DATA BASE MANAGEMENT

ASSESSING PERFORMANCE OF THE RELATIONAL DIVISION OPERATOR

Victor M. Matos and Rebecca Grasser

INTRODUCTION

The relational data model introduced by E. Codd in the early 1970s has become the default for conceptualizing and implementing modern database systems.\textsuperscript{1,2} The relational model deals with information held in simple two-dimensional tables (called relations), consisting of rows of interrelated data. Each row in a relation plays a role similar to a record in a conventional file system and each column is equivalent to a field. To manipulate these tables, Codd proposed a compact symbolic language called relational algebra and its equivalent tuple-calculus version. The basic operators of the relational algebra are the projection, selection, Cartesian product, union, and difference.\textsuperscript{1,2} These operators are the foundation for modern database query languages and have been extensively discussed in the database literature.

For convenience, Codd added other useful operators to his original algebra. The list includes different forms of joins (general, natural, left/right outer), rename, intersection, and division. Other authors have made extensions to Codd's relational model of data. In Klug's work,\textsuperscript{3} the data model has the ability to perform simple arithmetical and statistical computations; while the relational

PAYOFF IDEA

The purpose of the relational division operator is to verify whether or not a set of candidate records (known collectively as the dividend table) is related to each and every one of the tuples in another set (called the divisor table). Division-like queries are important as they frequently occur in practice. Unfortunately, expressing relational algebra division in terms of the commonly used SQL query language is difficult and could result in very time-consuming implementations. This article revises the operational characteristics of a typical implementation and compares it with a much faster version written by the authors. Each SQL implementation is assessed in Microsoft Access97 and Oracle8 database environments.
model of Ozsoyoglu et al.\textsuperscript{4} is extended to accept non-atomic fields and provide support for nested relations. As shown later, there is a significant connection between the division operator and nested relations.

The \textit{division} operator is less common than simple \textit{join-select-project} queries. However, it is naturally applied in many common, everyday queries. For example, \textit{division} could be used in solving the following problems:

- who are the customers who subscribe to \textit{each} of the TV premium channels
- find suppliers who supply \textit{all} the red parts
- find students who have taken \textit{all} the core courses
- find customers who have ordered \textit{all} items from a given line of products

The characteristic pattern in this family of inquires is the attempt to verify whether or not a candidate subject is related to each of the values held in a base set. The base set is called the divisor (or denominator T2[B]), and the table holding the subject's data is called the dividend (or numerator T1[A,B]). Without losing generality, the expression T1[A,B]/T2[B] selects the A values from the dividend table T1[A,B], whose B values are a superset of those B values held in the divisor table T2[B].

In today's database community, the SQL query language\textsuperscript{5} represents the conventional tool used for retrieving and maintaining data in relational databases. Surprisingly, expressing the relational algebra \textit{division} operator in terms of the SQL language is not a simple task. Empirically collected evidence suggests that both advanced programmers as well as novice programmers have a difficult time dealing with the SQL version of the algebraic division.

A large number of highly regarded database books\textsuperscript{6-12} describe the implementation of the \textit{division} operator using the SQL syntax of Q1. However, the authors have found that this solution is not only difficult for the programmers to understand and maintain, but also computationally complex. Its performance was found to reach unmanageable levels of degradation such that in some cases it cannot be considered a viable alternative.

This article evaluates four alternative ways of phrasing relational division using SQL. These queries are called Q0, Q1, Q2, and Q3, respectively. The syntax used in each of the solutions is different and mimics an attempt to phrase the division process in relational algebra and relational calculus. It was observed — with a great deal of surprise — that one of the SQL solutions using a \textit{join-and-count} approach produces a remarkably good performance.

The Microsoft Access and Oracle8 database environments were used for the study. Experiments measured the performance of each of the solutions Q0 through Q3 in the presence of (1) a varying number of
records in the dividend and (2) a changing selectivity factor between the dividend and divisor tables. A significant disproportion is encountered between the slow performance of the SQL code customarily suggested by the current literature and the much faster alternative recommended by the authors. The next sections formally describe the division operator and the experiment, followed by a discussion of the results.

