ON MAKING RELATIONAL DIVISION COMPREHENSIBLE

Lester I. McCann[1]

Abstract *- Division is the most intellectually challenging of the basic operators of relational algebra. Although its utility is unquestioned, its presentation by many textbooks, and therefore by many instructors, leaves much to be desired. This paper examines the standard approaches used to express division in relational algebra and SQL, explains the derivation of each, and presents a coherent sequence of examples that we have used successfully to teach division to students in introductory database classes.*

Index Terms – Division, Relational Algebra, SQL, Relational Database Management Systems.

INTRODUCTION

The formal languages of relational calculus and relational algebra were introduced by Codd as companions for the relational model [1,2]. Due in large part to its procedural nature, relational algebra is a useful stepping-stone for students who are learning the standard relational database management system (DBMS) language SQL, which, like relational calculus, is non-procedural. With an understanding of relational algebra, students can more easily comprehend the processing underlying most SQL expressions [10].

Although we do not have the space to review relational algebra, a brief review of its operators will help our presentation. Readers needing an in-depth introduction may refer to any of a number of suitable database texts (e.g., [3,6,8]).

A relation can be thought of as a two-dimensional matrix of rows (tuples) and columns (attributes). Each tuple's content describes an entity. Operators in relational algebra accept one or two relations as input and produce a relation as output; thus, operator composition is the basic mechanism of expression construction. The five fundamental relational algebra operators are *selection* (represented by the Greek letter σ), *projection* (π), *Cartesian Product* ($\times$), *union* ($\cup$), and *difference* (-). Three additional operators – *join* (|×|, or 'bowtie'), *intersection* ($\cap$), and *division* ($\div$) – can be defined in terms of the first five.

Ask an instructor of a database class which of these eight operators is the most difficult for students to grasp, and he or she will almost certainly reply "division."

Many instructors and even a few textbook authors gloss over division or ignore it completely. This is especially true of their coverage of division's expression in SQL, because SQL has no short-hand representation for division.

We feel that the presentations of the division operator in most database texts lack the detail necessary to help students and instructors understand division and appreciate its utility. We will demonstrate that division is not as mysterious as it might seem.

UNDERSTANDING DIVISION USING RELATIONAL ALGEBRA

Textbook presentations of the division operator usually begin by saying that it is useful for 'certain special kinds of queries,' and continue with an example of an "all" or "for all" query, such as "What are the names of the students taking all of the Computer Science seminar classes?" This is soon followed by a complex description of the sequence of basic relational algebra operations that comprise division, and perhaps another example. After such a 'feet-first' presentation, students are more than ready to ignore division, or worse, to try to find a shortcut way to solve "for all" queries.

Why is it Called 'Division?'

To help students understand division, we start by explaining the origin of its name. They already understand that division is the counterpart of multiplication in arithmetic: 2 * 3 = 6, and so 6 / 3 = 2 and 6 / 2 = 3. We build on that knowledge by asking them to consider two unary relations *m* and *n*:

m	C
	4
	8

n	D
	3
	1
	7

Because we have covered the fundamental relational algebra operators first, students can quickly tell us that $m \times n$ is:

o	C	D
	4	3
	4	1
	4	7
	8	3
	8	1
	8	7

It is not difficult for the students to accept (on faith, for the moment) that $\div$ is $\times$'s counterpart:

[1] Computer Science Department, University of Wisconsin - Parkside, 900 Wood Road, P.O. Box 2000, Kenosha, WI 53141-2000 [mccann at uwp.edu]

$o \div n =$

m	C
	4
	8

$o \div m =$

n	D
	3
	1
	7

We show $o \div n$ first because it is easier to see that the values 4 and 8 are paired with each member of *n*. Of course, the corresponding pairings hold in $o \div m$, but they are not as readily apparent.

Up to this point, students have not seen a formal definition of division, and yet they have a firm grasp on the basic idea: Division is used to identify attribute values of one relation that are associated with *every* member of another relation. With this presentation in mind, the students are ready to see more practical uses for division, as well as to explore its formal definition.

