and analyze division algorithms without any such advantages. The cost of relational division in database systems and physical database designs that could include suitable structures and orderings can be inferred from the provided cost formulas by subtracting the cost of preprocessing steps such as sorting.

2.1 A Naive Sort-Based Algorithm

The first algorithm 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, the *Transcript* relation 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 reminiscent of both nested loops and merge-join. The dividend serves as outer, the divisor as inner, relation. The dividend is

Student	Course
Jack	Intro to Databases
Jill	Database Performance
Jill	Intro to Databases
Jill	Intro to Graphics
Jill	Readings in Databases
Joe	Database Performance
Joe	Intro to Compilers
Joe	Intro to Databases

Dividend

÷

Course
Database Performance
Intro to Databases
Readings in Databases

Divisor

=

Student
Jill

Quotient

Fig. 1. Sorted inputs into naive division

scanned exactly once, whereas the divisor is scanned once entirely for each quotient tuple, and once partially for each candidate quotient tuple which actually does not participate in the quotient. Differently than in merge-join, however, both scans can be advanced when an equality match has been found. The dividend relation may contain a tuple that does not match with any of the divisor tuples, e.g., a *Transcript* tuple of a graphics course in the second example. This algorithm's scan logic ignores such extraneous records.

Figure 1 shows three tables marked Dividend, Divisor, and Quotient. The dividend and the divisor are sorted properly for naive division. We show string values in Figure 1 because they are easier to read; in a real application, these would typically be identifying keys such as *student-id* and *course-no*. Concurrent scans of the "Jack" tuples in the dividend (there is only one) and of the divisor determine that "Jack" is not part of the quotient because he has not taken the "Database Performance" course. A continuing scan through the "Jill" tuples in the dividend and a new scan of the entire divisor includes "Jill" in the output of the naive division. The fact that "Jill" has also taken an "Intro to Graphics" course is ignored by a suitably general scan logic for naive division. Finally, the "Joe" tuples in the dividend are matched in a new scan of the divisor, and "Joe" is not included in the output.

Essentially this algorithm was proposed by Smith and Chang [1975] and other early papers on algorithms for executing relational algebra expressions. It is the first algorithm analyzed in the performance comparisons later in this article.

SQL Formulations Using Two "NOT EXISTS" Clauses

In the introduction, we claimed that the typical SQL formulation of universal quantification and relational division queries, which employs two negated existential quantifications, leads not only to confusing complexity but also to very poor performance. For example, presuming that "NULL" values are

prohibited, the first example query is Date and Darwen [1993]; and O'Neil [1994]

```
SELECT DISTINCT t1.student-id FROM Transcript t1
WHERE NOT EXISTS (
  SELECT * FROM Course c
  WHERE NOT EXISTS (
    SELECT * FROM Transcript t2
    WHERE t2.student-id = t1.student-id
    AND t2.course-no = c.course-no)).
```

In order to process this SQL query, as many as three nested loops will be used. The outermost one loops over all unique values for *t1.student-id*. The middle one loops over the values for *c. course-no*. Notice that these two loops correspond to the Cartesian product in the relational algebra expression given in the introduction as well as to the two loops in naive division. The innermost loop determines whether or not there is a tuple in *Transcript* with the values of *t1.student-id* and *c. course-no*. Since this loop is essentially a search, an index on either attribute of *Transcript* can be very useful.

If *Transcript* and *Course* are clustered using B-tree indices, many of the index searches can be very fast. In this best case, the execution logic and run-time of the two "NOT EXISTS" clauses will be similar to naive division. Thus, when comparing other algorithms against naive division, we also compare these algorithms with the best case of the typical SQL "work-around" for universal quantification.

2.2 Implementing Division by Aggregation

Since the naive division algorithm requires sorting both inputs and repeated scans of the divisor, it may be rather slow for large inputs, and it seems worthwhile to search for alternative algorithms. One such alternative uses aggregations or, more specifically, counting. In fact, in most relational database management systems, counting is the most efficient way to express for-all predicates. However, it is left to the user to rephrase all universal quantifications into aggregate functions, which is a likely source of errors, e.g., incorrect counts due to the presence of duplicate tuples or the omission of a predicate enforcing referential integrity.

The first example query can be expressed as "find the students who have taken as many (different) courses as there are courses offered by the university (tuples in the *courses* relation)." In SQL, assuming that there are no duplicates in either relation, that all *Transcript* tuples refer to valid *course-no*'s, and that there are no "NULL" values, this query is

```
SELECT t.student-id FROM Transcript t
GROUP BY t.student-id
HAVING COUNT (t.course-no) = (SELECT COUNT (course-no) FROM Course).
```

This query is evaluated in three steps. First, the courses offered by the university are counted using a scalar aggregate operator. This step replaces the subquery expression with a constant. 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.

If the dividend and divisor relations do not contain unique keys as in our example, it would be necessary to explicitly request uniqueness of the *student-id*'s and *course-no*'s counted. For example, some students may take the same database course twice, if the first time resulted in a failing grade. In that case, the *Transcript* relation should include a *term* attribute, which would be part of the key. However, the dividend is the projection of *Transcript* on *student-id* and *course-no*; since part of the key is being removed, duplicates are possible and duplicate removal in the dividend is required before grouping and counting.

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." In SQL, this query can be expressed by

```
SELECT t.student-id FROM Transcript t, Course c
WHERE t.course-no = c.course-no AND c.title LIKE "%Database%"
GROUP BY t.student-id
HAVING COUNT (DISTINCT t.course-no) =
  (SELECT COUNT (DISTINCT course-no) FROM Course
  WHERE c.title LIKE "%Database%").
```

For illustration purposes, we included two "DISTINCT" key words where they would be necessary if duplicates could exist. A good query optimizer would infer key and uniqueness properties and ignore these "DISTINCT" key words where appropriate. More important, however, are the additional "WHERE" clauses. First, since referential integrity between dividend and divisor does not hold, an additional join clause must be specified. Second, the restriction on the divisor (on "%Database%") must be specified on both levels of subquery nesting.

Comparing the query evaluation plans for the two SQL queries, the first and the third steps of the plan above remain virtually the same, but the second step becomes significantly more complex. Since it is important to count only those tuples from the *Transcript* relation that refer to database courses, the aggregate function must be preceded by a semi-join of *Transcript* and *Courses* restricted to database courses.

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

Can All Universal Quantification Queries be Expressed by Counting?

While it seems to work for the example queries, one might ask whether all universal quantification can be expressed by counting. The essential, basic ideas for a proof are that (1) universal quantification and existential quantification are related to each other as each is negated using the other, and (2) existential quantification is equivalent to a count greater than zero. Negated,

existential quantification is equivalent to a count equal to zero. Similarly, universal quantification is equivalent to the count of non-qualifying items equal to zero. This, in turn, is equivalent to a count of qualifying items equal to the maximum possible, i.e., the count of qualifying items is equal to the count of items in the set over which the query is universally quantified. Thus, all universal quantification queries can indeed be rewritten using aggregation and counting.

Since this proof idea is quite simple and very intuitive, we do not develop a formal proof here. In order to capture all subtleties of SQL semantics, "NULL" values must be taken into proper consideration. Because most universal quantification and relational division queries will follow the query pattern given in the introduction, i.e., analyze many-to-many relationships and set-valued attributes, and therefore involve key attributes, we will ignore "NULL" values in the rest of this article.

Division Using Sort-Based Aggregation

The traditional way of implementing aggregate functions relies on sorting [Epstein 1979]. 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 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 [Bitton and DeWitt 1983]. We call this optimization *early aggregation*.

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

Since sort-based aggregation has been used in most database systems, e.g., INGRES [Epstein 1979] and DB2 [Cheng et al. 1985], it is important to understand its performance when used for relational division. Division using sort-based aggregation is the second algorithm analyzed in the performance comparisons below.

Division Using Hash-Based Aggregation

Sorting actually results in more order than necessary for many relational algebra operations. In the examples, it is not truly required that the *Transcript* tuples be rearranged in ascending *student-id* order; it is only necessary that *Transcript* tuples with equal *student-id* attribute values be brought together. The fastest way to achieve this uses hashing. Thus, several hash-based algorithms have been proposed for join, semi-join, intersection, duplicate removal, aggregate functions, etc. (e.g., in Bratbergsengen [1984],

DeWitt et al. [1984], DeWitt and Gerber [1985], Fushimi et al. [1986], Kitsuregawa et al. [1983], Shapiro [1986], Zeller and Gray [1990], and many others). The most efficient technique for inputs larger than memory is hybrid hashing [DeWitt et al. 1984; DeWitt and Gerber 1985; Shapiro 1986] if modified with techniques for managing non-uniform hash value distributions [Kitsuregawa et al. 1989; Nakayama et al. 1988].

Hash-based aggregate functions keep tuples of the output relation in a main memory hash table, but not input tuples. 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 an 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 is consumed, the result of the aggregate function is contained in the hash table.

If the aggregate function output does not fit into main memory, *hash table overflow* occurs and temporary files must be used to resolve it. Since the hash table contains only 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 need 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 based on sorting memory-sized runs using quicksort. The I/O requirements of aggregation based on sorting using replacement selection [Knuth 1973] are similar to those of aggregation based on hashing. If the output size is only slightly larger than memory, hybrid overflow techniques as considered for join operations [DeWitt et al. 1984; Shapiro 1986] can also be used for aggregation and duplicate removal.

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

Let us briefly consider duplicates again. In the naive division algorithm, duplicates in either input relation can be eliminated conveniently as part of the sort operation or, if the inputs are already sorted, by inserting a simple filter that compares consecutive tuples. In sort-based aggregation, the initial sort can be used to remove duplicates, although sorting cannot remove duplicates and aggregate tuples at the same time. In other words, if duplicate removal in the *Transcript* relation is required, the early aggregation can be used for duplicate removal but the actual aggregation (counting *course-no*'s) cannot be integrated into the sort operator. Similarly, hash-based aggregation cannot include duplicate removal, since only one tuple is kept in the hash table for each group. While efficient duplicate removal schemes based on hashing exist, they require that the entire duplicate-free output must be kept in main memory hash tables or in overflow files. Thus, duplicate removal

based on hashing may be expensive for a very large dividend relation, with about the same number of I/O operations as duplicate removal based on sorting and slightly lower CPU costs [Graefe 1993].

Hash-based aggregation has been used only in a small number of systems, e.g., in Gamma [DeWitt et al. 1986; DeWitt et al, 1990], Volcano [Graefe 1994], and Tandem's NonStop SQL [Zeller and Gray 1990]. Division using hash-based aggregation is the third algorithm analyzed in performance comparisons.

3. HASH-DIVISION

This section contains a description of a recently proposed algorithm introduced as *hash-division* in Graefe [1989], followed by a discussion of the algorithm. The design of hash-division was motivated by a search for a direct algorithm for universal quantification and relational division, similar to naive division but based on hashing. Table II shows where the recently proposed algorithm fits into a classification of universal quantification and relational division algorithms.

Algorithm Description

Figure 2 gives pseudo-code for the hash-division algorithm. 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*. An integer value is kept with each tuple in the divisor table, called the *divisor number*. With each tuple in the quotient table, a bit map is kept with one bit for each divisor tuple. Bit maps are common data structures for keeping track of the composition of a set; since the purpose of relational division is to ascertain which quotient candidates are associated with the entire divisor set, bit maps are a natural component of division algorithms.

Figure 3 illustrates the two hash tables used in hash-division. The inputs are the same as those in Figure 1; hash-division, however, does not require that the inputs be sorted. The divisor table on the left contains all divisor tuples and associates a divisor number with each item. The quotient table on the right contains quotient candidates, obtained by projecting dividend tuples on their quotient attributes, and a bit map for each item indicating for each divisor tuples there has been a dividend tuple. The fact that "Jack" and "Joe" have taken only one and two "Database" courses is indicated by their incompletely filled bit maps. The courses on subjects other than databases do not appear in either hash table, because it was immediately determined that there was no compilers and graphics course in the divisor relation

The hash-division algorithm proceeds in three steps. First, it inserts all divisor tuples into the divisor table. The hash bucket is determined by hashing on all attributes. In the process, the algorithm counts the divisor tuples, and assigns the current count as a tuple's divisor number when the tuple is inserted. Thus, a unique divisor number is assigned to each divisor tuple. At the end of this first step, the divisor table is complete, as shown on the left in Figure 3.

Table II. Classification of Universal Quantification Algorithms

	Based on sorting	*Based on hashing*
Direct	*Naive division*	*Hash-division*
Indirect (counting) by semijoin and aggregation	Sorting with duplicate removal, merge-join sorting with aggregation	Hash-based duplicate removal, hybrid hash join, hash-based aggregation

```
// step 1: build the divisor table
assign divisor count ← zero
for each divisor tuple
    insert divisor tuple into the divisor table's appropriate bucket
    assign tuple's divisor number ← divisor count
    increment the divisor count

// step 2: build the quotient table
for each dividend tuple
    scan appropriate hash bucket in the divisor table
    if a matching divisor tuple is found
        scan appropriate hash bucket in the quotient table
        if no matching quotient (candidate) tuple is found
            create new quotient candidate tuple
                from quotient attributes of dividend tuple
                incl. a bit map initialized with zeroes
            insert it into hash bucket of quotient table
        set bit corresponding to divisor tuple's divisor number

// step 3: find result in the quotient table
for each bucket in quotient table
    for each tuple in bucket
        if the associated bit map contains no zero
            print quotient tuple
```

Fig. 2. The hash-division algorithm.