AN EXAMPLE OF THE DIVISION OPERATOR

Consider the tables T1[A,B] and T2[B] depicted in Exhibit 1. T1 represents a list of customers and the options they bought for their new cars. Column A is the customer identification number and Column B represents the option included in the car. For example, customer a1 bought her vehicle with the b1, b2, and b3 options. Table T2[B] represents a particular set of options (that is, b2: leather seats, and b3: winter package). The resulting table T3[A] identifies the customers who acquired at least those items listed in table T2[A,B].

DEFINING THE DIVISION OPERATOR

Algebraic Version

Codd's division operator can be defined in terms of basic algebra constructors. For simplicity, assume that the numerator table T1 always consists of two columns A and B, and the denominator has only one B attribute. Under these conditions, the expression T1[A,B]/T2 is semantically equivalent to
### EXHIBIT 2 — Step-by-Step Evaluation of the Algebraic Division T1[A,B]/T2[B]

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a1</td>
<td>a1 a2 a3 a4</td>
<td>a2 a4</td>
<td>a2 a4</td>
</tr>
<tr>
<td>a2</td>
<td>a1 a2 a3 a4</td>
<td>a3 a4</td>
<td>a3 a4</td>
</tr>
<tr>
<td>a3</td>
<td>a1 a2 a3 a4</td>
<td>a3 a4</td>
<td>a3 a4</td>
</tr>
<tr>
<td>a4</td>
<td>a1 a2 a3 a4</td>
<td>a3 a4</td>
<td>a3 a4</td>
</tr>
<tr>
<td>T2[B]</td>
<td>b2 b3</td>
<td>b2 b3</td>
<td>b2 b3</td>
</tr>
</tbody>
</table>

\[
\]

(1)

A quick review of each of the primitive operators used in definition (1) reveals that the projection T1[A] extracts all of the A-values from table T1 and that duplicate rows are ignored. Exhibit 2(a) shows the four tuples in T1[A].

The Cartesian product (×) makes the cross-product of the two tables. For example T1[A] × T2[B] cross-tabulates \{a1,a2,a3,a4\} with \{b2,b3\} to produce each of the eight tuples of Exhibit 2(c). Notice that the new rows consist of all the combinations of values from T1[A] and T2[B].

The minus operator on T1 − T2 selects all those rows that exclusively appear in T1 and have no matching counterpart in T2. Exhibits 2(d) and 2(e) illustrate the two subtractions performed in the original definition of division. For example, row (a2,b2) of Exhibit 2(d) appears in the table of Exhibit 2(c) but not in the T1[A,B] table listed in Exhibit 1. The final solution is given in Exhibit 2(f).

#### Tuple-Calculus Version

Codd’s relational model described two equally expressive database query languages: (1) the relational algebra and (2) its sibling, the predicate calculus-based query language. Using relational tuple-calculus language, the division operator can be rephrased as follows.

\[
T1[A,B]/T2[B] = \{t1[A]/t1 \in T1 \text{ and For-All } t2 (t2 \in T2 \rightarrow \text{Exists } t3 (t3 \in T1\text{ and} \\
(t1[A] = t3[A]) \text{ and } \\
t2[B] = t3[B]))\}
\]

(2)
This expression is rather complex and needs to be carefully evaluated. As before, assume that T1 represents a list of customers and the options they included in their new vehicles. T2 is a set of specific options. Now analyze each part of definition (2).

A candidate tuple t1[A] holding the customer's ID is placed in the output set if it satisfies the following conditions:
- t1 is a current tuple in the T1 table,
- for-all tuples t2 instantiated from table T2, there must exist a tuple named t3 in T1, such that
  - the ID numbers of t1 and t3 are the same (t1[A] = t3[A]),
  - the option numbers of t2 and t3 have a match (t2[B] = t3[B])

Because this test is done for each t2 held in T2, the candidate t1 is selected only if the complete collection of T2 records has a successful match with tuples in T1. Failure to do so would immediately disqualify the candidate output t1[A].