A MORE PRACTICAL EXAMPLE

Due to space considerations, rather than create our own schema and data, we will base our example on the P (parts) and SPJ (relationship) relations of Date's well-known suppliers-parts-projects relational schema and its sample data [3], which is based on examples from Codd's introduction to the relational model [1]. Here are the schemata of P and SPJ:

P	<u>pno</u>	pname	color	weight	city

SPJ	<u>sno</u>	<u>pno</u>	<u>jno</u>	qty

The underlined attributes are the primary keys; the primary key of SPJ is a three-attribute compound key. SPJ's pno field is a foreign key to P's pno field. Similarly, sno and jno are foreign keys to the S (supplier) and J (project) relations, respectively. The schemata of these relations are not provided here, as we do not use them in this paper. Also, we have replaced '#' with 'no' in the key field names, because many DBMSes do not permit field names to include the '#' character.

Consider this query: "Find the sno values of the suppliers that supply all parts of weight equal to 17." With perhaps a refresher on the ×/÷ relationship, students can tell us the schemata of the dividend (α) and divisor (β) relations which are required to set up the division:

α	sno	pno

β	pno

Further, they should be able to state that the relational algebra queries necessary to create those relations are $a \leftarrow p_{sno,pno}(SPJ)$ and $b \leftarrow p_{pno}(s_{weight=17}(P))$. When these queries are applied to Date's sample data, the resulting relations are:

α	sno	pno
	S1	P1
	S2	P3
	S2	P5
	S3	P3
	S3	P4
	S4	P6
	S5	P1
	S5	P2
	S5	P3
	S5	P4
	S5	P5
	S5	P6

β	pno
	P2
	P3

We have removed duplicate values from α, for clarity. With these relations in front of them, students will be able to see that supplier S5 is the only one that supplies both parts P2 and P3.

IMPLEMENTING DIVISION…

If our goal is to have our classes understand what relational division is and how it can be useful, the preceding may be all that is required. However, if we want to show our students how to express division in terms of the fundamental relational algebra operators (and eventually in SQL), we will need to explain a fairly complex query.

…In Relational Algebra

Having defined the relations α and β, we next show our students division in terms of π, ×, and -. The expression is:

$$a \div b = p_{sno}(a) - p_{sno}((p_{sno}(a) \times b) - a)$$

where *sno* is the difference in the schemata of α and β (or, *sno* is the only attribute found in α but not in β). Texts typically present this expression in completely generic terms, which only seems to confuse our students. We prefer to introduce it in terms of this concrete suppliers-parts-projects example.

Even with our preliminary discussion of the construction of α and β, very few students will be able to understand the logic underlying that query without help. We explain this expression using a top-down approach. At the top level, the expression has the form $C - D$, where $C = p_{sno}(a)$ and $D = p_{sno}((p_{sno}(a) \times b) - a)$. In the end, we want to produce a set of *sno* values, and C is easily seen to be the set of available *sno* values. Because the operator is set difference, to get the desired result D must be a set of *sno* values that are *not* desirable. This, then, is the plan: Given a set of possible answers, remove from it those that are not answers; the tuples that remain must be the answers.

How do we determine which *sno* values we are *not* interested in keeping? Again, it helps to think about it backwards: In the end, we want to keep the *sno* values that, within α, are paired with all of members of β; thus, the *sno* values that we *do not* want are those that, when we examine α, we *do not* find paired with all of the members of β. Now, if we just happen to have a group of all possible *sno, pno* pairings and we remove the ones that also exist in α, we will be left with the pairs that we did not find in α. This is easily done: We create all possible pairings of α's *sno* values and β's *pno* values $(p_{sno}(a) \times b)$ and use set difference to remove from those pairings any pairs that we find in α.

When presenting the relational algebra expression for division, we like to begin by presenting the sample data and allowing the students to manually find the result of the division through inspection of the dividend and divisor relations. However, finding the answer by inspection is a potential problem, one which we address in a later section.

After the students know the result, we work through the division expression on the same data and show most of the intermediate relations produced by the expression. This helps the students focus on how the steps of the expression work together to produce the desired result.

