IDERA® | Database Tools Whitepaper

RELATIONAL DIVISION

BY JOE CELKO

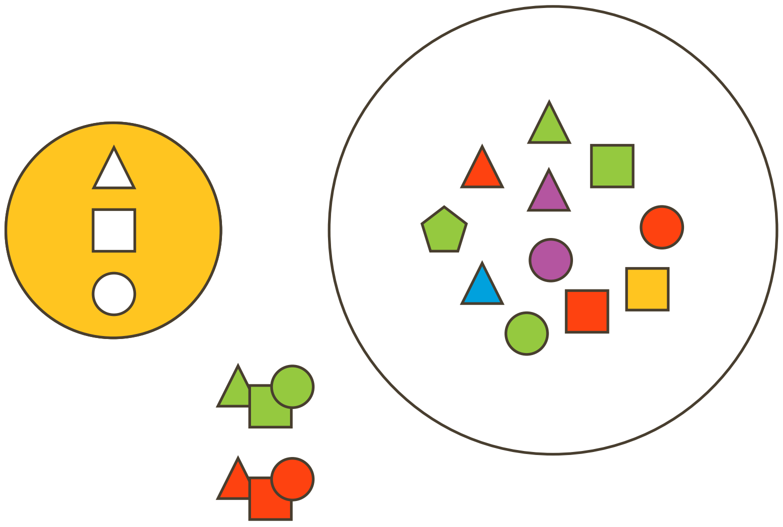
INTRODUCTION

Dr. Codd's original relational algebra had eight basic operations. Since RDBMS is based on set theory, the first four are traditional set operations: intersection, set difference, union, and product. These operations are available in SQL, respectively, as INTERSECT, EXCEPT, UNION, and CROSS JOIN.

The next four are row-oriented: restriction, projection, (natural) join, and divide. These operations are available in SQL, respectively, as rows picked with a WHERE or ON clause, the column list in a SELECT list, a simple INNER JOIN..ON operator and, well, we do not have a simple divide in SQL!

SQL also has several OUTER JOINs, OUTER UNION, variants of the ON clause, and the multi-set INTERSECT ALL, EXCEPT ALL and UNION ALL extensions. But we never added relational division. It can be written with the other operators, and it turns out that it is not so simple after all.

The idea of relational division is that a divisor table is used to partition a dividend table and produce a quotient or results table. The quotient table is made up of those values of one column for which a second column had all of the values in the divisor.



Divide colored shapes by {triangle, square, circle}
to get {red, green}

RELATIONAL DIVISION OPERATORS

When I teach this operator, I use colored foam shape tiles used in elementary school to teach naive set theory and counting. You can get them at any school supply house. Pull out a set of tiles, then draw an outline of several of them on paper.

In this diagram, I have a set with squares, circles, triangles, and pentagons in red, yellow, green, blue, and purple.

I then make a “divisor” with the set of {square, circle, triangle}; it is just a piece of paper with outlines on it.

```
CREATE TABLE Colored_Shapes --dividend

(color_name CHAR(10) NOT NULL,

shape_name CHAR(10) NOT NULL,

PRIMARY KEY (color_name, shape_name))
```

COLORED SHAPES

COLOR NAME	SHAPE NAME
Green	Triangle
Red	Triangle
Green	Square
Green	Pentagon
Purple	Triangle
Red	Circle
Blue	Triangle
Purple	Circle
Yellow	Square
Green	Circle
Red	Square

```
CREATE TABLE Shapes --divisor

(shape_name CHAR(10) NOT NULL PRIMARY KEY);
```

SHAPE NAME
Triangle
Square
Circle

```
Colored Shapes DIVIDED BY Shapes --quotient
```

COLOR NAME
Red
Green

I first put the tiles into piles by colors. I pick up each such pile and see if I can match a tile to the outlines on the divisor paper. And I can do this for Red and Green tiles; pick them up and physically try it.

The important characteristic of a relational division is that the CROSS JOIN of the divisor and the quotient produces a valid subset of rows from the dividend. This is where the name comes from, since the CROSS JOIN acts “kind of” like a multiplication operator and the symbol in relational algebra is the simple cross that is also used for multiplication.