THE DIVISION OPERATOR AND NON-FIRST-NORMAL FORM DATABASES

Several authors have extended the basic relational database model to accept tables that are not in 1NF format. It turns out that by relaxing the data model, the definition of relational division becomes trivial. On these extended relational data models, also known as nested relations or NFNF (Non-First-Normal Form) databases, fields could be either atomic values or sets of simple atomic values.

Example

Exhibit 3 shows an NFNF representation of the database introduced in Exhibit 1. Attribute *B in T1 and T2 is defined as a set of atomic values. For example, customer a1 has acquired a vehicle with options b1, b2, and b3, while customer a4 has requested only the option b1.

<table>
<thead>
<tr>
<th>T1</th>
<th>*B</th>
<th>T2</th>
<th>*B</th>
<th>TA</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>a1</td>
<td>{b1, b2, b3}</td>
<td>{b2, b3}</td>
<td></td>
<td></td>
<td>a1</td>
</tr>
<tr>
<td>a2</td>
<td>{b1, b3}</td>
<td></td>
<td></td>
<td></td>
<td>a3</td>
</tr>
<tr>
<td>a3</td>
<td>{b2, b3, b4}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a4</td>
<td>{b1}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note: Fields are allowed to include atomic values, as well as (one level) set of atomic values.
For simplicity, one can concentrate on the model suggested by Ozsrayoglu et al., which tolerates the normal atomic fields and sets of simple atomic fields. In this model, the division can be defined as:

\[
T1/T2 = \{ t1[A]/t1 \in T1(t1) \text{ and } t2 \in T2 \text{ and } ( t2[*B] \subseteq t1[*B] ) \}
\] (3)

That is, a customer ID represented by \( t1[A] \) is placed in the resulting table if the set of option values collected in \( t2[*B] \) is a sub-set of its \( t1[*B] \) field. For example, customer \( a1 \) is selected because her \( t1[*B] \) collection \{b1, b2, b3\} is a superset of the divisor set \{b2, b3\}.

This definition is much simpler than its equivalent 1NF version. However, one still needs operators to nest and unnest the fields of a relation to create a set, or dissolve a set of values into independent tuples. The notation \( t1[*B] \) identifies a nested field holding a set of atomic values, while the notation \( t1[B] \) represents the conventional atomic value. These operators are extensively discussed by Ozsrayoglu et al. and further discussion is beyond the scope of this article.

FOUR SQL VERSIONS OF THE DIVISION OPERATOR

This section presents four SQL solutions inspired by the definitions identified as (1), (2), and (3). An additional version constructed on the basis of grouping and counting is also included. The four solutions are Q0, Q1, Q2, and Q3, respectively. This section investigates the performance of the solutions on different samples of the database whose skeleton is suggested in Exhibit 1, and then explores its behavior in the presence of an empty divisor table.

Q0

Computing relational division using membership test, Group-by, counting, and having SQL constructors

\[
\text{Q0: SELECT A} \\
\text{FROM T1} \\
\text{WHERE B IN ( SELECT B FROM T2 )} \\
\text{GROUP BY A} \\
\text{HAVING COUNT(*) = ( SELECT COUNT(*) FROM T2 );}
\]

Version Q0 is our interpretation of the division operator, and from now on this code is referred as the version suggested by the authors (clearly, the authors do not claim to be the first using such a formula). In a way, it resembles the interpretation of division using NFNF nested relations.

This SQL implementation of the division operator splits the T1 tuples in groups identified by their A values. The GROUP BY A clause is responsible for creating the non-overlapping A-partitions. Tuples in each A-group
have already been restricted by the \texttt{WHERE}... predicate to those whose B-value matching any entry in T2. The count of tuples in each partition is compared with the size of table T2. Only those A-groups \texttt{HAVING}... the same count are selected, and their A-value is finally selected.

\textbf{Example}

When Q0 is applied to table T1, the following three groups of \texttt{<A,B>} tuples are generated: \texttt{<a1, b2, b3>}, \texttt{<a2, b3>}, \texttt{<a3, b2, b3>}. Observe that B-values are restricted only to those B-values currently held in T2. Customer a4 is not considered because his only option, b1, is not one of the rows in T2. Only the first and last groups, identified by a1 and a3, respectively, have the same number of tuples as T2. Finally, the A-identifiers a1 and a3 are retrieved.