Using Date's data, here is the walk-through for our example. The expression $p_{sno}(a) \times b$ produces the possible *sno, pno* pairings:

$p_{sno}(a)$	sno
	S1
	S2
	S3
	S4
	S5

×

β	pno
	P2
	P3

=

γ	sno	pno
	S1	P2
	S1	P3
	S2	P2
	S2	P3
	S3	P2
	S3	P3
	S4	P2
	S4	P3
	S5	P2
	S5	P3

Using set difference, we remove from γ the pairings that already exist in α (the 'victims' are shaded):

γ	sno	pno
	S1	P2
	S1	P3
	S2	P2
	S2	P3
	S3	P2
	S3	P3
	S4	P2
	S4	P3
	S5	P2
	S5	P3

-

α	sno	Pno
	S1	P1
	S2	P3
	S2	P5
	S3	P3
	S3	P4
	S4	P6
	S5	P1
	S5	P2
	S5	P3
	S5	P4
	S5	P5
	S5	P6

=

δ	sno	pno
	S1	P2
	S1	P3
	S2	P2
	S3	P2
	S4	P2
	S4	P3

Note that all of γ's S5 tuples were 'victims.' This means that S5 is the only supplier that supplies all parts of weight 17. Removing γ's *sno* values from the list of available *sno* values will produce the result of the division:

$p_{sno}(a)$	sno
	S1
	S2
	S3
	S4
	S5

-

$p_{sno}(d)$	sno
	S1
	S2
	S3
	S4

=

÷	sno
	S5

One of the difficulties in giving a class a feel for relational algebra is that commercial DBMSes do not accept relational algebra queries. We have had success using the relational algebra based LEAP DBMS [5] to help students familiarize themselves with relational algebra in general and division in particular. LEAP does not have a built-in division operator, the lack of which forces students to work through the implementation of division on their own.

… In SQL

After acquiring a relational algebra foundation, students have little trouble understanding how the parts of basic SQL queries correspond to the fundamental relational algebra operators. As in relational algebra, in SQL it is necessary to construct a division query manually. The resulting SQL queries are easily as confusing as the relational algebra expression. However, with the relational algebra version behind them, students are better prepared to understand an SQL version or two. As an additional benefit, concepts such as sub-queries and nested correlated queries can be demonstrated at the same time.

Although efficient division algorithms exist [4], we know of no database management systems that recognize division queries and use a specific algorithm for their evaluation. This decision is understandable, given that division queries expressed in SQL would be difficult to detect and are uncommon when compared with other expensive operators (e.g., joins). One nice effect of this is that instructors can use their DBMS's query plan explanation option to explore how a DBMS executes a division query. This investigation can lead to an in-class discussion of cases other than division in which one might wish to extract data from a DBMS and process it with a special-purpose algorithm.

In the following subsections, we present four SQL implementations of division. We do not advocate presenting all of them during lecture. Instead, we suggest that instructors select for presentation one or two that integrate naturally into the students' relational algebra and SQL backgrounds.

Direct Application of the Expression

Perhaps the most approachable formulation of division in SQL is through a direct translation of the relational algebra expression. This translation is easily performed with a bottom-up approach.

Relations α and β can be created using the SQL queries **`select sno, pno from spj`** and **`select pno from p where weight = 17`**, respectively. These relations could be materialized as temporary relations in advance, or could be embedded in the translation as dynamically created sub-queries. We opt for the latter, as we feel the introduction of temporary relations does not add appreciably to the clarity of the example.

Recall that the relational algebra expression for division is in two parts, $C - D$, and that $D = p_{sno}((p_{sno}(a) \times b) - a)$. Creating $p_{sno}(a) \times b$ in SQL requires the use of sub-queries in the **`from`** clause of a **`select`** statement:

```
select sno, pno
  from (select sno from spj) as t1,
       (select pno from p
         where weight=17) as t2;
```

When sub-queries are used in the **`from`** clause, most DBMSes (including the one used for these examples, PostgreSQL v. 7.2.1 [7]) require the use of aliases to label the sub-queries. A **`select`** with no **`where`** clause produces a Cartesian Product of the supplied relations.

Modern versions of SQL provide **`except`** as the equivalent of the relational algebra set difference operator. The presence of **`except`** makes the creation of the rest of the translation straight-forward:

```
select distinct sno from spj
except
select sno
  from ( select sno, pno
           from (select sno from spj) as t1,
                (select pno from p
                  where weight=17) as t2
         except
         select sno, pno
           from spj
       ) as t3;
```

In PostgreSQL, a query plan explanation can be displayed by adding the keyword **`explain`** at the front of the query. When run on a database containing Date's sample data, an explanation of this expression revealed a complex plan that includes a nested-loops join, two sorts, three base-relation scans, and five sub-query scans. Thus, a direct translation does not result in an inexpensive execution. In spite of this, we feel that the connection to the relational algebra version makes it worth presenting to students who have already been exposed to relational algebra. Note that the exact plan chosen by a query optimizer depends on a variety of factors, including past queries presented to the database and the sizes of the relations.