In the second step, the algorithm consumes the dividend relation. For each dividend tuple, the algorithm first checks whether or not the dividend tuple corresponds to a divisor tuple in the divisor table by hashing and matching the dividend tuple on the divisor attributes. If no matching divisor tuple exists, the dividend tuple is immediately discarded. In the second example query, a student's *Transcript* tuple for an "Intro to Graphics" course does not pass this test and is not considered further, and the algorithm advances to the next dividend tuple. 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 or not a matching quotient candi-

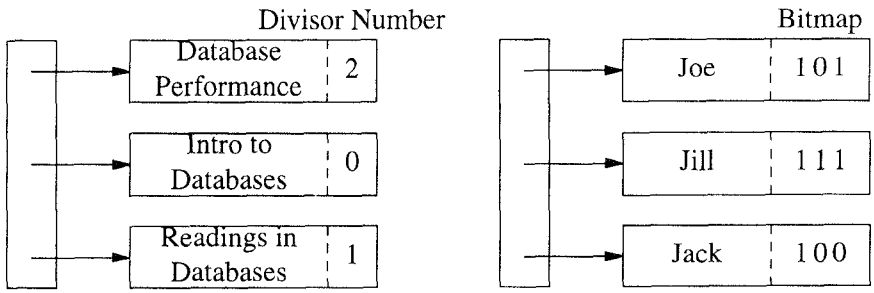


Fig. 3 Divisor table and quotient table in hash division.

date 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. At the end of this second step, the quotient table is complete, as shown on the right in Figure 3, and the divisor table can be discarded.

Finally, the third step determines the quotient of the two inputs, which consists exactly of those tuples in the quotient table for which the bit map contains no zero. This set can easily be determined by scanning all buckets in the quotient table.

Algorithm Discussion

A number of observations can be made on hash division. First, duplicates in the divisor can be eliminated while building the divisor table. In order to do so, the first step of the algorithm must be augmented to test for an existing duplicate in the divisor table's hash bucket before inserting a new tuple into the divisor table, which is the standard technique used for hash-based duplicate removal and aggregate functions.

Second, duplicates in the dividend are ignored automatically. Two identical dividend items map to the same divisor tuple and therefore the same divisor number; moreover, they map to the same quotient candidate, and therefore to exactly the same bit in the same bit map. To appreciate this effect of bit maps, compare this method of detecting and ignoring duplicates in the large input, the dividend, with duplicate removal in the indirect strategy using hash-based duplicate removal. The latter strategy requires building a hash table that contains all fields of all unique tuples from the entire dividend. In hash-division, dividend tuples are split into two sections (vertically partitioned into the divisor and the quotient attributes), and the repetition factors

in each section are expected to be high. The divisor numbers and the bit maps relate entries in these hash tables, which is much more memory-efficient than the large hash table required in hash-based duplicate removal.

Third, it is instructive to compare hash-division against division using hash-based aggregation with prior semi-join. In both cases, there are two hash tables. In hash-division, they are the divisor and quotient tables. In hash-based aggregation with prior semi-join, the first hash table contains the divisor relation for the semi-join. The second hash table, used in the aggregation step, contains the aggregation output, i.e., the quotient candidates. The counter used in the aggregation serves the same function as the bit map employed in hash-division. Bit maps, however, provide the additional functionality that duplicates in the dividend are ignored. If duplicates are known not to be a problem, the hash-division algorithm can be modified to employ counters instead of divisor numbers and bit maps.

Fourth, if the divisor is also free of duplicates and referential integrity holds between dividend and divisor, the divisor table can be eliminated entirely and the bit maps associated with quotient candidate tuples can be replaced by counters. These are precisely the conditions required for division based on aggregation without prior semi-join and duplicate removal, and the resulting hash-division algorithm is, not surprisingly, similar to division using hash-based aggregation.

Fifth, hash-division creates a hash table on its result (the quotient table) that can be used for subsequent operations. For example, consider the example queries to find *student-id*'s of students who have taken a set of classes. Typically, the *student-id*'s themselves are not useful; a subsequent join is needed to attach useful information such as the students' names and majors to these *student-id*'s. The quotient table available at the end of hash-division enables an immediate hash-join with a third input, e.g., the *Student* relation. A fairly simple and efficient modification of the hash-division algorithm discussed so far could permit a third input and effect the subsequent join. Hash-based aggregation implementations typically do not permit retaining the hash table built for the aggregate function across the step that compares the divisor count with the individual counts of quotient candidates. Thus, hash-based aggregation has a clear disadvantage if the quotient is to be joined with a third relation. The sort-based algorithms, naive division and sort-based aggregation, have the useful property of delivering their output sorted on the quotient attributes, e.g., *student-id*'s, which permits an immediate merge-join with a third relation such as *Student* without sorting the quotient. However, as we will see in the performance comparisons, the sort-based division algorithms are not really competitive.

Finally, hash-division depends on sufficient main memory to hold both hash tables. Recall 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 may actually contain more tuples than the quotient, namely all quotient candidates, 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 take advantage of a multi-processor system. In the next section, we consider techniques for handling hash table overflow in a single processor database system. Adaptations of the hash-division and the other algorithms to multi-processor systems are discussed in Section 5.

4. USING TEMPORARY FILES FOR LARGE INPUTS

In this section we consider the modifications required for each of the algorithms discussed in the last two sections if one of the inputs (dividend or divisor) or the quotient candidates do not fit into the available memory. For sort-based algorithms, i.e., naive division and sort-based aggregation and semi-join, the underlying sort operator must perform a disk-based merge-sort as described in the literature, e.g., in Graefe [1993], Knuth [1973], and Salzberg et al. [1990]. For hash-based aggregation and semi-join, standard overflow techniques can be used, e.g., overflow avoidance or fragmentation [Fushimi et al. 1986; Sacco 1986] (possibly combined with bucket tuning and dynamic destaging [Kitsuregawa et al. 1989; Nakayama et al. 1988]) or hybrid hash overflow resolution [DeWitt et al. 1984; DeWitt and Gerber 1985; Shapiro 1986]. In the following, we outline the alternatives for overflow management in hash-division.

If the available memory is not sufficient for divisor table and quotient table, the input data must be partitioned into disjoint subsets called *partitions* that can be processed in multiple *phases*. The partitions are processed one at a time. The first partition is kept in main memory while the other partitions are spooled to temporary files, one for each partition, in a way similar to hybrid hash join [Shapiro 1986]. 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. For the example queries, the set of *student-id*'s would be partitioned, e.g., into odd and even values. Each phase produces a *quotient partition*, which is the quotient of one dividend partition and the divisor. The quotient of the entire division is the concatenation (disjoint union) of all quotients partitions. Translated into the concrete example, the set of students who have taken all database courses is the set of students with odd *student-id*'s who have taken all database courses plus the set of students with even *student-id*'s who have taken all database courses. Since all dividend partitions are dividend with the entire divisor, the divisor table must be kept in main memory during all phases. While this is no problem for small divisors, it certainly can be a problem if the divisor is very large.

The second strategy, called *divisor partitioning*, partitions both the divisor and the dividend relations using the same partitioning function applied to the

divisor attributes. For example, both the *Courses* and the *Transcript* relations are partitioned into undergraduate and graduate courses. Each phase performs the division algorithm for a pair of partition files, one from the dividend and one from the divisor input, producing one quotient partition. Notice that the quotient partitions are quite different for quotient partitioning and divisor partitioning. For divisor partitioning, the quotient partitions 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 partitions, participate in the final result. Indeed, only students who have taken all undergraduate database courses and all graduate database courses have really taken all database courses. This set can easily be determined since the problem of finding the quotient candidates that appear in all quotient partitions is exactly the division problem again. Thus, in order to obtain the final result, each quotient tuple produced by a single-phase division is tagged with the phase number. The collection phase divides (in the sense of relational division) the union (concatenation) of all quotient partitions 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, because the phase number replace the divisor numbers.

Let us compare these two partitioning strategies with overflow and partitioning strategies used when division is executed by hash-based aggregation with a preceding hash-based semi-join. Divisor partitioning is similar to partitioning in the semi-join on the join attribute, *course-no* in the examples. Quotient partitioning is similar to partitioning the input to the aggregate function on the grouping attribute, *student-id* in our examples. Since the methods are somewhat similar in their algorithms and similar partitioning strategies could be used for hash-based aggregation and hash-division, we expect that the two methods exhibit similar performance for small and large files. In fact, the similar partitioning strategies also suggest that similar parallel algorithms can be designed, as we will discuss in the next section.

5. MULTIPROCESSOR IMPLEMENTATIONS

In this section we consider how well the four division algorithms can be adapted for multiprocessor systems. In our discussion, we will include those differences between shared-nothing (distributed memory) and shared-everything (shared memory) architectures [Stonebraker 1986] that go beyond the obvious differences in communication and synchronization overhead.

As for all query processing based on operators and sets, parallelism can be exploited effectively by using program parallelism (pipelining between operators) and data parallelism (partitioning of sets into disjunct subsets). The standard partitioning methods are hash- and range-partitioning; since both forms of partitioning can be combined freely with sort- and hash-based query processing algorithms, we assume here that either one is used such that the partitions are practically equal sizes.

Parallel Naive Division

For naive division, program parallelism can only be used between the two sort operators and the division operator. Both quotient partitioning and divisor partitioning can be employed as described below for hash-division.

Parallel Aggregation Algorithms

For algorithms based on aggregation, both program and data parallelism can be applied using standard techniques for parallel query execution, e.g., those outlined in DeWitt et al. [1990] and Graefe [1993]. While partitioning seems to be a promising approach, there is a problem if a semi-join is needed for referential integrity enforcement. Recall that the join attribute in the semi-join (e.g., *course-no*) is different from the grouping attribute in the subsequent aggregation (e.g., *student-id*). Thus, the large dividend relation (e.g., *Transcript*) may have to be partitioned twice, once for the semi-join and once for the aggregation. Partitioning the large dividend relation twice can be expensive.

Parallel aggregation permits a number of optimizations that are of interest here. First, parallel aggregation does not require that the entire aggregation input be shipped between machines. It is frequently more effective to perform local aggregations first (e.g., count within each existing partition) and then partition and finalize the aggregation (sum local counts). For example, after the *Transcript* and *Courses* relations have been partitioned and semi-joined on *course-no*, each student's number of courses is counted within each partition, and only then are local counts partitioned on *student-id* such that local counts can be summed for each student. The effect is that not the entire dividend, but only a relation about the size of the aggregation output from each machine is partitioned (moved between machines) for the aggregation.

Second, if sort-based aggregation is employed to derive local counts, the dividend partitions shipped across the network will already be sorted. Finalization of the aggregation can take advantage of this fact by merging a partitions and summing the local counts within a particular partition, avoiding another complete sort and aggregation of the partially aggregated input.

Third, a worthwhile idea for hash table overflow resolution in parallel aggregation and duplicate removal is to send overflow buckets to their final sites rather than writing them to overflow files. On each receiving site, the records can be aggregated directly into the existing hash table, possibly without creating any new hash table entries and without requiring any additional memory. Considering that certainly all shared-memory machines but also many modern distributed-memory machines provide faster communication than disk I/O, this seems to be a viable alternative that deserves investigation.

To summarize, division using sort- or hash-based aggregate functions is most likely to be competitive if semi-join is not required. However, if a semi-join is required, the dividend relation must be partitioned and shipped twice across the interconnection network, once for the semi-join and once for

the aggregation, thus significantly increasing the cost in most environments, although the optimization using local aggregation can be used.

Parallel Hash-Division

For hash-division, program parallelism has only limited promise beyond the separation of the division inputs' producers and consumer from the actual division operator, because the entire division is performed within a single operator. However, both partitioning strategies discussed earlier for hash table overflow, i.e., quotient partitioning and divisor partitioning, can also be employed to parallelize hash-division.

For hash-division with quotient partitioning, the divisor table must be *replicated* in the memories of all participating processors. After replication, all local hash-division operators work completely independently of each other, and each local division result will immediately be part of the global division result. Clearly, replication is trivial in a shared-memory machine, in particular since the divisor table can be shared without synchronization among multiple processes once it is complete.

In divisor partitioning, the resulting partitions are processed in parallel instead of in phases as discussed for hash table overflow. However, 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 case that the central collection site is a bottleneck, the collection step can be decentralized using quotient partitioning.

6. QUERY EVALUATION PLANS AND COST FORMULAS

Beginning with this section, we analyze the performance of the four strategies for universal quantification and relational division. This comparison is very detailed in order to support our claim that the power of universal quantification and relational division does not imply poor performance for non-trivial database sizes. In fact, if division algorithms are appropriately implemented and chosen by the query optimizer, universal quantification and relational division are as fast as the simple and efficient hybrid hash join and can therefore be included in database query languages without creating a performance or throughput problem.

In this section, we illustrate query evaluation plans and develop their cost formulas for the algorithms discussed above, including their variants providing duplicate removal or referential integrity enforcement. Some readers may find this detailed analysis tedious—we recommend that those readers skip over the cost functions, as well as their derivations and explanations, and focus solely on the query evaluation plans in this section. For each of the four relational division algorithms, we show up to four query evaluation plans for the cases indicated earlier in Table I. Tuples flow from the leaves to the roots of the plans, and the data paths between operators are labeled with the number of tuples traveling each path.

Table III. Variables Used in the Analysis

Variable	*Description*
$\|R\|$	Dividend cardinality
r	Dividend size in pages
$\|S\|$	Divisor cardinality
s	Divisor size in pages
$\|Q\|$	Quotient cardinality
q	Quotient size in pages
$\|C\|$	Quotient candidate cardinality
c	Quotient candidate size in pages
m	Memory size in pages
α	Size reduction of divisor by duplicate removal
β	Size reduction of dividend by duplicate removal
δ	Size reduction of dividend by referential integrity enforcement
P	Degree of parallelism

We assume dividend relation R ($|R|$ tuples in r pages) and divisor relation S ($|S|$ tuples in s pages) with quotient relation Q ($|Q|$ tuples in q pages). The projection of R on the quotient attributes is the quotient candidate relation C ($|C|$ tuples in c pages). Obviously, $|Q| \leq |C| \leq |R|$ and $q \leq c \leq r$. Further, we assume m memory pages, with $s + c \leq m$. In other words, we only analyze cases in which divisor and quotient are smaller than memory. Given that the dividend relation is the Cartesian product of quotient and divisor, this restriction will cover most practical cases. All variables are summarized in Table III, together with those used in the analyses of duplicate removal, referential integrity enforcement, and parallelism.