\textbf{Q1: Byzantine Method}

Solution made using \texttt{NOT EXISTS} operator to simulate the predicate calculus definition. The \texttt{for-all} universal quantifier is not implemented in SQL. However, it could be indirectly implemented using the identity

\[ \text{For-all } x \ (f(x)) = \text{not Exists } x \ (\text{not } f(x)) \]

Using the above equality, one could manipulate the tuple-calculus definition (2) to obtain a description agreeable to SQL syntax. One can re-write definition (2) as follows

\[ T1[A,B]/T2[B] = \{t1 \ [A]/t1 \in T1 \text{ and} \]
\[ \text{Not Exists } t2 \ (\text{not } (t2 \in T2 \rightarrow \text{Exists } t3 \ (t3 \in T1 \text{ and} (t1[A] = t3[A]) \text{ and} (t2[B] = t3[B])))) \} \]

The logical implication \(X \rightarrow Y\) can be replaced by its equivalent formulation (\texttt{not}(X) \texttt{or} Y), giving the following result:

\[ T1[A,B]/T2[B] = \{t1 \ [A]/t1 \in T1 \text{ and} \]
\[ \text{Not Exists } t2 \ (\text{not } (\text{not}(t2 \in T2) \text{ or (Exists } t3 \ (t3 \in T1 \text{ and} (t1[A] = t3[A]) \text{ and} (t2[B] = t3[B])))))) \} \]

which is equivalent to:

\[ T1[A,B]/T2[B] = \{t1 \ [A]/t1 \in T1 \text{ and} \]
\[ \text{Not Exists } t2 \ ((t2 \in T2) \text{ and (not Exists } t3 \ (t3 \in T1 \text{ and} (t1[A] = t3[A]) \text{ and} (t2[B] = t3[B]))))) \} \]
This expression can finally be converted to SQL using a double *not exists* clause:

```
Q1: SELECT DISTINCT x.A
    FROM T1 AS x
    WHERE NOT EXISTS ( SELECT *
        FROM T2 y
        WHERE NOT EXISTS ( SELECT *
            FROM T1 AS z
            WHERE (z.A=x.A) AND (z.B=y.B)) );
```

x, y, and z are aliases of the tables t1, t2, and t3, respectively. Here the outer **SELECT** statement picks a candidate x.A as a potential solution. For this candidate to become part of the final solution, it should not exist a tuple y in T2 for whom it does not exist a tuple z in T1 which matches simultaneously the candidate's ID (x.A=z.A) and each y value (y.B=z.B). If such y tuple exists, it would create a contradiction, because there is data in T2 to which the candidate is not related, and therefore the candidate must be rejected.

It is important to mention that the above SQL version of the division operator is typically suggested by the computer science literature for implementing the division operator. More on this topic later.

**Q2**

Uses EXISTS clause to simulate the algebraic definition of Codd's *divide* operator.

```
Q2A: SELECT DISTINCT y.A, z.B INTO T3
    FROM T1 AS y, T2 AS z
    WHERE NOT EXISTS ( SELECT *
        FROM T1
        WHERE (T1.A=y.A) AND (T1.B=z.B));
Q2B: SELECT DISTINCT A
    FROM T1
    WHERE NOT EXISTS ( SELECT *
        FROM T3
        WHERE (T3.A=T1.A) );
```

The algebraic formula (2) is broken in two steps. The first step computes the list of A-values from T1 that fail to join at least one of the B-values in T2. The z.B field indicates the missing B for a given A with respect to T2. The partial results are stored in a temporary table T3, which in the final step is subtracted from the list of A-values in T1. Those As that appear exclusively in T1 will be selected, and those that appear in T3 will be discarded.
Q3
Simulating the NNF algebra of Ozsoyoglu et al.4