Through the Application of Logical Equivalences

The cost of execution of the direct translation helps explain why mainstream texts usually opt for formulations resulting from the application of quantification tautologies.

Consider the tautology $\forall a(\exists b\ f(a,b)) \leftrightarrow \overline{\exists} a(\overline{\exists} b\ f(a,b))$. We can use it to replace a universal quantification (which SQL does not support) with the negation of an existential quantification, which SQL provides (**`not exists`**). The effect is to convert a query of the form "Find *sno* values such that <u>for all</u> parts of weight 17 <u>there exist</u> suppliers that supply them all" into one of the form "Find *sno* values such that <u>there do not exist</u> any parts of weight 17 for which <u>there do not exist</u> any suppliers that supply them all." We are exchanging inexpressibility in SQL for a double negation, which most people find difficult to comprehend. The resulting 'double $\overline{\exists}$' SQL query looks like this:

```
select distinct sno from spj as globl
 where not exists
      ( select pno from p
          where weight = 17 and not exists
               ( select * from spj as locl
                   where locl.pno = p.pno
                     and locl.sno = globl.sno));
```

Because PostgreSQL v. 7.2.1 complains about the use of the identifiers **`global`** and **`local`** as aliases, we have used obvious misspellings in our queries.

It is essential for students to understand that, in a correlated nested query such as this, the aliases (a.k.a. tuple variables) have scope across parts of the query based on their declaration position just as programming language variables have scope across code. In this example, the alias **`globl`** represents a unique tuple from relation SPJ for each execution of the innermost **`select`**. Further, **`globl`** is accessible, as its name suggests, from all parts of the query. By contrast, **`locl`**'s scope is limited to just the innermost **`select`**.

Our experience has been that students find this approach to be difficult to understand because of the double-negative construction and the disconnect from the logic of the relational algebra expression. It is no surprise that at least one textbook resorts to a 'cookbook' approach for the construction of division queries based on this tautology [9].

An alternate approach uses the idea of set containment (i.e., superset). If a supplier supplies a superset of the parts of weight 17, the supplier clearly supplies them all. SQL does not have a containment operator (though it was once proposed), but logic comes to our rescue again: If $A \supseteq B$, $B-A$ will be empty (or, $\overline{\exists}(B-A)$). Here, *A* is the set of parts of weight 17 that a supplier supplies, and *B* is the set of all available parts of weight 17. The resulting SQL query scans through the *sno* values, computes *A* based on the current *sno* value, and includes the *sno* value in the quotient if the difference is empty:

```
select distinct sno from spj as globl
 where not exists (
      ( select pno from p where weight = 17 )
        except ( select p.pno from p, spj
                   where p.pno = spj.pno
                     and spj.sno = globl.sno ));
```

The logic of this version does not bother students greatly. If students are not familiar with relational algebra, this version of division can be more easily introduced than the direct translation and is easier for students to understand than the double-negative representation.

The plans generated by PostgreSQL for these two formulations both require fewer operations than does the direct expression translation version, with the `double $\overline{\exists}$' version having by far the fewest.

A Counting We Will Go

Tuples resulting from the application of a division operator exist because their data corresponds with all of the values in another relation. The superset approach uses set containment to count (in effect) the number of members in two sets in hopes of finding the sums to be equal. Having observed that the result is being determined by the cardinality of these sets, we can construct a fourth approach to the expression of division in SQL using the **`count()`** aggregate function.

The idea is to find the *sno* values of the suppliers that supply parts of weight 17 and how many of those parts each supplies. This is easily accomplished with a join of P and SPJ that counts the *pno* values of weight 17 associated with each *sno* value. A **`having`** clause compares each count to the result of a sub-query that finds the total number of parts of weight 17. The resulting SQL query's plan is not as efficient as that of the `double $\overline{\exists}$' version, but it is better than the other two, and may be more easily understood.

```
   select distinct sno from spj, p
    where spj.pno = p.pno and weight = 17
 group by sno
   having count(distinct p.pno) =
          (select count (distinct pno)
             from p where weight = 17);
```

SOME DIVISION PITFALLS

As explained in most database texts, division is used to answer "all" or "for all" queries, such as the "What are the names of the students taking all of the Computer Science seminar classes?" example mentioned at the start of the second section of this paper. The inherent suggestion is that division is the equivalent of the universal quantifier (∀) of the relational calculus family of formal relational languages.