Our cost measure combines CPU and I/O costs, both measured as times without overlap of CPU and I/O activity. The cost formulas capture all essential operations such as I/O operations and record comparisons. We tried to roughly match a contemporary workstation when setting the weights of the cost units. The cost units, their values (in milliseconds), and their description are given in Table IV.

For each of the four algorithms (naive division, division by sort-based aggregation and semi-join, division by hash-based aggregation and semi-join, and hash-division), we first derive cost formulas for the case that neither duplicate removal nor referential integrity enforcement are required and then also analyze the cost for these preprocessing steps. However, we first estimate some common costs.

Costs for Original Inputs and Final Output

The cost of reading the divisor and dividend relations, assembling pages of the quotient file (copying), and writing the quotient is

$$(r + s) \times SeqIO + (|R| + |S|) \times RdRec + q \times Move + |Q| \times WrRec + q \times SeqIO. \quad (2)$$

Table IV. Cost Units

Unit	*Time [ms]*	*Description*
RndIO	20	random I/O, one page from or to disk (incl. seek and latency)
SeqIO	10	sequential I/O, one page from or to disk (incl. latency)
RdRec	0.2	CPU cost per record when reading a permanent file
WrRec	0.2	CPU cost per record when writing a permanent file
RdTmp	0.1	CPU cost per record when reading a temporary file
WrTmp	0 1	CPU cost per record when writing a temporary file
Comp	0.03	comparison of two tuples
Aggr	0.03	initializing aggregation, or aggregation 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
Map	0.0001	initializing or scanning a bit in a bit map
SMXfer	0.1	transfer of one page between processes in shared memory
DMXfer	4	transfer of one page in distributed memory

We omit this cost in the following cost formulas, because it is common to all algorithms and does not contribute to their comparison. In the analytic performance evaluation in the following section, we include this base cost in the diagrams to provide a reference point for the algorithms' costs.

Sort Costs

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

$$Sort(S) := 2 \times |S| \times \log_2 (|S|) \times Comp + s \times Move \tag{3}$$

for relation S since it fits in memory.

For relations larger than memory, we assume a disk-based merge-sort algorithm with fan-in F, in which runs are written with sequential I/O and read with random I/O. Its sort cost is the sum of sorting the initial runs using quicksort one memory load at a time and the product of the number of merge passes with the cost of each merge, both I/O and CPU costs, which is

$$\begin{aligned} Sort(R) := {} & 2 \times |R| \times \log_2 (|R|/(r/m)) \times Comp + \log_F (r/m) \\ & \times (r \times (Move + SeqIO + RndIO) + |R| \\ & \times (WrTmp + RdTmp + \log_2 (F) \times Comp)). \end{aligned} \tag{4}$$

Quicksort will be invoked r/m times, each time for $|R|/(r/m)$ tuples. The merge depth is approximated by the logarithm without ceiling to reflect the advantages of optimized merging [Graefe 1993]. The cost of writing a run, e.g., an initial run, is included in the cost of reading and merging it.

6.1 Naive Division

Figure 4 shows three query evaluation plans for relational division using naive division. The labels on the data paths between operators indicate the cardinalities of intermediate results.

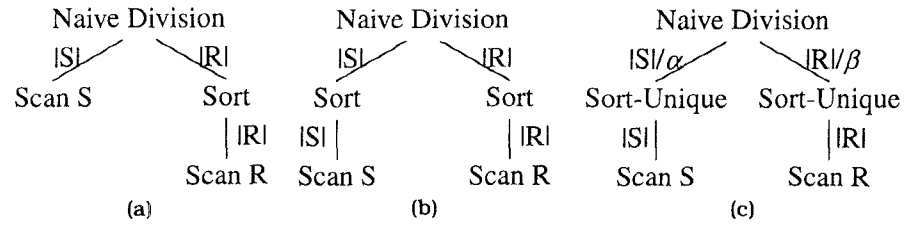


Fig. 4(a–c). Query plans for naive division.

If referential integrity is known to hold for the inputs and duplicate removal is not required, sorting the divisor can actually be omitted and replaced by counting. It is only necessary to sort and count the tuples with equation quotient attribute in the dividend R. This plan is shown in Figure 4a. The cost of naive division algorithm in this case is

$$Sort(R) + |R| \times (Comp + Aggr) + |S| \times Aggr + |C| \times Comp. \quad (5)$$

This formula reflects the costs of sorting R, dividing R into quotient candidates groups, counting the size of each group, counting the divisor tuples, and comparing the size of each group to the number of divisor tuples. Note that this plan is quite similar to the plan using sort-based aggregation for the same case, which will be discussed shortly.

Referential Integrity Enforcement

Naive division can enforce referential integrity as part of its nested merging scans, if the two inputs are sorted properly, e.g., as shown in Figure 1. The plan in Figure 4b shows naive division with two sort operations. The cost of this plan is

$$\begin{aligned} &Sort(S) + Sort(R) + |R| \times Comp + |Q| \times |S| \times Comp \\ &\quad + (|C| - |Q|) \times |S|/2 \times Comp. \end{aligned} \quad (6)$$

The new cost components reflect sorting divisor S in addition to dividend R, comparing consecutive R tuples to divide the sorted dividend into quotient candidate groups, and comparing R and S tuples for referential integrity enforcement. The latter cost is calculated separately for successful and unsuccessful quotient candidates. We presume that half of the divisor file is scanned for each unsuccessful quotient candidate before the failure is identified and the remaining dividend tuples of the quotient candidate are skipped over without pursuing the scan in the divisor. Note that we assumed that $s < m$; therefore, the repeated scans of the divisor can be performed without I/O.

Duplicate Removal

If the inputs contain duplicates, the sort operations must identify and remove them, which requires comparisons, costing $(|R| + |S|) \times Comp$. We presume

that duplicate removal reduces the divisor size by a factor α and the dividend by a factor β, as shown in Figure 4c. Thus, the cost for naive division with duplicate removal but without referential integrity enforcement is

$$Sort(S) + Sort(R) + (|R| + |S|) \times Comp + |R|/\beta \times (Comp + Aggr) + |S|/\alpha \times Aggr + |C| \times Comp. \quad (7)$$

The cost for naive division with both duplicate removal and referential integrity enforcement is

$$Sort(S) + Sort(R) + (|R| + |S|) \times Comp + |R|/\beta \times Comp + |Q| \times |S|/\alpha \times Comp + (|C| - |Q|) \times |S|/\alpha/2 \times Comp. \quad (8)$$

6.2 Division by Sort-Based Aggregation

Performing a division by a sort-based aggregate function requires counting the elements in S, sorting and counting groups in R, and verifying that the counts for the groups of R equal the count of S. Figure 5a (upper left) shows the query evaluation plan. Determining the scalar aggregate, i.e., counting the cardinality of the divisor, and verifying that the correct count was found for each group in the aggregate function costs $|S| \times Aggr + |C| \times Comp$. If we assume for simplicity that the aggregate function on R is formed in the output procedure of the sort's final merge step, the costs of comparing each tuple of R with its predecessor and the actual aggregation must be added to the cost of sorting, i.e., $|R| \times (Comp + Aggr)$. Thus, the total cost for division by sort-based aggregation is

$$Sort(R) + |R| \times (Comp + Aggr) + |S| \times Aggr + |C| \times Comp. \quad (9)$$

Referential Integrity Enforcement

For sort-based referential integrity enforcement, the inputs must be sorted on the divisor attributes and followed by a semi-join version of merge-join. Figure 5b (upper right) shows the corresponding query evaluation plan. The reduction factor of referential integrity enforcement is indicated by δ. We presume that the scalar aggregate can be performed as a side-effect of the sort or at least can obtain its input from the sort; thus, we do not count the cost of two divisor scans, only of one. Thus, only the costs of two sort operations and of a merge-join must be added, which is $(|R| + |S|) \times Comp$.

We assume that the entire S relation is kept in buffer memory during sort and join processing and ignore any cost of memory-to-memory copying in the merge-join since the join is actually a semi-join, which can be implemented without copying. Thus, the cost for sort-based aggregation with referential integrity enforcement is

$$Sort(S) + Sort(R) + (|R| + |S|) \times Comp + Sort(R/\delta) + |R|/\delta \times (Comp + Aggr) + |S| \times Aggr + |C| \times Comp. \quad (10)$$

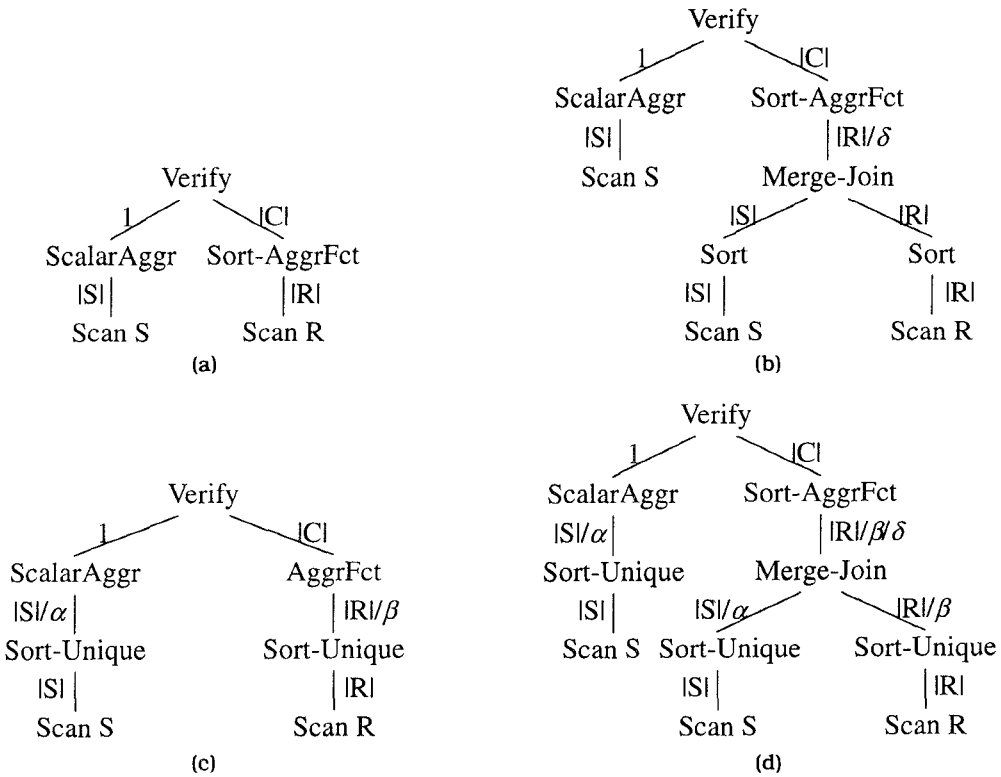


Fig. 5(a–d). Query plans for division by sort-based aggregations.

Duplicate Removal

When the dividend and divisor contain duplicate records, duplicate removal must be performed prior to aggregation in the aggregation-based algorithms. Figure 5c-d (lower left and right) show the query plans including duplicate removal. A sort operator can perform either duplicate removal or an aggregate function, but not both. Thus, the figures show a separate aggregate function, which relies on sorted input and thus can identify groups by simply comparing tuples immediately following each other in the input stream. The common subexpression (sorting and duplicate removal from S) is shown twice but charged only once in the following formulas. The cost of sort-based aggregation with duplicate removal but without referential integrity enforcement is

$$Sort(S) + Sort(R) + (|S| + |R|) \times Comp + |R|/\beta \times (Comp + Aggr) + |S|/\alpha \times Aggr + |C| \times Comp. \quad (11)$$

The cost of sort-based aggregation with both duplicate removal and referential integrity enforcement is

$$\begin{aligned} &Sort(S) + Sort(R) + (|R| + |S|) \times Comp + (|R|/\beta + |S|/\alpha) \times Comp \\ &\quad + Sort(R/\beta/\delta) + |R|/\beta/\delta \times (Comp + Aggr) \\ &\quad + |S|/\alpha \times Aggr + |C| \times Comp. \end{aligned} \tag{12}$$

6.3 Division by Hash-Based Aggregation

The query plan for hash-based aggregation, shown in Figure 6a (upper left), is similar to the one for sort-based aggregation. The cost of hash-based aggregation in memory is $|R| \times (Hash + hc \times Comp + Aggr)$, where hc is the average number of comparisons required to traverse each hash bucket. Recall that we assumed that $s + q \leq m$, meaning that no hash table overflow occurs in the aggregate function. The cost of finding the scalar aggregate and verifying that the same value was found for each group in the aggregate function is the same as for sort-based aggregation. Thus, the cost for the plan in Figure 6a is

$$|R| \times (Hash + hc \times Comp + Aggr) + |S| \times Aggr + |C| \times Comp. \tag{13}$$

Referential Integrity Enforcement

For hash-based aggregation, the additional cost of a prior semi-join as shown in Figure 6b (upper right) is $|S| \times Hash + |R| \times (Hash + hc \times Comp)$ for building a hash table with the tuples from S and probing it with the tuples from R. Thus, the total cost for division by hash-based aggregation and hash-based semi-join for referential integrity enforcement is

$$\begin{aligned} &|S| \times Hash + |R| \times (Hash + hc \times Comp) + |R|/\delta \\ &\quad \times (Hash + hc \times Comp + Aggr) + |S| \times Aggr + |C| \times Comp. \end{aligned} \tag{14}$$

Duplicate Removal

The next two figures show the previous two query evaluation plans modified to include duplicate removal. If the implementation of hash-based duplicate removal were very smart, it would be possible to ensure that the duplicate removal operator in Figure 6c (lower left) produces its output grouped suitably for the subsequent aggregate function, making a separate hash table in the aggregate function obsolete. Similarly, in Figure 6d (lower right), the duplicate removal operation for the divisor could be integrated into the hybrid hash join operation. However, we ignore these optimizations in our cost formulas.