Q3: SELECT DISTINCT x.A  
    FROM T1 AS x  
    WHERE (SELECT Count(*) FROM T2 )  
        = 
        (SELECT Count(*) FROM T1, T2 WHERE (T1.A = x.A) AND 
         (T1.B = T2.B ) );

Division is performed using the COUNT clause but with a varying number of T1 and T2 tuples, as well as a varying degree of selectivity for does not use HAVING/GROUP-BY. This version simulates the set containment operation of expression (3). In a way, it approximates the divide operator of the nested relational algebra of Ozsoyoglu et al.4 In the innermost SELECT statement, the tables T1, and T2 are joined using their common attribute B. The number of tuples having the same A-value of the external candidate x.A is computed. If this number is exactly the same as the size of T2, the current x.A value is selected.

DESCRIPTION OF THE EXPERIMENT
The four SQL versions of the relational division operator were tested to profile their run-time behavior. A database similar to that of Exhibit 1 was used. The performance of each query was evaluated under the effect of two control variables: (1) table size and (2) selectivity factor. Table size refers to the actual numbers of records stored in each table. Selectivity factor is a ratio expressing the coherence between the two tables. For example, assume the T1(A,B) tuples holding a1 under the A-column are [(a1,b1), (a1,b2), (a1,b3)]. If the divisor T2(B) consists of the records [(b1), (b2), (b3), (b4), (b5)], the selectivity factor is 3/5 or 60 percent.

The tests were executed on an IBM-PC machine under Windows 98. The processor was a Pentium-II 266, 64 MB of RAM, and 6 GB of disk space. The software environment was provided by Microsoft Access97 and Personal Oracle8 version 8.04. The rationale for this selection is twofold. First, the results produced by a consumer-grade system (Access7) versus an industrial-grade product (Oracle8) could be contrasted. Second, the query optimizer of each system provided a plan for the efficient implementation of each query. In this analysis, there was concern for the relative performance of one query with respect to the others, rather than the absolute execution times. There was no attempt to compare one DBMS to the other. Times are reported to include the execution of each query and the transferring of the results to a permanent table.

All evaluations were performed under the same conditions. Two tables, T1(A,B) and T2(B), were artificially produced from a common set
of values. The pattern used to make T1(A,B) consisted of adding exactly five B-values for each A-value in the table. The B-values in the T2(B) table varied from zero to five entries.

For example, the largest instance of the table T2(B) consisted of [b1, b2, b3, b4, b5]. In successive evaluations, that size was reduced to the four records [b1, ..., b4], then three, etc. The final configuration of T2(B) contained no data.

Meanwhile, the records in T1(A,B) were made with 100, 200, 500, and 1000 different A-values such as [a1, a2, ..., a1000]. The first T1(A,B) sample consisted of 500 records. A total of 100 A-values were paired with five B-values to produce the collection ([a1,b1], ..., (a1,b5), ..., (a100,b1), ..., (a100,b5)). The next sample of 1000 records had 200 A-values, each coupled with five B-values. In the last sample, a total of 1000 A-values were connected with B-values to generate a total of 5000 records.

Exhibit 1 summarizes the results of the experiment. Observe that query Q1 (under Access) has several cells annotated "na." (not available). The reason for this is that a symbolic cutoff limit of 36 hours was imposed to prevent the tests from running indefinitely. The authors intentionally decided against manipulating the queries in order to obtain better performance. On the contrary, the initial strategy was to allow each DBMS optimizer to prepare its own evaluation plan. However, the inept results obtained from Access when working on raw tables forced the authors to add indices on the involved attributes to carry out the computation of the samples. Only the results for Q1 using Microsoft Access were affected by the presence of additional indices. For example, the cell for Q1 at 20 percent selectivity using the sample with 500 records is recorded as "179/7."

These numbers show that 197 seconds of computing time are required for the evaluation of Q1 without indices; at the same time, only 7 seconds are needed by Q1 to operate on the indexed sample. The next section discusses the results.

**Discussion**

Observation of Exhibit 4 clearly suggests that query Q0 outperforms all of the other SQL implementations of the division operator. The most striking characteristic of Q0 is the relative quickness of execution, and independence from the two control variables. Both DBMS systems (Oracle and Access) produced virtually the same results for Q0. However, Oracle was generally much faster than MS-Access in each of the other evaluations.