This suggestion does not withstand close scrutiny. The seminar query just mentioned can be used to demonstrate a key problem. The query is to return the names of the students who are enrolled in all of the CS seminar classes being offered (during the current term, presumably). Clearly, we need a relation with name and course identification attributes, and another with the course identifications of the seminar classes:

enroll	name	class

seminar	class

Dividing *enroll* by *seminar* gives the desired result, unless *seminar* is an empty relation! Logically, if no seminar classes are being offered, any student is taking all of the seminar classes. This result is not satisfying and is not in keeping with the spirit of the original query. More accurately, the query that division is answering in this case is "What are the names of the students taking all of the Computer Science seminar classes, *assuming that at least one is being offered*?". Because the divisor relation is often the result of a sub-query, the divisor could easily be empty. Students should be made aware of this possibility.

A second problem with the "for all" wording is that students, desperate to avoid dealing with division, sometimes work hard to invent new interpretations of the meaning of queries in order to make less confusing operators seem suitable.

Consider this slight variation on our suppliers-parts-projects query: "Find the *sno* values of the suppliers that supply all parts of weight equal to 19." Having seen how to find the answer by inspection, with Date's sample data available students soon discover that suppliers S4 and S5 both supply all of the parts having a weight of 19. Rather than formulate a logically sound query using division, some students cannot resist the temptation to construct a simpler but incorrect query that happens to produce the correct answer:

```
select distinct sno
  from (select sno, pno from spj) as one,
       (select pno from p
         where weight = 19) as two
 where one.pno = two.pno;
```

This is merely a join of the α and β relations followed by a projection on *sno*, but the resulting relation has the same content as the result of the correct division query. What the students have constructed is a query to answer the question "Find the *sno* values of the suppliers that supply parts of weight equal to 19," which is the original question without the "all."

Instructors can help students avoid temptation by selecting queries and sample data that produce a β relation with more than one tuple. In this example, only part P6 is of weight 19. If we attempt the same logically incorrect join query with our original weight of 17, the result will contain three suppliers - S2, S3, and S5 – because all three supply at least one of the two parts of weight 17. Division produces only S5, as we saw earlier. Although this suggestion merely reduces the odds that the two types of queries will having matching results, we have found it to be an effective way for students to discover the importance of division on their own.

CONCLUSION

The relational algebra division operator need not be the source of confusion that many database students feel it to be. With a bit of time and a coherent progression of well-chosen examples, students can learn when to apply it, how to express it in SQL, and why those expressions are correct. We find our approach to be more effective and complete than those found in most database textbooks.

REFERENCES

[1] Codd, E. F., "A relational model of data for large shared data banks," *Communications of the ACM*, 13, 6, 1970, pp. 377-387.

[2] Codd, E. F., "Relational completeness of data base sublanguages," in *Data Base Systems* (Courant Computer Science Symposium 6), Rustin, R., editor, Prentice-Hall, 1972.

[3] Date, C. J., *An Introduction to Database Systems*, Addison-Wesley, seventh edition, 2000.

[4] Graefe, G., "Relational division: Four algorithms and their performance," *Proceedings of the IEEE International Conference on Data Engineering*, 1989, pp. 94-101.

[5] Leyton, R., *LEAP*, http://leap.sourceforge.net/, 2002.

[6] O'Neil, P., and O'Neil, E., *Database -- Principles, Programming, and Performance*, Academic Press, second edition, 2001.

[7] PostgreSQL Global Development Group. *PostgreSQL*, http://www.postgresql.org/, 2002.

[8] Ramakrishnan, R., and Gehrke, J., *Database Management Systems*, McGraw-Hill, third edition, 2003.

[9] Watson, R.T., *Data Management: Databases and Organizations*, John Wiley and Sons, Inc., third edition, 2002.

[10] Welty, C., and Stemple, D.W., "Human factors comparison of a procedural and a non-procedural query language," *ACM Transactions on Database Systems*, 6, 4, 1981, pp. 626-649.