Since the duplicate-free dividend may be larger than memory, i.e., $|R|/\beta > m$, hash table overflow may occur in the duplicate removal operation. In this case, recursive and hybrid hashing must be employed. The recursion depth can be approximated by the logarithm of the duplicate removal output size divided by the memory size, with the logarithm based equal to the partitioning fan-out, which we assume to be equal to the fan-in during merge-sort F [Graefe 1993; Graefe et al. 1994]. Thus the cost of hash-based duplicate

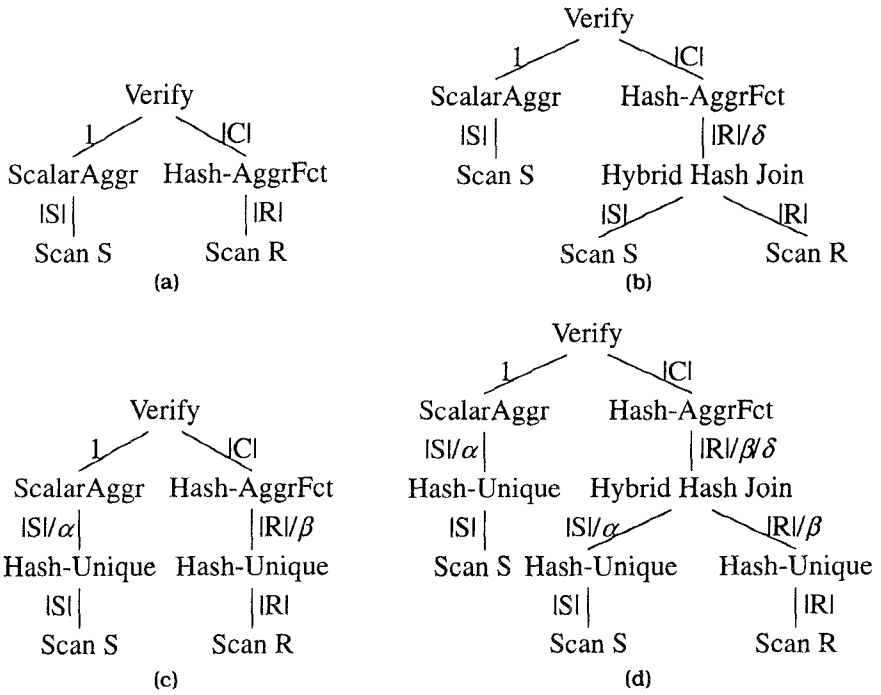


Fig. 6(a–d). Query plans for division by hash-based aggregations.

removal for the large dividend is—considering the actual duplicate removal step in the deepest recursion level first and calculating the cost of (sequentially) reading a partition file together with the cost of writing it (using random writes):

$$\begin{aligned} HashUnique(R) := {} & |R| \times (Hash + hc \times Comp) \\ & + \log_F (r/m) \times (|R| \times Hash + |R| \\ & \times (WrTmp + RdTmp) + r \times (RndIO + SeqIO)). \end{aligned} \tag{15}$$

Thus, the cost of relational division using hash-based aggregation including duplicate removal on the inputs but not referential integrity enforcement is

$$\begin{aligned} & HashUnique(R) + |S| \times (Hash + hc \times Comp) \\ & \quad + |R|/\beta \times (Hash + hc \times Comp + Aggr) + |S|/\alpha \times Aggr + |C| \times Comp. \end{aligned} \tag{16}$$

If both duplicate removal and referential integrity enforcement are required, the cost is

$$\begin{aligned} HashUnique(R) & + |S| \times (Hash + hc \times Comp) + |S|/\alpha \times Hash \\ & + |R|/\beta \times (Hash + hc \times Comp) \\ & + |R|/\beta/\delta \times (Hash + hc \times Comp + Aggr) \\ & + |S|/\alpha \times Aggr + |C| \times Comp. \end{aligned} \tag{17}$$

6.4 Hash-Division

When hash-division is used, a prior semi-join for referential integrity enforcement or an explicit duplicate removal step is never necessary. Thus, the query evaluation plan shown in Figure 7 always applies, although the actual hash-division algorithm might have the divisor table and the bit maps enabled or disabled. If it is known that neither duplicate removal nor referential integrity enforcement is required, the simplest version of the algorithm can be used. This version mimics the behavior of division by hash-based aggregation without duplicate removal or referential integrity enforcement. It uses neither divisor table nor bit maps and just counts divisor tuples and dividend tuples by groups. The cost of this variant of hash-division is

$$|S| \times Aggr + |R| \times (Hash + hc \times Comp + Aggr), \tag{18}$$

which is the sum of the costs of counting the divisor tuples and of inserting and counting in the quotient table. We use this cost formula although the counting is done very efficiently with program variables of the hash-division algorithm, not as a more general interpretation of a query evaluation plan as in hash-based aggregation, which may invoke a relatively expensive abstract data type facility, even for tasks as simple as counting.

Referential Integrity Enforcement

If, however, the validity of the referential integrity constraint is not known, a divisor table must be used. In this case, the cost of hash-division is

$$\begin{aligned} &|S| \times Hash + |R| \times (Hash + hc \times Comp) + |R|/\delta \\ &\quad \times (Hash + hc \times Comp + Aggr). \end{aligned} \tag{19}$$

The differences to the cost for hash-division without referential integrity enforcement reflect building and probing the divisor table.

Duplicate Removal

If the inputs contain duplicates, the divisor table must be probed while it is being built and the quotient table must contain bit vectors instead of counters, with $|S|/\alpha$ bits per bit vector. If duplicates exist but referential integrity is known to hold, the divisor table need not be probed with dividend tuples, and the cost of hash-division is

$$\begin{aligned} &|S| \times (Hash + hc \times Comp) + |R| \times (Hash + hc \times Comp + Bit) \\ &\quad + 2 \times |C| \times |S|/\alpha \times Map. \end{aligned} \tag{20}$$

The first two terms account for building the divisor table and the quotient table. The third term reflects initializing and scanning the bit maps.

The cost of the most powerful variant of hash-division algorithm, which includes both duplicate removal and referential integrity enforcement, is

$$\begin{aligned} &(|S| + |R|) \times (Hash + hc \times Comp) + |R|/\delta \times (Hash + hc \times Comp + Bit) \\ &\quad + 2 \times |C| \times |S|/\alpha \times Map. \end{aligned} \tag{21}$$

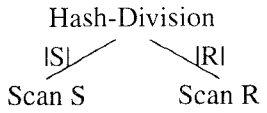


Fig. 7. Query plan for hash division.

The first term reflects probing the divisor table, both by the divisor while building the divisor table and by the dividend. The second term accounts for probing and building the quotient table, which includes setting bits in the bit maps, which can be very fast if it is done word by word, not bit by bit.

6.5 Parallel Relational Division

The four algorithms for relational division and all their variants can readily be parallelized with both pipelining and partitioning, i.e., inter-operator and intra-operator parallelism, as discussed earlier. Figure 8 shows the most complex plan for each of the algorithms, with datapaths requiring data repartitioning and replication marked by "part." and "repl." For the direct algorithms, i.e., naive division and hash-division, we assume quotient partitioning, which requires that the divisor be broadcast to all sites. For the component algorithms of the indirect strategies, i.e., duplicate removal, semi-join, and aggregation, we assume partitioning on the join attributes or the grouping attributes. Moreover, we assume a shared-nothing (distributed-memory) machine with local disk drives or a shared-everything (shared-memory) machine with disk drives dedicated to processors in order to reduce contention and interference. In any case, we assume that the available bandwidth is sufficient that communication can occur in parallel between any pairs of sites; thus, communication delay can be estimated by determining the amount of data sent from each site to other processors. We also assume that dividend and divisor are partitioned originally across all disks but that the partitioning cannot be used for processing (e.g., partitioning is round-robin).

In order to determine the cost of each plan, the cost of the underlying sequential plan is increased by the cost of repartitioning data. If P is the degree of parallelism, some data remain with the same processor, namely the fraction $1/P$. The cost of exchanging data across process and machine boundaries for a relation R is $r \times Xfer$, where $Xfer$ should reflect shared memory or distributed memory, i.e., either $Xfer = SMXfer$ or $Xfer = DMXfer$ given in Table IV. Thus, the costs of naive division and of hash-division using quotient partitioning are increased by

$$(r + B \times s) \times (1 - 1/P) \times Xfer, \tag{22}$$

where B represents the relative cost of broadcasting the divisor versus sending it to only one recipient ($B = 1$ for full broadcast support, $B = P$ for no support). The same cost is added for aggregation-based algorithms that do not enforce referential integrity explicitly by means of a semi-join. Parallel

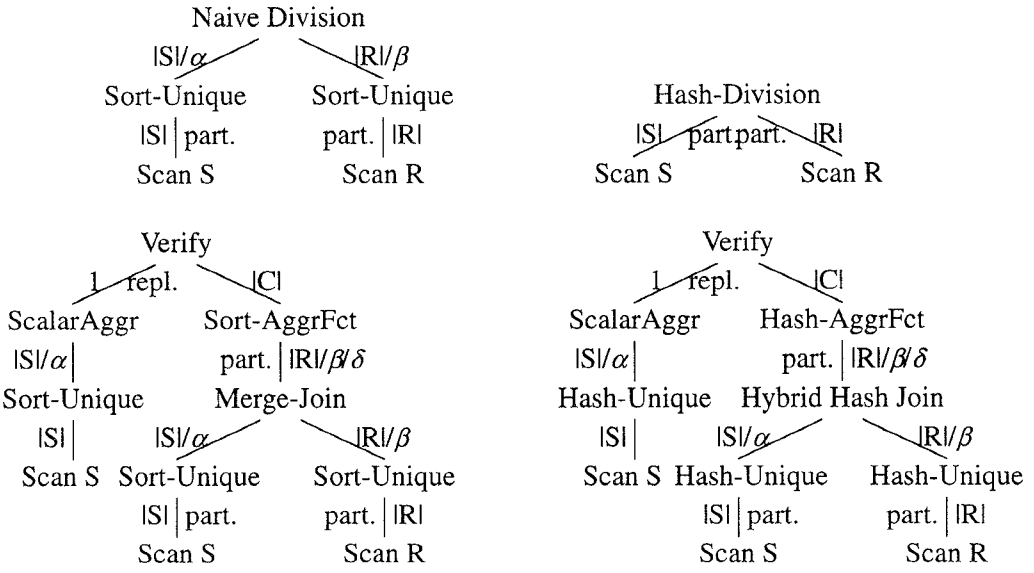


Fig. 8. Parallel query plans for relational division.

aggregation-based algorithms with referential integrity enforcement require repartitioning the intermediate result, and their costs are increased by

$$r/\beta/\delta \times (1 - 1/P) \times Xfer. \tag{23}$$

This can be reduced by splitting the aggregation operations into local and global components, as discussed before. In this case, the data transfer cost for aggregation is about

$$P \times c \times (1 - 1/P) \times Xfer. \tag{24}$$

Of course, all costs are incurred at multiple sites such that the response time actually decreases even if the total resource consumption increases. Moreover, the cost formulas derived above must be modified to reflect the smaller dividend size at each site and the local, not the global, memory size.

For divisor partitioning, suitable plans and their cost can easily be found by modifying the broadcast to repartitioning (remove factor B above) and adding a final collection operator and its cost (division by the set of node identifiers) to each plan. We omit illustrations and cost formulas here.

7. ANALYTICAL PERFORMANCE COMPARISONS

In this section we compare the algorithms' performance by applying the cost formulas derived above to various input sizes and to situations with and without explicit referential integrity enforcement and duplicate removal. In this analysis, we assume that the tuple size for R is 32 B and for S and Q 16 B. We chose fairly small tuples because in most practical cases, the tuples

will consist of keys only as in our examples with *student-id*'s and *course-no*'s. The memory used for sorting or hash tables is 1,024 pages of 4 KB each, which is a realistic size for a single operator of a single query on a machine used to serve many clients. The fan-in for merging and the fan-out for partitioning is 16. The average number of comparisons required for scanning a hash bucket is $hc = 2$. Furthermore, we assume that neither R nor S are sorted originally.

7.1 Sequential Algorithms

Since there is no typical or average case for division algorithms, the conceptually simplest case will be our first comparison: the dividend is the cross-product of the divisor and quotient, i.e., $R = Q \times S$, and neither duplicate removal nor referential integrity enforcement are required.

Figure 9a shows the performance of the four division algorithms for $|S|$ varying from 16 to 4,096 with constant $|Q| = 256$. Since $R = Q \times S$ for all data points, $|R|$ varies from 4,096 to 1,048,576. The bottom label in Figure 9a, as in all subsequent figures, explains the varying parameter in the first line and other pertinent parameters in the second line. The unmarked curve indicates the base cost, i.e., the cost of reading the inputs and writing the output, as previously mentioned when this cost was derived in the previous section. The base cost is included in each algorithm's cost.

It is immediately obvious that the two sort-based algorithms perform almost alike; their only difference is the small cost for quick-sorting the divisor. The two hash-based algorithms are also alike in their performance. The reason is that in all algorithms, the treatment of the dividend relation R dominates the cost, and both sort-based algorithms require sorting R while both hash-based algorithms require probing a hash table. However, the hash-based algorithms outperform the sort-based algorithms by quite a large margin (a factor of 2 to 5) and most of their costs are base costs, i.e., scanning inputs (dividend and divisor) and writing the final output to disk (the quotient). Moreover, the sort-based algorithms' performance deteriorates as $|S|$ and $|R|$ grow while the hash-based algorithms consistently perform well relative to the base cost. The reasons are that hash-aggregation can proceed without overflow whereas naive division and sort-aggregation must use run files, and that hashing permits direct access to the correct hash bucket whereas sorting requires $N \log N$ comparisons.