The Byzantine version of Q1 is notoriously slow and sensitive to both changes in the size of the numerator table as well as the selectivity factor. The performance of Q1 in Access is essentially unacceptable. The sample T1/T2 holding 2500 and 5 records, respectively, took about a day and a half to complete. In an operational environment, those sizes would be
### EXHIBIT 4 — Summary of Access97 and Oracle8 Benchmarks for the SQL Division Operator

<table>
<thead>
<tr>
<th>Query</th>
<th>Denominator Selectivity (%)</th>
<th>Access Numerator (records)</th>
<th>Oracle 8 Numerator (records)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>500</td>
<td>1000</td>
<td>2500</td>
</tr>
<tr>
<td>Q0</td>
<td>0</td>
<td>Empty Div</td>
<td>1</td>
</tr>
<tr>
<td>Q0</td>
<td>0</td>
<td>Non Empty</td>
<td>1</td>
</tr>
<tr>
<td>Q0</td>
<td>20</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Q0</td>
<td>40</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Q0</td>
<td>60</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Q0</td>
<td>80</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Q0</td>
<td>100</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Q1*</td>
<td>0</td>
<td>Empty Div</td>
<td>1</td>
</tr>
<tr>
<td>Q1*</td>
<td>0</td>
<td>Non Empty</td>
<td>2</td>
</tr>
<tr>
<td>Q1*</td>
<td>20</td>
<td></td>
<td>7</td>
</tr>
<tr>
<td>Q1*</td>
<td>40</td>
<td></td>
<td>12</td>
</tr>
<tr>
<td>Q1*</td>
<td>60</td>
<td></td>
<td>552</td>
</tr>
<tr>
<td>Q1*</td>
<td>80</td>
<td></td>
<td>729</td>
</tr>
<tr>
<td>Q1*</td>
<td>100</td>
<td></td>
<td>942</td>
</tr>
</tbody>
</table>

* Cells with two values represent elapsed times for computation of Q1 without and with indices.

na. indicates evaluations requiring a time in excess of 36-hours; therefore, they are considered “not available.”
EXHIBIT 5 — Evaluation of Queries Q0 and Q1

considered small; however, the execution time would certainly be a major concern. Observe that the Access 97 version Q1 is roughly $10^5$ times slower than Q0 for the same sample.

Exhibit 5 shows the performance of queries Q0 and Q1 for the experimental sample under the Oracle DBMS. The $x$-axis shows the changes in the selectivity factor in steps of 20 percent. The $y$-axis represents time in seconds along a logarithmic scale. The figure suggests a predictable and moderate growth on the time requirements for the execution of the query Q1. For a selectivity ratio of 60 percent or better, the pattern of the samples becomes remarkably similar. Observe that at 80 percent selectivity, the difference in performance between samples of 1000 and 5000 records is a factor of 10. Observe that Q0 remains constant at 1 second for all sample sizes and selectivity factors.

Microsoft Access evaluations improved considerably after the creation of additional indices on the A and B attributes. For example, one of the samples with 2500 T1-records consumes 129,860 seconds on Access97 without the assistance of indices. The same sample requires only 972 seconds to complete when indices are used. The 972 seconds reported also includes the time needed to prepare the indices. Therefore, an improvement of 133 times better performance is observed in Access97. Nonetheless, the action of the Q1–Access combination on the sample holding 5000 records is almost 4000 times slower than the implementation of Q0–Access on the same data set.

Note the rather disconcerting operational behavior on divisions involving a divisor table which is either empty or has a zero selectivity factor. For example, consider the sample with the numerator T1(A,B) holding 2500 records. The Byzantine query Q1 takes more time to oper-
ate on an empty divisor than on a non-empty divisor holding a 20 per-
cent selectivity factor.

The other versions, Q2 and Q3, are operationally much better than 
Q1. However, in each case, they were not as good performers as the sug-
gested version of Q0. Although the results are mixed, Q3 is generally a 
better choice than Q2. In particular, the larger the sample and the stron-
ger the selectivity, the better Q3 becomes.