DIVISION WITH REMAINDER

There are two kinds of relational division. Division with a remainder allows the dividend table to have more values than the divisor, which was Dr. Codd's original definition. For example, if a color pile has more tiles than just those we have in the divisor, such as the green pentagon, this is fine with us. This is the remainder. See the analogy to simple division from grade school?

The query can be written as:

```
SELECT DISTINCT color_name
  FROM Colored_Shapes AS CS1
 WHERE NOT EXISTS
  (SELECT *
    FROM Shapes
    WHERE NOT EXISTS
      (SELECT *
      FROM Colored_Shapes AS CS2
      WHERE (CS1.color_name = CS2.color_name)
      AND (CS2.shape_name = Shapes.shape_name)));
```

In English, this says, "There is no shape in the divisor that I cannot match in the dividend," a sort of double negative. Not great English, but good logic. The use of the NOT EXISTS() predicates is for speed. Most SQL implementations will look up a value in an index rather than scan the whole table. This query for relational division was made popular by Chris Date in his textbooks, but it is neither the only method nor always the fastest. Another version of the division can be written so as to avoid three levels of nesting. While it is not original with me, I have made it popular in my books.

```
SELECT CS1.color_name
    FROM Colored_Shapes AS CS1, Shapes AS S1
  WHERE CS1.shape_name = S1.shape_name
  GROUP BY CS1.color_name
HAVING COUNT(CS1.shape_name)
    = (SELECT COUNT(shape_name) FROM Shapes);
```

In English, this is a one-to-one mapping. The inner join puts each element of my divisor with an element of the dividend (if it exists). If the count of elements in the divisor is the same as the count of matched elements, then this color is in the quotient. Think about an aborigine putting arrowheads and sea shells in pairs. If he has arrowheads left over, then he know he has more arrowheads. If he has sea shells left over, then he knows he has more sea shells.

If there is no surplus of either, then the sets have the same cardinality. Notice there is no concept of a number in this operation; that is, his math is so limited he cannot say anything like, "I have 34 more sea shells than arrowheads," with this mapping concept.

There is a serious difference in the two methods. Burn the paper with the Shapes, so that the divisor is empty. Because of the NOT EXISTS() predicates in Date's query, all colors are returned from a division by an empty set. Because of the COUNT() functions in my query, no colors are returned from a division by an empty set.

Currently in its eighth edition, in the book Introduction to Database Systems, author Chris Date defined another operator (DIVIDEBY ... PER) which produces the same results as my query, but with more complexity. The philosophical question is, should a relational division by an empty set mimic the behavior of a numeric division by zero in some way, or be more "set-oriented" in its outcome?

EXACT DIVISION

The second kind of relational division is exact relational division. The dividend table must match exactly to the values of the divisor without any extra values.

```
SELECT CS1.color_name
  FROM Colored_Shapes AS CS1
    LEFT OUTER JOIN
    Shapes AS S1
    ON CS1.shape_name = S1.shape_name
  GROUP BY CS1.color_name
HAVING COUNT(CS1.shape_name)
    = (SELECT COUNT(shape_name) FROM Shapes)
  AND COUNT(S1.shape_name)
    = (SELECT COUNT(shape_name) FROM Shapes);
```

The LEFT OUTER JOIN will create NULL-padded rows if the Colored_Shapes dividend is larger than the Shapes divisor. If there are no extra tiles, then both the dividend and the divisor are equal in size. Please do not make the mistake of trying to reduce the HAVING clause with a little algebra to:

```
HAVING COUNT(CS1.shape_name) = COUNT(S1.shape_name)
```

because it does not work; it will tell you that the Shapes has (n) shape_name in it and the color_name is certified for (n) shape_name, but not that those two sets of shape_name are equal to each other.

NOTE ON PERFORMANCE

As mentioned previously, the nested EXISTS() predicates version of relational division was made popular by Chris Date's textbooks, while this author is associated with popularizing the COUNT(*) version of relational division. The Winter 1996 edition of the now-defunct Db2 online magazine had an article entitled "Powerful SQL: Beyond the Basics" by Sheryl Larsen which gave the results of testing both methods. Her conclusion for the then-current version of Db2 was that the nested EXISTS() version is better when the quotient has less than 25% of the dividend table's rows and the COUNT(*) version is better when the quotient is more than 25% of the dividend table.