Referential Integrity Enforcement

Figure 9b shows the analytical cost of division when the referential integrity constraint is explicitly enforced. The cardinalities of the dividend, divisor, and quotient are $|Q| = 256$, $|S| = 256$, and $|R| = |Q| \times |S| \times \delta$. In other words, this comparison expands upon the midpoint of the previous comparison shown in Figure 9a with the size of the dividend relation increased by a factor which appears as reduction factor δ in the cost functions for query evaluation plans with explicit referential integrity enforcement. For example, if $\delta = 1$, then $|R| = 65{,}536$, and when $\delta = 8$, then $|R| = 524{,}288$.

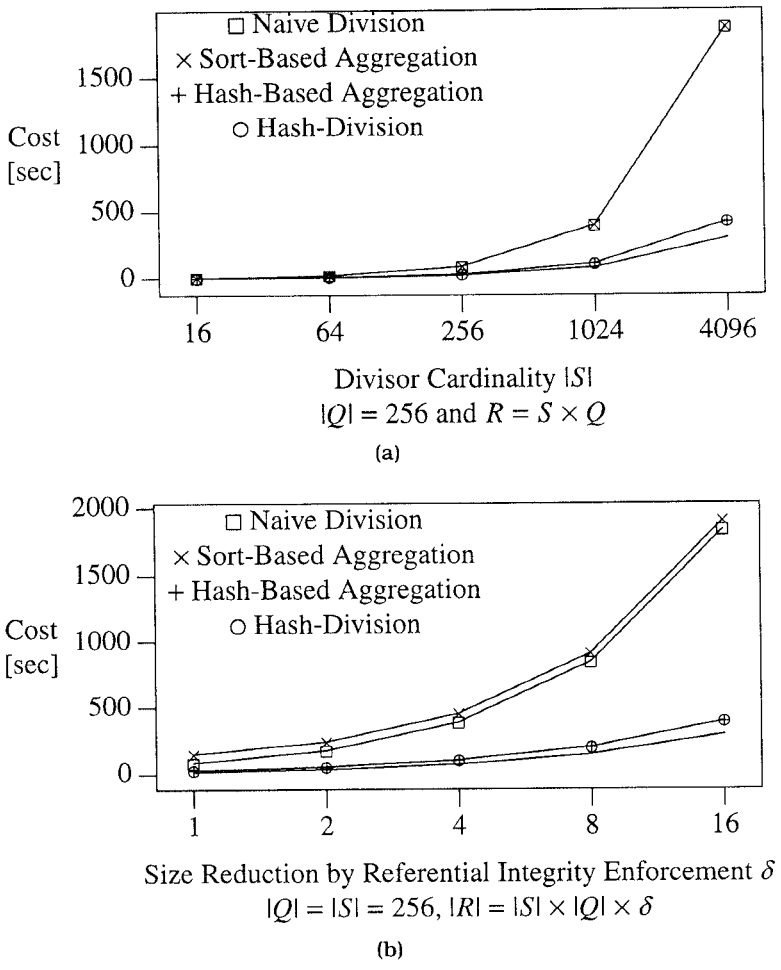
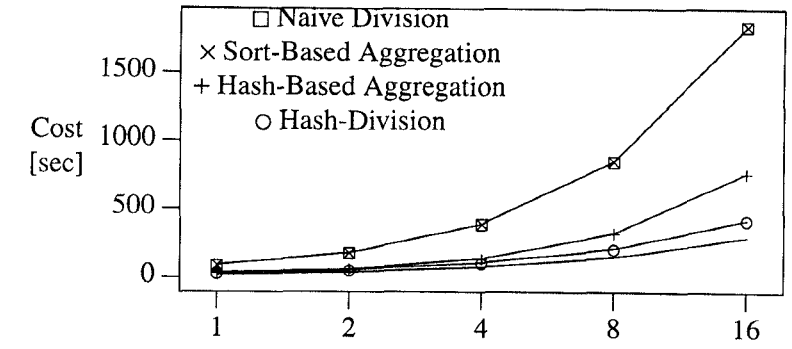


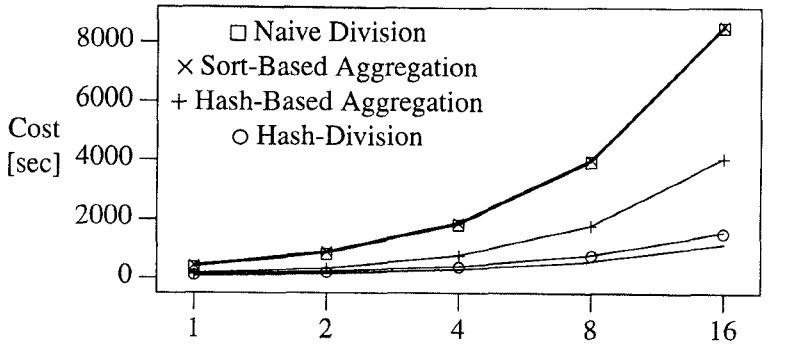
Fig. 9. (a) Cost of division algorithms. (b) Division with referential integrity enforcement.

When the referential integrity constraint is explicitly enforced, as shown in Figure 9b, the naive-division algorithm increases in a familiar N log N pattern with the size of the dividend before the referential integrity enforcement and size reduction.

The cost of sort-based aggregation significantly varies within Figure 9b because of two sort operations for the semi-join and the aggregation. For $\delta = 1$, i.e., if the semi-join does not remove any tuples, the cost of sort-based aggregation with explicit referential integrity enforcement is almost twice that of naive division if the base cost is ignored. Thus, if referential integrity must be enforced, sort-based aggregation is a particularly poor query evaluation plan, although it is the only one available in most current systems. For larger values of δ, the semi-join removes a large fraction of the dividend before the second dividend sort, and the second sort becomes relatively



(c)



(d)

Fig. 9. (c) Division with duplicate removal. (d) Division with duplicate removal and referential integrity enforcement.

insignificant compared to the first dividend sort. Since the first dividend sort has the same input size and the same cost as the sort of the dividend required for naive division, the costs of the sort-based direct and indirect algorithms become comparable for large values of δ, although the sort-based algorithms are still much more expensive than the hash-based division algorithms.

In hash-based aggregation with referential integrity enforcement, the semi-join reduces the number of dividend tuples by a factor of δ for a relatively small cost. In hash-division, the algorithm's divisor table is enabled, which has exactly the same effect as a hash-based semi-join. Thus, for larger values of δ, the overhead of hash-division relative to the base cost actually decreases very slowly.

The hash-based algorithms have a significant performance advantage over their sort-based counterparts: because none of the situations illustrated in

Figure 9b causes hash table overflow, I/O to temporary files is avoided. The reduction of the larger input R can be achieved by filtering it through the in-memory hash table built on S. This is an example of the superiority of hash-based join algorithms for different sizes of the two inputs also observed in earlier research [Bratbergsengen 1984; Graefe et al. 1994].

Duplicate Removal

Figure 9c shows the costs of the four relational division algorithms when the inputs require duplicate removal. For the duplication factors α and β, we assumed that $\alpha = \beta$ and varied their value from 1 to 16.

Again we see that the sort-based algorithms are more expensive because of the duplicate removal processing prior to aggregation. In addition, the cost of hash-based aggregation can now be discerned from that of hash-division. This is because the cost of duplicate removal prior to aggregation is quite high, requiring recursive processing since the duplicate-free dividend can no longer be held by the in-memory hash table. Hash-division continues to be the fastest algorithm. Duplicates in the inputs only require enabling of the quotient table bit map processing, at only a small additional cost.

Figure 9d shows the midpoint of Figure 9c ($\alpha = \beta = 4$) augmented with referential integrity enforcement. The reduction factor for the dividend relation δ varies from 1 to 16. Naive division outperforms sort-based aggregation if referential integrity enforcement is required, as had been observed earlier in Figure 9b. As the referential integrity reduction factor increases and the input to the second dividend sort in sort-based aggregation becomes quite small compared to the input to the first dividend sort, the difference between naive division and sort-based aggregation all but vanishes. With increasing sizes of the original dividend relation, the cost of hash-based duplicate removal becomes larger and larger. However, it does not approach the cost of sort-based aggregation. The performance of hash-division is very steady. Over the entire range, the cost of hash-division relative to the base cost is constant. For both duplicate removal and referential integrity reduction factors equal to 4, hash-division is almost 5 times faster than naive division, a little more than 5 times faster than sort-based aggregation, and $2\frac{1}{2}$ times faster than hash-based aggregation.

Preliminary Conclusions

Since we realize that the formulas are not precise, we will also report on experimental measurements in the next section. Nevertheless, we believe that these analytical comparisons already permit some conclusions.

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 dividend tuples are counted, the additional sort cost makes division by aggregation significantly more expensive.

Second, if division is executed by aggregation, the division can be performed much faster if hash-based algorithms are used for the aggregation and semi-join, because the two input sizes, dividend and divisor, are very

different. This parallels the observations by Bratbergsengen and others on sort- and hash-based matching operators (join, intersection, etc.) [Bratbergsengen 1984; Graefe et al. 1994; Schneider and DeWitt 1989; Shapiro 1986]. In sort-based division algorithms, the number of merge levels is determined for each file individually by the file's size. Therefore, sorting the large dividend with multiple merge levels dominates the cost for both the aggregation and the join. In hash-based algorithms, on the other hand, the recursion depth of overflow avoidance or resolution is equal for both inputs and determined by the smaller one, in our case the relatively small divisor relation. Since the inputs of the semi-join, dividend and divisor, differ by a factor of at least the cardinality of the quotient, using a hash-based semi-join operator is significantly faster. This conclusion can also be stated in a different way: if a database system's only means to evaluate universal quantification and relational division queries is aggregation with prior duplicate removal and semi-join, the semi-join as well as the aggregation should be hash-based because in the case of relational division, the semi-join inputs have very different sizes and therefore give hash-based join algorithms a significant performance advantage.

Third, the new hash-division algorithm consistently performs as well as or better than division by hash-based aggregation. However, if a semi-join or duplicate removal is needed, hash-division significantly outperforms division by hash-based aggregation as well as both sort-based algorithms.

Finally, hash-based algorithms outperform sort-based ones not only for existential quantification but also for universal quantification problems and algorithms. This result is different from the earlier observation cited above because those papers pertained only to one-to-many relationship operations such as join, semi-join, intersection, difference, and union, while this paper is the first to demonstrate this fact for relational division. Most importantly, it outperforms sort-based techniques consistently in a wide variety of situations.

Effect of Clustering Indices

Let us briefly consider the effect of clustering B-tree indices, despite our general focus on algorithms that work on all relations, including intermediate results. There are two effects that are interesting here. First, clustering B-tree indices may save sort operations. Second, if an index contains all attributes relevant for a query, the index may be scanned instead of the base file, which permits significant I/O savings if index entries are shorter than base file records.

The latter effect is not pertinent here, because we assumed very short record sizes in our analysis. The former effect, however, is worth analyzing, which we do in Figure 10. For this comparison, we modified the last comparison (Figure 9d) to presume sorted relations suitable for the first processing step. Thus, the sort-based plans save the initial sort step for each input, whereas the cost of the hash-based algorithms remains unchanged.

Most strikingly, hash-based aggregation is not competitive with the outer three algorithms, because it requires temporary files for hash table overflow.

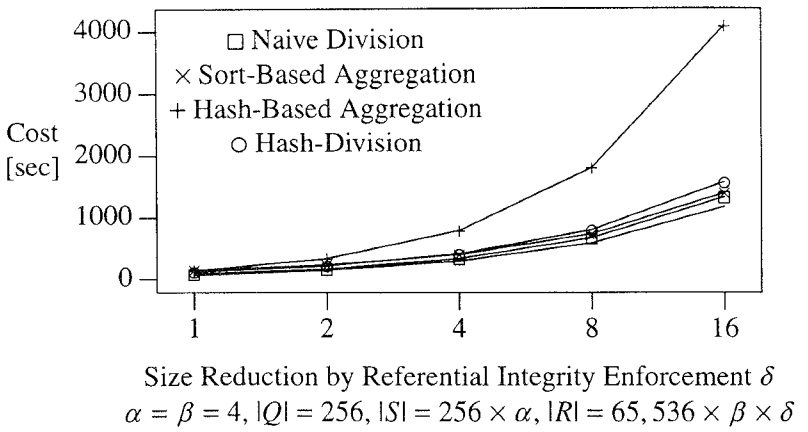
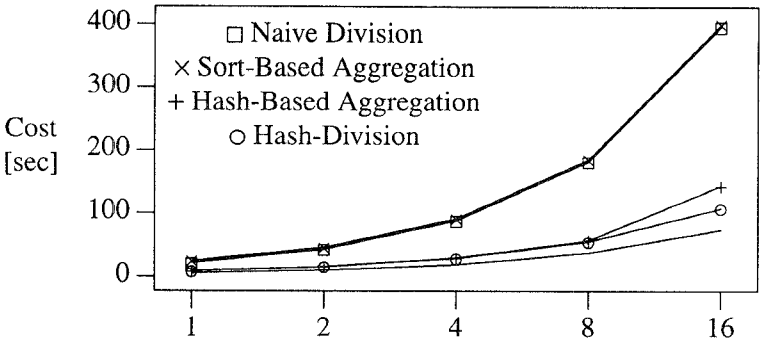


Fig. 10 Division with duplicate removal and referential integrity enforcement; no initial sorts

The other three algorithms are fairly close to each other as well as to the base cost, i.e., the cost of reading the inputs and writing the final output. In other words, reading the inputs and writing the output are the dominant costs for these algorithms.

If Figure 10 had finer resolution, the following detailed trends would be apparent. Naive division is consistently about 15% more expensive than the base cost. Hash-division is 18–43% slower than naive division, with improving relative performance for higher reduction factors δ and increasing dividend sizes $|R|$. Sort-based aggregation shows the most interesting behavior, with costs 20–112% higher than the base cost. For small reduction factors δ, the first sort step between merge-join and aggregation weighs in heavily because the size of the dividend has not been reduced by the merge-join. For large δ, i.e., referential integrity enforcement with large size reduction, the second sort operation becomes almost negligible, and sort-based aggregation even outperforms hash-division.