Zero Division

The Codd's divide operator (1) is defined in such a way that 
T1(A,B)/T2(B) produces exactly all those values in T1(A) each time that 
T2(B) is either empty or has a zero selectivity with respect to T1(A,B). 
The lack of intuitive interpretation for this type of result creates a serious 
philosophical problem.\textsuperscript{13}

Consider the following example. Suppose the table T1(A,B) of Exhibit 
1 is available. As before, it identifies the B-items acquired by the A-cus-
tomers for the new vehicles. Further suppose that a new table T2(B) 
holds the identification of the "orange" car interior decorations (mats, 
steering wheel cover, and seat covers). Assume that either no customer 
has actually acquired such a selection, or that no collection of interior 
decoration exists in that color scheme. However, the expression 
T1(A,B)/T2(B) would in this case produce the entire list of customers.

Now what is the meaning of this result? Obviously, it should not be 
implied that all customers have acquired such a package or have chosen 
a set that does not exist. Perhaps the empty set is a more appropriate an-
swer to the question.

Assume the programmer opts for this interpretation. Therefore, as 
soon as the divisor is known to be empty or disassociated from T1(A,B), 
the empty table must be offered as the result. A simple test could be 
done in the Oracle PL/SQL language. Consider the following fragment:

\begin{verbatim}
FUNCTION emptyDivision RETURN Boolean 
IS 
cursor theDivisor is 
    select * 
    from T1, T2 
    where T1.B = T2.B;

BEGIN 
    Open theDivisor;
    if theDivisor%ROWCOUNT = 0 then
        Return False;
    else
        Return True;
    end if;
END;
\end{verbatim}
In the above code fragment, a cursor called *theDivisor* is produced to compute the result of joining tables T1 and T2. If the size of this table — determined by *theDivisor*%ROWCOUNT — is zero, then either the divisor table T2 is empty, or the relationship between the two tables is empty, leading to a case of zero selectivity factor.

**CONCLUSION**

This article has assessed the performance of several SQL implementations of the *division* operation. The implementations were identified as Q0, Q1, Q2, and Q3. The version Q0 is based on a simple match and count mechanism, while Q1 mimics the predicate calculus definition for division originally suggested by Codd. Q2 and Q3 are closely related to the algebraic implementation of the division using other primitive algebra operators, including projection, selection, cartesian, and subtraction, as well as extensions suggested by Ozsoyoglu.4

The query optimization implicitly performed by Access and Oracle does not seem to be a sufficient effort to close the operational gap between Q0 and Q1. Empirically, one could say that execution of Q1 ranges from bad in Oracle to unacceptable in Access97 (at least the version that operates without indices).

The implementation identified as Q1 proved to be very slow. Surprisingly, this is the format widely suggested by the database literature. Conversely, the version Q0, which is functionally equivalent to Q1, rendered remarkable results. Q0 consistently outperformed the Q1 solution. The execution ratio Q1:Q0 for the sample size of 2500 records was shown to be between 200 to 130,000 times slower.

In addition, Q0 produces a desirable response in the case of empty divisor or zero correlation between the two tables. It was found that Q0 generated an empty table each time either the divisor was empty or the selectivity factor was zero. This result is more consistent with the intuitive expectation of the operator. From a psychological standpoint, it tells the user that no record from T1 is connected to the empty or unrelated collection held in T2. Meanwhile, all other solutions produced for result all the T1[A] values. Although this result is compatible with the original definition of the operator, it is at best misleading — and perhaps wrong.

The results obtained in these experiments suggest the choice of Q0 over all of the other versions. The performance figures seem to indicate that Q0 is independent of the actual DBMS tool used to compute the queries. In addition, the Q0 format seems not to be sensitive to other factors, such as the growing size of the tables and variations of the selectivity factor. The database programmer needs to be aware that in a consumer product DBMS, such as Access97, the execution times of Q1 are unreasonable — even for very small samples — and should be avoided at all costs. Therefore, the choice of Q0 is clearly encouraged.
Bibliography