On the other hand, Matthew W. Spaulding at SnapOn Tools reported his test on SQL Server 2000 with the opposite results. He had a table with two million rows for the dividend and around 1,000 rows in the divisor, yielding a quotient of around 1,000 rows. The COUNT method completed in well under one second, whereas the nested NOT EXISTS() query took roughly five seconds to run. The moral of the story is to test both methods on your particular release of your product, and draw your own conclusions.

TODD'S DIVISION

A relational division operator proposed by Stephen Todd is defined on two tables with common columns that are joined together, dropping the JOIN column and retaining only those non-JOIN columns that meet a criterion.

We are given a table, JobParts(job_nbr, part_nbr), and another table, SupParts(sup_nbr, part_nbr), of suppliers and the parts that they provide. We want to get the supplier and job_nbr pairs such that supplier sn supplies all of the parts needed for job_nbr jn. This is not quite the same thing as getting the supplier and job_nbr pairs such that job_nbr jn requires all of the parts provided by supplier sn.

You want to divide the JobParts table by the SupParts table. A rule of thumb: The remainder comes from the dividend, but all values in the divisor are present.

JobParts

job_nbr	part_nbr
j1	p1
j1	p2
j2	p2
j2	p4
j2	p5
j3	p2

SupParts

sup_nbr	part_nbr
s1	p1
s1	p2
s1	p3
s1	p4
s1	p5
s1	p6
s2	p1
s2	p2
s3	p2
s4	p2
s4	p4
s4	p5

Result = JobSups

job_nbr	sup_nbr
j1	s1
j1	s2
j2	s1
j2	s4
j3	s1
j3	s2
j3	s3
j3	s4

Pierre Mullin submitted the following query to carry out the Todd division:

```
SELECT DISTINCT JP1.job_nbr, CS1.supplier
  FROM JobParts AS JP1, SupParts AS CS1
WHERE NOT EXISTS
    (SELECT *
      FROM JobParts AS JP2
    WHERE JP2.job_nbr = JP1.job_nbr
  AND JP2.part
    NOT IN (SELECT SP2.part
      FROM SupParts AS SP2
      WHERE SP2.supplier = CS1.supplier));
```

This is really a modification of the query for Codd's division, extended to use a JOIN on both tables in the outermost SELECT statement. The IN predicate for the second subquery can be replaced with a NOT EXISTS() predicate; it might run a bit faster, depending on the optimizer.

Another related query is finding the pairs of suppliers who sell the same parts. In this data, that would be the pairs (s1, p2), (s3, p1), (s4, p1), and (s5, p1).

```
SELECT S1.sup, S2.sup
  FROM SupParts AS S1, SupParts AS S2
WHERE S1.sup < S2.sup -- different suppliers
  AND S1.part = S2.part -- same parts
GROUP BY S1.sup, S2.sup
HAVING COUNT(*)
    = (SELECT COUNT (*) -- same count of parts
      FROM SupParts AS S3
      WHERE S3.sup = S1.sup)
        AND COUNT(*)
          = (SELECT COUNT (*)
            FROM SupParts AS S4
            WHERE S4.sup = S2.sup);
```

This can be modified into Todd's division easily by adding the restriction that the parts must also belong to a common job.

Steve Kass came up with a specialized version that depends on using a numeric code. Assume we have a table that tells us which players are on which teams.

```
CREATE TABLE Team_Assignments
(player_id INTEGER NOT NULL
REFERENCES Players(player_id)
  ON DELETE CASCADE
  ON UPDATE CASCADE,
team_id CHAR(5) NOT NULL
REFERENCES Teams(team_id)
  ON DELETE CASCADE
  ON UPDATE CASCADE,
PRIMARY KEY (player_id, team_id));
```