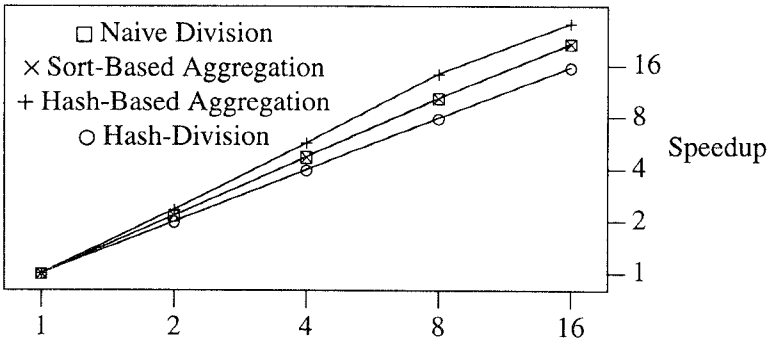
Thus, Figure 10 demonstrates that sorting the large dividend is the dominating cost for sort-based relational division algorithms. If sorting can be avoided by use of clustering B-tree indices, sort-based relational division algorithms become competitive with hash-division. Non-clustering indices would not have the same effect, unless they contain all required attributes and permit index-only retrievals [Graefe 1993]. On the other hand, if the dividend is an intermediate query result of suitable B-tree indices do not exist, hash-division is the algorithm of choice. Seen in a different way, the hash tables of hash-division organize the divisor and dividend tuples into associative structures, just like permanent indices on disk. However, since they are in memory, they are faster than non-clustering indices on disk but not quite as fast as clustering indices, which can deliver the input data immediately in the proper organization (for naive division).



Size Reduction by Referential Integrity Enforcement δ

$P = 16, \alpha = \beta = 4, |Q| = 256, |S| = 256 \times \alpha, |R| = 65,536 \times \beta \times \delta$

(a)



Degree of Parallelism P

$\alpha = \beta = \delta = 4, |Q| = 256, |S| = 256 \times \alpha, |R| = 65,536 \times \beta \times \delta$

(b)

Fig. 11. (a) Parallel division on distributed memory. (b) Speed-up for parallel division algorithms.

7.2 Parallel Algorithms

If division is executed on a parallel machine, inputs and intermediate data must be partitioned across the processors, which add a new cost to all cost formulas above. Figure 11a shows the cost of the algorithms with duplicate removal and referential integrity enforcement for a distributed-memory database machine with $P = 16$ processors and no broadcast assistance ($B = P$). Each processor is assumed to be equipped with CPU, disk, and memory like the single processor in the previous subsection. The indicated relation sizes pertain to the entire system, i.e., each processor stores and processes $1/P$ of these sizes. The base cost of Figure 11a does not include partitioning costs.

The curves for the sort-based algorithms, i.e., naive division and sort-based aggregation and semi-join, are similar to the corresponding curves in Figure 9d. The same is true for hash-division. The curve for hash-based aggregation is different from that in Figure 9d because the parallel machine has a larger total memory. Therefore, hash-based duplicate removal is possible without overflow even for the larger dividend relation, saving significantly on I/O costs. In the cases where in-memory hash-based aggregation is feasible, i.e., $\delta \leq 2$, it outperforms hash-division by a slight margin because counting is faster than manipulating and scanning bit maps.

The similarity of Figure 9d and Figure 11a indicates that the algorithms appear to permit linear speedup. Figure 11b illustrates the speedup behavior of the four parallel division algorithms with 1 to 16 processors in a distributed-memory architecture. While hash-division comes very close to linear speedup, the other algorithms exhibit super-linear speedup. The reason is the same as discussed for hash-division in Figure 11a: If only CPU and disk bandwidth were added, the speedup should be at most linear. However, memory is also added as the system grows, permitting not only a higher total bandwidth but also more effective use of it, because the fraction of data that fits in memory in hybrid hashing as well as the fan-in for sorting and partitioning increases. However, as demonstrated in Figure 11a, hash-division is still the most efficient division algorithm. Unfortunately, super-linear speed-up could not be achieved for any of the algorithms in our real experiments, as we will see in the next section.

8. EXPERIMENTAL PERFORMANCE COMPARISON

To validate the analytical comparisons, we implemented the four division algorithms in the Volcano query processing system. Volcano provides an extensible set of algorithms and mechanisms for efficient query processing [Graefe 1994]. All operators are designed, implemented, debugged, and tuned on sequential systems but can be executed on parallel systems by combining them with a special "parallelism" operator, called the *exchange* operator in Volcano [Graefe and Davison 1993]. Volcano has been used for a number of algorithm studies, such as Graefe [1989], Graefe and Thakkar [1992], and Graefe et al. [1994].

In Volcano, all relational algebra operators are implemented as iterators, i.e., by means of *open*, *next*, and *close* procedures [Graefe 1994]. For example, *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. If the entire input fits into the sort buffer, it is kept there until demanded by the *next* function. Final house-keeping, e.g., deallocating a merge heap, is done by the *close* function. Operators implemented in this model are the standard in commercial relational systems [Graefe 1993].

A tree-structured query evaluation plan is used to execute queries by demand-driven dataflow passing record identifiers and 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 it first consumes the entire sorted divisor relation, building a linked list of divisor tuples fixed in the buffer pool. It then consumes the sorted dividend relation, advancing in the linked list of divisor tuples as matching dividend tuples are produced by the dividend input, producing a quotient tuple each time the end of the divisor list is reached.

Sort-based aggregation and duplicate removal are implemented within Volcano's sort operator. It performs aggregation and duplicate removal as early as possible, that is, no intermediate run contains duplicate sort keys. A 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. For all algorithms involving sort, the run-length was 4,099 records and the fan-in was 20.

The hash-based algorithms use bucket chaining for conflict resolution in hash tables. The hash algorithms use Volcano'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. For all hash-based algorithms, a hash table size (number of buckets) of 4,099 was chosen, providing parity with the sort-based algorithms.

Record sizes of the test data were 32 bytes for divisor and dividend and 8 bytes for the quotient. Dividend and divisor attributes were pseudo-randomly generated and uniformly distributed integers such that all attributes were unique and no data skew was present.

8.1 Sequential Algorithms

The experiments were run on a Sun SparcStation running SunOS with two CDC Wren VI disk drives. One disk was used for normal UNIX file systems for system files, source code, executables, etc., while the other was accessed directly by Volcano as a raw device. Of the system's 24 MB of main memory, 4 MB was allocated for use by Volcano's buffer manager, in order to guarantee that the experiments would not be affected by virtual-memory thrashing. The unit of I/O, between buffer and disk, was chosen to be 4 KB, which was also the page-size.

Simplest Case

Figure 12a-b shows the measured performance of the four algorithms for quotient cardinalities $|Q|$ of 16 and 1,024. Each figure illustrates division performance for a fixed number of quotient records. The divisor cardinality $|S|$ varies from 16 to 4,096 or from 16 to 1,024. In all cases, the dividend is the Cartesian product of the divisor and the quotient, i.e., $R = Q \times S$. This

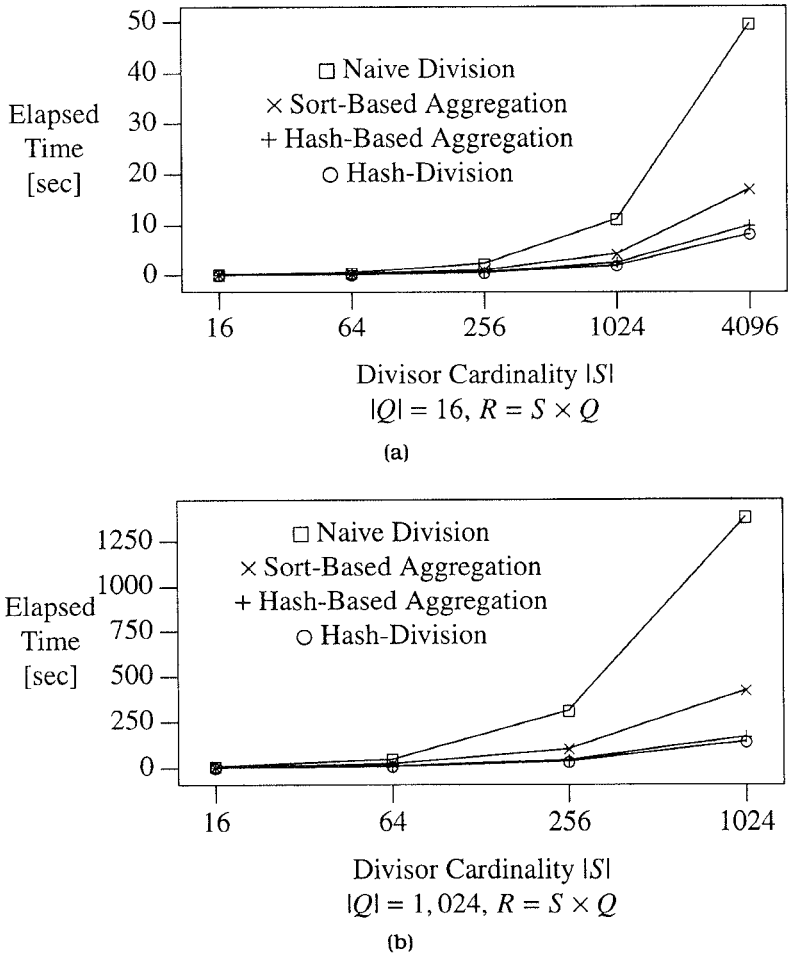


Fig. 12. (a) Measured sequential performance, very small quotient. (b) Large quotient.

implies that neither duplicate removal nor referential integrity enforcement are required, and the simplest form of each algorithm can be applied.

These experimental results (referential integrity holds and no duplicates) substantiate the observations made in the analytical comparisons. In particular, the ranking of the algorithms here is the same one as in the analytical comparison. The sort-based algorithms perform significantly less well than the hash-based algorithms, and the direct implementation of division slightly outperforms the general-purpose hash-based aggregation algorithm. The one "surprise" is that sort-based aggregation appeared equal to naive division in the analysis for this case ($R = Q \times S$) but was significantly faster than naive division in the experiments illustrated in Figure 12 (although not as fast as the hash-based algorithms). This can be attributed to the fact that our analytical model did not reflect early aggregation of the individual sort runs,

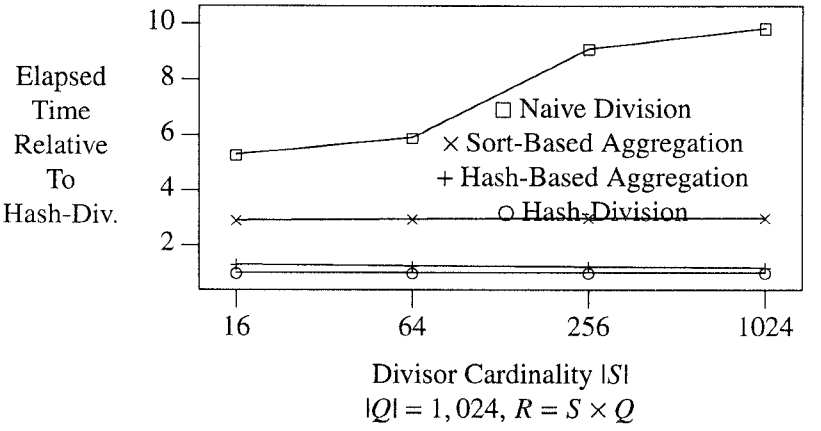


Fig. 13. Performance relative to hash division, large quotient.

and therefore the cost of sort-based aggregation is overestimated in the cost formulas.

The difference in performance for small relation sizes are small, but the difference grows fast as the relations grow. Figure 13 shows the same data as Figure 12b with all elapsed times divided by the elapsed time of hash-division for the same input sizes. The difference between the slowest and the fastest algorithms shown in Figure 12b and Figure 13 ranges from a factor of 5 to a factor of 10. Differences of this magnitude occur for all input sizes in Figure 12a-b. For very small relation sizes, e.g., $|R| = 1{,}024$, $|S| = 64$, and $|Q| = 16$ in Figure 12a, the performance differed by a factor of 3 between the slowest and fastest division algorithms; 0.53 versus 0.17 seconds. For larger relations, e.g., $|R| = 4{,}194{,}304$, $|S| = 4{,}096$, and $|Q| = 1{,}024$ in Figure 12b, the performance differed by a factor of almost 10: 1,380 versus 140.2 seconds. Naive division became relatively worse because it requires that the larger input relation, the dividend, be sorted; however, the sort cannot take advantage of early aggregation like sort-based aggregation [Bitton and DeWitt 1983]. The implementation of division may not be crucially important for small dividend and divisor relations, but if the relations are large, it is imperative to choose the division algorithm carefully. The same is true if the dividend or the divisor are results of other database operations such as selections and joins and the possible error in the selective estimate makes it impossible to predict divisor and dividend sizes accurately.

Referential Integrity Enforcement

Figure 14, shows the measured performance of the four algorithms when the referential integrity constraint between dividend and divisor must be explicitly enforced during query execution. In Figure 14, the dividend cardinality $|R| = 65{,}536$ and the quotient cardinality $|Q| = 256$ are fixed for all data points while the divisor cardinality $|S| = 256/\delta$ varies. δ is the factor by which the dividend size decreases in the referential integrity enforcement

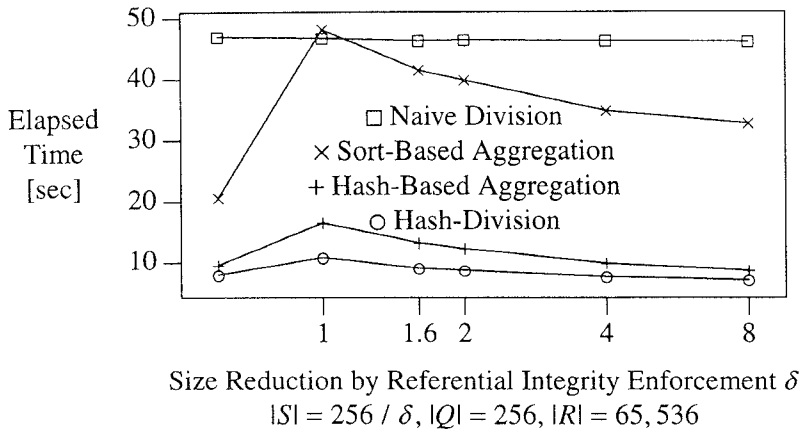


Fig. 14. Measured sequential performance with referential integrity enforcement.

step and varies from 1 to 8. The elapsed times indicated to the left of $\delta = 1$ are the performance of the division algorithms without explicit referential integrity enforcement. In other words, the rise in elapsed times from the leftmost measurements to $\delta = 1$ represents the cost of a redundant referential integrity enforcement step in cases where referential integrity actually already held for the inputs.