To get pairs of Players on the same team:

```
SELECT P1.player_id, P2.player_id
  FROM Players AS P1, Players AS P2
WHERE P1.player_id < P2.player_id
GROUP BY P1.player_id, P2.player_id
HAVING P1.player_id + P2.player_id
  = ALL (SELECT SUM(P3.player_id)
    FROM Team_Assignments AS P3
    WHERE P3.player_id
      IN (P1.player_id, P2.player_id)
    GROUP BY P3.team_id);
```

DIVISION WITH SET OPERATORS

The Standard SQL set difference operator, EXCEPT, can be used to write a very compact version of Dr. Codd's relational division. The EXCEPT operator removes the divisor set from the dividend set. If the result is empty, we have a match; if there is anything left over, it has failed. Using the Colored_Shapes table example, we would write:

```
SELECT DISTINCT color_name
  FROM Colored_Shapes AS CS1
  WHERE (SELECT shape_name FROM Shapes
  EXCEPT
  SELECT shape_name
    FROM Colored_Shapes AS CS2
    WHERE CS1.color_name = CS2.color_name) IS NULL;
```

Again, informally, you can imagine that we got a list from each color_name, walked over to the Shapes, and crossed off each shape_name we could match. If we marked off all the shape_name in the Shapes, we would keep this guy. Another trick is that an empty subquery expression returns a NULL, which is how we can test for an empty set. The WHERE clause could just as well have used a NOT EXISTS() predicate instead of the IS NULL predicate.

ROMLEY'S DIVISION

This somewhat complicated relational division is due to Richard Romley, a DBA retired from Solomon Smith Barney. The original problem deals with two tables. The first table has a list of managers and the projects they can manage. The second table has a list of Personnel, their departments and the project to which they are assigned. Each employee is assigned to one and only one department and each employee works on one and only one project at a time. But a department can have several different projects at the same time, so a single project can span several departments.

```
CREATE TABLE Mgr_Projects
(mgr_name CHAR(10) NOT NULL,
  project_id CHAR(2) NOT NULL,
  PRIMARY KEY(mgr_name, project_id));

INSERT INTO Mgr_Project
VALUES ('M1', 'P1'), ('M1', 'P3'),
  ('M2', 'P2'), ('M2', 'P3'),
  ('M3', 'P2'),
  ('M4', 'P1'), ('M4', 'P2'), ('M4', 'P3');

CREATE TABLE Personnel
(emp_id CHAR(10) NOT NULL,
  dept_id CHAR(2) NOT NULL,
  project_id CHAR(2) NOT NULL,
  UNIQUE (emp_id, project_id),
  UNIQUE (emp_id, dept_id),
  PRIMARY KEY (emp_id, dept_id, project_id));

-- load department #1 data
INSERT INTO Personnel
VALUES ('Al', 'D1', 'P1'),
  ('Bob', 'D1', 'P1'),
  ('Carl', 'D1', 'P1'),
  ('Don', 'D1', 'P2'),
  ('Ed', 'D1', 'P2'),
  ('Frank', 'D1', 'P2'),
  ('George', 'D1', 'P2');
```

```
-- load department #2 data
INSERT INTO Personnel
VALUES ('Harry', 'D2', 'P2'),
  ('Jack', 'D2', 'P2'),
  ('Larry', 'D2', 'P2'),
  ('Mike', 'D2', 'P2'),
  ('Nat', 'D2', 'P2');

-- load department #3 data
INSERT INTO Personnel
VALUES ('Oscar', 'D3', 'P2'),
  ('Pat', 'D3', 'P2'),
  ('Rich', 'D3', 'P3');
```

The problem is to generate a report showing for each manager of each department whether he is qualified to manage none, some or all of the projects being worked on within the department. To find who can manage some, but not all, of the projects, use a version of relational division.

```
SELECT M1.mgr_name, P1.dept_id_name
 FROM Mgr_Projects AS M1
    CROSS JOIN
    Personnel AS P1
 WHERE M1.project_id = P1.project_id
 GROUP BY M1.mgr_name, P1.dept_id_name
HAVING COUNT(*) <> (SELECT COUNT(emp_id)
    FROM Personnel AS P2
   WHERE P2.dept_id_name = P1.dept_id_name);
```