All four algorithms except naive division show significant costs for referential integrity enforcement. For large values of δ, the performance improves, because fewer dividend tuples participate in the actual division. The sort-based aggregation algorithm is most strongly affected, from 20.66 (no referential integrity enforcement) to 48.18 ($\delta = 1$) to 32.77 ($\delta = 8$) seconds, due to the additional semi-join and its sort required to remove the non-matching dividend records. The hash-based aggregation algorithm also requires this additional semi-join, but the impact on its performance is less because it can proceed without hash table overflow and without I/O. Its elapsed times range from 9.56 to 16.59 to 8.67 seconds. The hash-division algorithm must use the divisor table, but incurs relatively little performance degradation, from 7.95 to 10.83 to 7.09 seconds.

Note that the experimental results parallel the analytical results, except for sort-based aggregation, which fared worse in the analytical comparison than seems justified from the experimental results. Again, this is because we did not include sort run aggregation into the analytical model, which significantly speeds up sort-based aggregation. This also explains why the elapsed times of sort-based aggregation more than double from the leftmost measurements to $\delta = 1$ when a second dividend sort without early aggregation is added for referential integrity enforcement using a semi-join version of merge-join.

Duplicate Removal

Figure 15 shows the run-time performance of the algorithms when duplicate records are present in the dividend and divisor. In Figure 15, the quotient

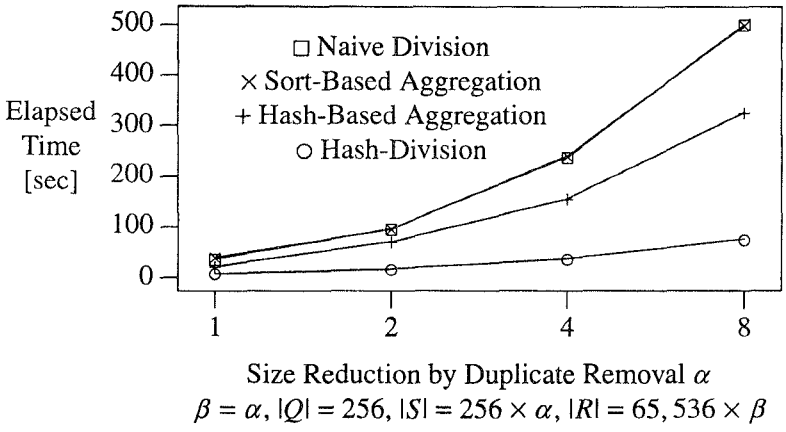


Fig. 15. Sequential performance comparison with duplicates.

cardinality $|Q| = 256$ is constant while the divisor cardinality $|S| = 256 \times \alpha$ and dividend cardinality $|R| = 65{,}536 \times \beta$ vary with replication factors α and β ranging from 1 to 8. In all experiments, $\alpha = \beta$. For example, if $\alpha = 1$, $|R| = 65{,}536$ and $|S| = 256$, and if $\alpha = 8$, $|R| = 524{,}288$ and $|S| = 2.048$.

The performance difference between the worst and best algorithms is again dramatic. For $\delta = 8$, the difference is a factor 6,500.98 seconds versus 76.20 seconds. Here the most important result is the difference between hash-based aggregation and hash-division. In the measurements for division without duplicate removal, the difference between these two algorithms was small, whereas for $\delta = 8$, it is more than a factor of 4,326.63 seconds versus 76.20 seconds.

Differences in the relative performance of the algorithms between the analytical and the experimental results are due to the omission of early aggregation in the analysis and 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 a small percentage of the entire buffer is used as sort space, 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 semi-join, is this effect not observed, and the preceding semi-join and additional sort almost double the cost of sort-based aggregation, e.g., for $|R| = 65{,}536$, $|S| = 256$, and $|Q| = 256$, the run-time is 20.48 versus 39.63 seconds.

It is important to realize that if a universal quantification is expressed in terms of an aggregate function with preceding semi-join and the query optimizer does not rewrite the query to use a direct division algorithm, the query may be evaluated using an inferior strategy. This is true for both sort-based query evaluation systems such as System R, Ingres, and most commercial database systems (sort-based aggregation versus naive division)

and for hash-based systems such as Gamma (hash-based aggregation versus hash-division). Since it is much easier to implement a query optimizer that recognizes a division operation and rewrites it into an aggregation expression than vice versa, universal quantification should be included as a language construct in query languages, for example, as the "contains" clause between table expressions originally included in SQL [Astrahan et al. 1976].

In summary, hash-division is always faster than any of the other algorithms presented here. Even if we compare the most complex version of hash-division, i.e., the version with automatic duplicate removal and referential integrity enforcement, against hash-based aggregation without duplicate removal or referential integrity enforcement, hash-division is only slightly slower than hash-based aggregation. However, when non-matching or duplicate records must be removed from the inputs, hash-division is significantly faster than the next best algorithm, hash-based aggregation.

8.2 Parallel Algorithms

Considering that parallel processing has been demonstrated to permit order-of-magnitude speed-ups in database query processing, any new algorithm or optimization technique should also be evaluated in parallel settings. In addition to the comparisons of our four algorithms on a sequential machine, we also measured speed-up and scale-up on a shared-memory parallel machine. Speed-up compares elapsed times for a constant problem size, whereas scale-up compares elapsed times for a constant ratio of data volume to processing resources. Linear speed-up is achieved if N times more processing resources reduce the elapsed time to $1/N^{th}$ of the time. Linear scale-up is achieved if the elapsed time is constant and independent of the input size, provided the ratio of data volume to processing power is constant. In the following diagrams, both linear speed-up and linear scale-up are indicated as 100% parallel efficiency.

The experimental environment was a Sequent Symmetry with twenty 80386 processors running at 16 MHz (about 3 MIPS each) and twenty disk drives connected to four dual-channel controllers. Of these, we used up to eight processors and eight disks, because we could not obtain more disks for our experiments. In each experiment, we use the same number of disks as processors. The data are round-robin partitioned at the start of each experiment; thus, operations such as aggregation, join, and division require that the first query execution step be repartitioning.

Figures 16a-b show run-times and speed-ups of the four algorithms using two different partitioning schemes. (The mapping of tags on the curves to algorithms is the same as in the previous figures; we omitted it here to keep the figure readable.) While none of the experiments show linear speed-up, all algorithms become faster through parallelism. In other words, the basis for intra-operator parallelism, namely the division of sets (relations) into disjoint subsets, applies not only to the known algorithms for sorting, merge-join, hash-aggregation, and hash join but also to the new universal quantification algorithm, hash-division. This observation holds not only in theory, as reflected in our analytical cost comparisons, but also in practice.

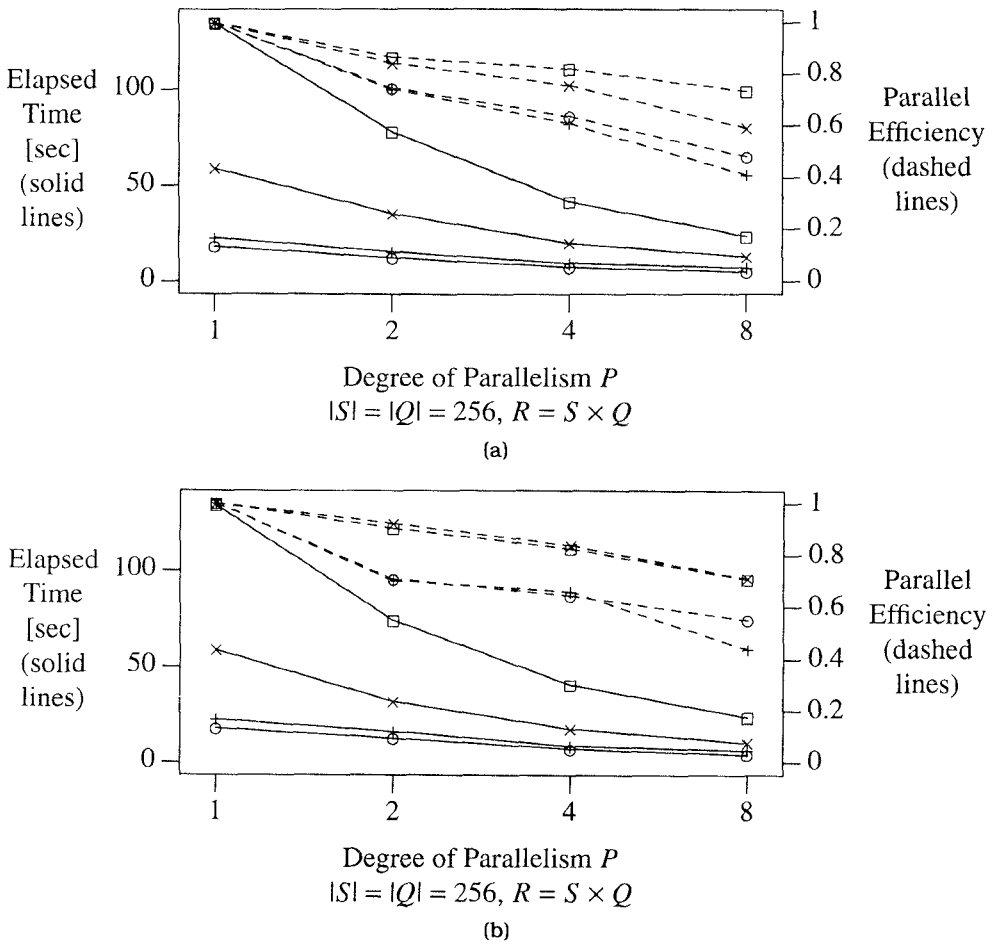


Fig. 16. (a) Speed-up with divisor partitioning. (b) Speed-up with quotient partitioning.

In fact, it would not have been realistic to expect linear speed-up with these relatively small data volumes. First, in this experiment using a shared-memory machine, only processors and disks are added while memory remains constant. Thus, the effect that permitted super-linear speed-up in the analytical comparison is missing. Second, since the Volcano implementation of operators and buffers has been tuned very carefully and is quite efficient [Graefe and Thakkar 1992], the CPU effort for manipulating records is very low and a large fraction of all CPU effort is due to pre-process overhead and protocols for data exchange. For example, for $P = 8$, a scan process for the divisor scans as few as 32 tuples. Using a pool of "primed" Volcano processes helps, but process allocation is still not free. In addition, each process opens and closes its own partition file. With respect to data exchange note that, in Volcano's shared-memory implementation, each data packet going from one process to another requires a synchronization, i.e., a semaphore system call,

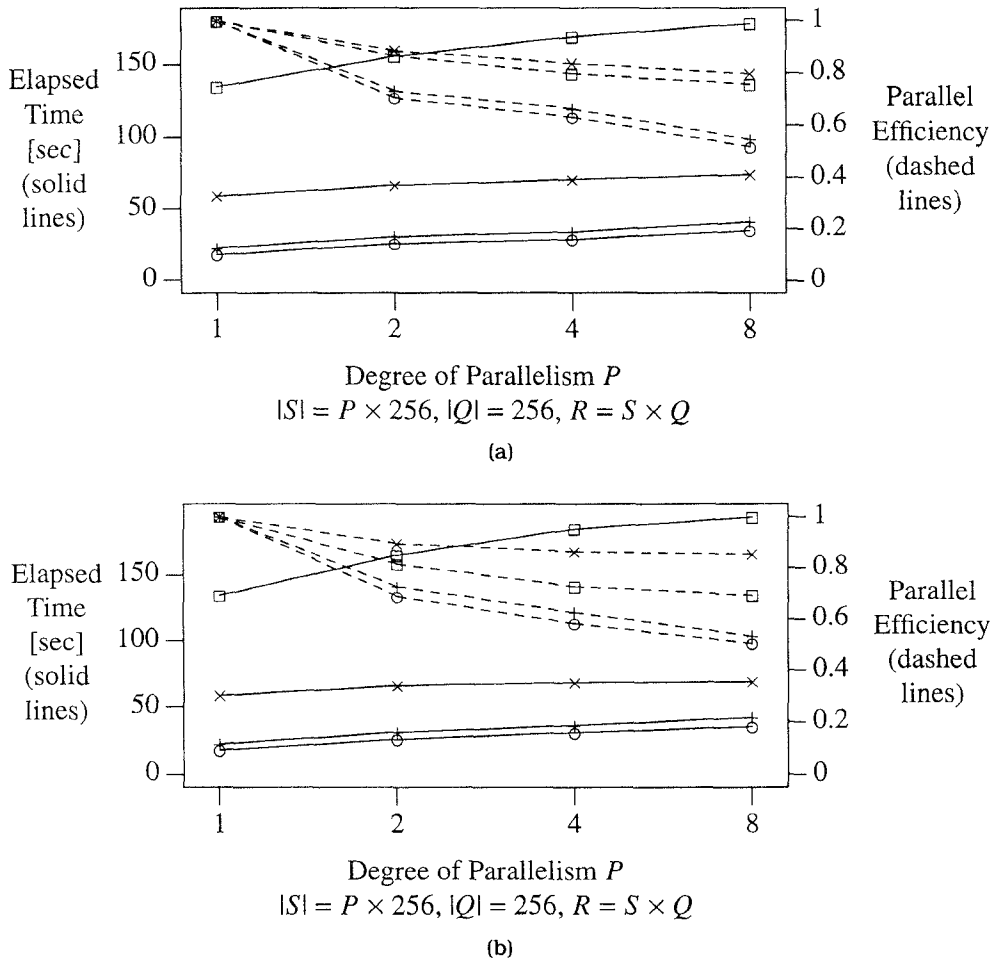


Fig. 17. (a) Scale-up with divisor partitioning. (b) Scale-up with quotient partitioning.

in each process. Since only $1/P$ of all data stay on the same processor in any data exchange step, the fraction of data that must go through the relatively slow interprocess communication system calls increases with increasing degree of parallelism P. These observations are supported by the fact that the faster an algorithm, the poorer its parallel efficiency.