The query is simply a relational division with a <> instead of an = in the HAVING clause. Richard came back with a modification of my answer that uses a characteristic function inside a single aggregate function.

```
SELECT DISTINCT M1.mgr_name, P1.dept_id_name
  FROM (Mgr_Projects AS M1
    INNER JOIN
    Personnel AS P1
    ON M1.project_id = P1.project_id)
    INNER JOIN
    Personnel AS P2
    ON P1.dept_id_name = P2.dept_id_name
GROUP BY M1.mgr_name, P1.dept_id_name, P2.project_id
HAVING MAX (CASE WHEN M1.project_id = P2.project_id
      THEN ‘T’ ELSE ‘F’ END)
    = ‘F’;
```

This query uses a characteristic function while my original version compares a count of Personnel under each manager to a count of Personnel under each project_id. The use of “GROUP BY M1.mgr_name, P1.dept_id_name, P2.project_id” with the “SELECT DISTINCT M1.mgr_name, P1.dept_id_name” is really the tricky part in this new query. What we have is a three-dimensional space with the (x, y, z) axis representing (mgr_name, dept_id_name, project_id) and then we reduce it to two dimensions (mgr_name, dept_id) by seeing if Personnel on shared project_ids covers the department or not.

That observation led to the next change. We can build a table that shows each combination of manager, department and the level of authority they have over the projects they have in common. That is the derived table T1 in the following query; (authority = 1) means the manager is not on the project and (authority = 2) means that he is on the project_id.

```
SELECT T1.mgr_name, T1.dept_id_name,
  CASE SUM(T1.authority)
  WHEN 1 THEN ‘None’
  WHEN 2 THEN ‘All’
  WHEN 3 THEN ‘Some’
  ELSE NULL END AS authority_scope
FROM (SELECT DISTINCT M1.mgr_name, P1.dept_id_name,
      MAX (CASE WHEN M1.project_id = P1.project_id
        THEN 2 ELSE 1 END) AS authority
    FROM Mgr_Projects AS M1
  CROSS JOIN
  Personnel AS P1
  GROUP BY m.mgr_name, P1.dept_id_name, P1.project_id) AS T1
GROUP BY T1.mgr_name, T1.dept_id_name;
```

We can now sum the authority numbers for all the projects within a department to determine the power this manager has over the department as a whole. If he had a total of one, he has no authority over Personnel on any project in the department. If he had a total of two, he has power over all Personnel on all projects in the department. If he had a total of three, he has both a 1 and a 2 authority total on some projects within the department. Here is the final answer.

RESULTS

mgr_name	dept_id	authority_scope
M1	D1	Some
M1	D2	None
M1	D3	Some
M2	D1	Some
M2	D2	All
M2	D3	All
M3	D1	Some
M3	D2	All
M3	D3	Some
M4	D1	All
M4	D2	All
M4	D3	All

CONCLUSION AND A PROGRAMMING PROBLEM

As you can see, there are several kinds of relational division. We have not touched on pulling out subsets based on the counts of various elements in the divisor and dividend. For example, if we define a widget as three circles and two squares, then we need to add a quantity column to both of the tables.

Now use what you have seen and write the SQL for this kind of problem:

```
CREATE TABLE Colored_Shapes --dividend
(color_name CHAR(10) NOT NULL,
shape_name CHAR(10) NOT NULL,

PRIMARY KEY (color_name, shape_name),
onhand_qty INTEGER NOT NULL
    CHECK (onhand_qty > 0));
```

color_name	shape_name	onhand_qty
Blue	Triangle	4
Green	Triangle	3
Green	Square	2
Green	Pentagon	1
Green	Circle	5
Purple	Circle	2
Purple	Triangle	1
Red	Triangle	2
Red	Circle	2
Red	Square	2
Yellow	Square	23

HINT: We know immediately that a color with fewer than three circles or fewer than two squares is disqualified from the results.

ABOUT THE AUTHOR

Joe Celko is the author of a series of ten books on SQL and RDBMS (MKP/Elsevier) that have been in print for over 20 years. He served for 10 years on the ANSI/ISO database standards committee. He has written columns and articles for the IT trade press for over 30 years. He currently enjoys being a TEALS volunteer and judging the local High School Science Fest once a year.