Similar observations apply to the scale-up experiments shown in Figures 17a-b. Figures 17a-b show run-times and scale-ups of the four algorithms with constant ratios of input size to processing power for two partitioning strategies. All algorithms suffer slight increases in processing time for larger data volumes and degrees of parallelism. None of them exhibits truly linear scale-up in our experiments. Nonetheless, the speed-ups and scale-ups obtained are sufficient to support parallel processing for large universal quantification and relational division problems.

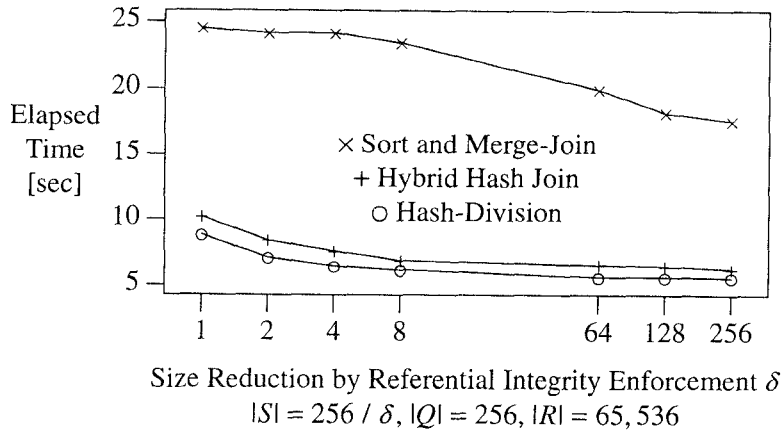


Fig. 18. Existential versus universal quantification.

8.3 Comparison of Existential and Universal Quantification

Existential quantification is well-supported in most query processing systems by semi-join algorithms such as merge-join and hybrid hash join. Figure 18 shows the measured performance of existential and universal quantification, i.e., hash-based semi-join and hash-division, over the same inputs as those in Figure 14, i.e., $|R| = 65{,}536$ and $|R| = 256$ are fixed while $|S| = 256/\delta$ varies. δ represents the dividend's "reduction factor" due to referential integrity enforcement, that is, there are increasing numbers of non-matching records in the dividend. When $\delta = 1$, the semi-join outputs all 65,536 records, whereas the size of the quotient produced by hash-division is only $|Q| = 256$. However, when $\delta = 256$, the output of the semi-join and $|Q|$ are both equal to 256 records, and the algorithms for existential and universal quantification have the same costs for output record processing. We can see that the run-times for hash-based semi-join and hash-division are nearly the same regardless of the number of output records. In fact, the difference between hash-division and hash-join is much smaller than the difference between merge-join and hash-join. This is another strong argument for directly supporting universal quantification in today's database query languages—performance of such operations need no longer be an issue because the performance of universal quantification and relational division can be equivalent to that of existential quantification and relational semi-join.

9. SUMMARY AND CONCLUSIONS

We have described and compared four algorithms for relational division, namely naive division, sort- and hash-based aggregation (counting), and a recently proposed algorithm called hash-division. Analytical and experimental performance comparisons demonstrate the competitiveness of hash-division. Naive division is slow because it requires sorted inputs; it is competitive only if neither input incurs costs for sorting. Division using sort-based

aggregate functions is almost as expensive as naive division; if a semi-join and an additional sort are required to ensure that only proper dividend tuples are counted, sort-based aggregation can be more expensive than naive division. Division using hash-based aggregate functions is exactly as fast as hash-division when neither a preceding semi-join nor a duplicate removal step are required. If either one is required because referential integrity or uniqueness must be enforced, hash-division outperforms all other algorithms because it automatically enforces referential integrity and ignores duplicates. However, as in our second example (students who have taken all database courses), referential integrity is unlikely to hold if the divisor is the result of a prior selection or other restrictive preprocessing step. On the other hand, if referential integrity holds and both dividend and divisor are duplicate-free, the basic hash-division algorithm can be simplified by omitting the divisor table and substituting counters for the bit maps associated with quotient candidates, thus making it similar to hash-based aggregation in both algorithm and performance.

Like the other algorithms consider here, hash-division is easy to parallelize. Its speed-up and scale-up behavior is similar to that of hash-based aggregation and join, while its absolute run-times are consistently lower than those of universal quantification using hash-based aggregation. Moreover, since hash-division does not require referential integrity to hold, it never requires partitioning the dividend input twice, as is necessary in indirect division algorithms based on aggregation, i.e., for the semi-join and the subsequent aggregate function. Therefore, hash-division is particularly valuable in parallel systems, where it outperforms all other algorithms. Our primary conclusion from this study is that universal quantification and relational division can be performed very efficiently in sequential and parallel database systems by using hash-division.

An additional result of our comparisons is that direct algorithms tend to outperform indirect strategies based on aggregate functions, both in sort-based and in hash-based query evaluation systems. In direct algorithms, the larger input, the dividend, has to be partitioned, sorted, or hashed only once, not twice as in indirect strategies based on aggregation. Thus, division problems should be evaluated by direct division algorithms. However, it is much easier for a query optimizer to detect a "for-all" or "contains" expression and to rewrite it into a query evaluation plan using aggregation than it is to determine that a complex aggregation expression or a pair of nested "NOT EXISTS" clauses are actually "work-arounds" for a universal quantification problem. Thus, query languages should include "for-all" quantifiers and "contains" predicts, e.g., as the "contains" clause between table expressions originally included in SQL [Astrahan et al. 1976]. In other words, it was a mistake to remove that clause from the original language design, and it should be restored in future versions of SQL and in other database query languages.

Finally, hash-division is fast not only when compared to other division algorithms but also when compared to existential quantification algorithms, i.e., semi-joins. We have shown experimentally that hash-division is as fast as

hash-based semi-join for the same two inputs, and performs much better than sort-based semi-join. Thus, the recently proposed algorithm, hash-division, permits database implementors and users to consider universal quantification and relational division with a justified expectation of excellent performance.

ACKNOWLEDGMENTS

Dale Favier, David Maier, Barb Peters, and the anonymous reviewers made many excellent suggestions for the presentation of this paper.

REFERENCES

ASTRAHAN, M. M., BLASGEN, M. W., CHAMBERLIN, D. D., ESWARAN, K. P., GRAY, J. N., GRIFFITHS, P. P., KING, W. F., LORIE, R. A., MCJONES, P. R., MEHL, J. W., PUTZOLU, G. R., TRAIGER, I. L., WADE, B. W., AND WATSON, V. 1976. System R: A relational approach to database management. *ACM Trans. Database Syst. 1*, 2 (June), 97. Reprinted in Stonebraker, M., 1988 *Readings in Database Systems*, Morgan Kaufmann, San Mateo, Calif.

BITTON, D., AND DEWITT, D. J. 1983. Duplicate record elimination in large data files. *ACM Trans. Database Syst.* 8, 2 (June), 255.

BRATBERGSENGEN, K. 1984. Hashing methods and relational algebra operations. *Proc. Int'l. Conf. on Very Large Data Bases* (Singapore, August), VLDB Endowment, 323.

CARLIS, J. V. 1986. HAS: A relational algebra operator, or divide is not enough to conquer. *Proc. IEEE Int'l. Conf. on Data Eng.* (Los Angeles, Calif., February), IEEE, New York. 254

CHENG, J., LOOSELY, C., SHIBAMIYA, A., AND WORTHINGTON, P. 1985. IBM database 2 performance: design, implementation, and tuning. *IBM Syst. J. 23*, 2.

CODD, E. F. 1972. *Relational Completeness of Database Sublanguages*. Prentice-Hall, New York.

DATE, C. J., AND DARWEN, H. 1993. *A Guide to The SQL Standard*, 3rd ed., Addison-Wesley, Reading, Mass.

DEWITT, D. J., KATZ, R., OLKEN, F., SHAPIRO, L., STONEBRAKER, M., AND WOOD, D. 1984. Implementation techniques for main memory database systems. *Proc. ACM SIGMOD Conf.* (Boston, Mass., June), 1.

DEWITT, D. J., AND GERBER, R. H. 1985. Multiprocessor hash-based join algorithms. *Proc. Int'l. Conf. on Very Large Data Bases* (Stockholm, Sweden, August) VLDB Endowment, 151.

DEWITT, D. J., GERBER, R. H., GRAEFE, G., HEYTENS, M. L., KUMAR, K. B., AND MURALIKRISHNA, M. 1986. GAMMA—A high performance dataflow database machine. *Proc. Int'l Conf. on Very Large Data Bases* (Kyoto, Japan, August), 228. Reprinted in Stonebraker, M., 1988. *Readings in Database Systems*, Morgan Kaufmann, San Mateo, Calif.

DEWITT, D. J., GHANDEHARIZADEH, S., SCHNEIDER, D., BRICKER, A., HSIAO, H. I., AND RASMUSSEN, R. 1990. The GAMMA database machine project. *IEEE Trans. Knowledge Data Eng. 2*, 1 (March), 44.

EPSTEIN, R. 1979. Techniques for processing of aggregates in relational database systems. Univ. of California at Berkeley, UCB/ERL Memorandum M79/8, (February).

FUSHIMI, S., KITSUREGAWA, M., AND TANAKA, H. 1986. An overview of the system software of a parallel relational database machine GRACE. *Proc. Int'l Conf. on Very Large Data Bases* (Kyoto, Japan, August), VLDB Endowment, 209.

GRAEFE, G. 1989. Relational division: Four algorithms and their performance. *Proc. IEEE Int'l. Conf. on Data Eng.* (Los Angeles, Calif., February), IEEE, New York, 94.

GRAEFE G., AND THAKKAR, S. S. 1992. Tuning a parallel database algorithm on a shared-memory multiprocessor. *Softw.-Pract. Exper. 22*, 7 (July), 495.

GRAEFE, G. 1993. Query evaluation techniques for large databases. *ACM Comput. Surv. 25*, 2 (June), 73–170.

GRAEFE, G., AND DAVISON, D. L. 1993. Encapsulation of parallelism and architecture-independence in extensible database query processing *IEEE Trans. Softw Eng 19*, 8 (August), 749.

GRAEFE, G 1994. Volcano: An extensible and parallel dataflow query processing system. *IEEE Trans. Knowl. Data Eng. 6*, 1 (February), 120.

GRAEFE, G., LINVILLE, A., AND SHAPIRO, L. D. 1994. Sort versus hash revisited. *IEEE Trans Knowl. Data Eng. 6*, 6 (December), 934.

KITSUREGAWA, M., TANAKA, H , AND MOTOOKA, T. 1983. Application of hash to data base machine and its architecture. *New Generation Comput. 1*, 1 63.

KITSUREGAWA, M., NAKAYAMA, M., AND TAKAGI, M. 1989. The effect of bucket size tuning in the dynamic hybrid GRACE hash join method. *Proc. Int'l. Conf. on Very Large Data Bases* (Amsterdam, The Netherlands, August), VLDB Endowment, 257

KNUTH, D. 1973. *The Art of Computer Programming, Vol. III: Sorting and Searching*. Addison-Wesley, Reading, Mass.

MAIER, D 1983. *The Theory of Relational Databases*. Computer Science Press, Rockville, Md.

NAKAYAMA, M., KITSUREGAWA, M., AND TAKAGI, M. 1983. Hash-partitioned join method using dynamic destaging strategy. *Proc. Int'l. Conf. on Very Large Data Bases* (Los Angeles, Calif., August), VLDB Endowment, 468.

O'NEIL, P. E. 1994 *Introduction to Relational Databases*. Morgan Kaufmann, San Mateo, Calif.

SACCO, G. M. 1986. Fragmentation: A technique for efficient query processing. *ACM Trans. Database Syst 11*, 2 (June), 113

SALZBERG, B., TSUKERMAN, A., GRAY, J , STEWART, M., UREN, S., AND VAUGHAN, B. 1990. FastSort: A distributed single-input single-output external sort. *Proc. ACM SIGMOD Conference* (Atlantic City, N.J., May), ACM, New York, 94.

SCHNEIDER D., AND DEWITT, D. 1989. A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment. *Proc. ACM SIGMOD Conference* (Portland Ore., May-June), ACM, New York, 110.

SHAPIRO, L. D. 1986 Join processing in database systems with large main memories. *ACM Trans. Database Syst. 11*, 3 (September), 239.

SMITH, J. M., AND CHANG, P. Y. T. 1975. Optimizing the performance of a relational algebra database interface. *Commun. ACM 18*, 10 (October), 568.

STONEBRAKER, M.. 1986. The case for shared-nothing. *IEEE Data Eng. Bull. 9*, 1 (March).

WHANG, K. Y., MALHOTRA, A., SOCKUT, G., AND BURNS, L. 1990. Supporting universal quantification in a two-dimensional database query language. *Proc. IEEE Int'l. Conf. on Data Eng.* (Los Angeles, Calif., February), IEEE, New York, 68.

ZELLER, H. AND GRAY, J. 1990. An adaptive hash join algorithm for multiuser environments. *Proc. Int'l. Conf. on Very Large Data Bases*. (Brisbane, Australia, August) VLDB Endowment, 186

Received April 1994; revised January 1995; accepted March 1995