Relational Division

RELATIONAL DIVISION IS one of the eight basic operations in Codd's relational algebra. The idea 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.

This is easier to explain with an example. We have a table of pilots and the planes they can fly (dividend); we have a table of planes in the hangar (divisor); we want the names of the pilots who can fly every plane (quotient) in the hangar. To get this result, we divide the Pilot_Skills table by the planes in the hangar.

```
CREATE TABLE Pilot_Skills
(pilot_name CHAR(15) NOT NULL,
 plane_name CHAR(15) NOT NULL,
 PRIMARY KEY (pilot_name, plane_name));
```

Pilot_Skills

pilot_name	plane_name
'Celko'	'Piper Cub'
'Higgins'	'B-52 Bomber'
'Higgins'	'F-14 Fighter'

pilot_name	plane_name
'Higgins'	'Piper Cub'
'Jones'	'B-52 Bomber'
'Jones'	'F-14 Fighter'
'Smith'	'B-1 Bomber'
'Smith'	'B-52 Bomber'
'Smith'	'F-14 Fighter'
'Wilson'	'B-1 Bomber'
'Wilson'	'B-52 Bomber'
'Wilson'	'F-14 Fighter'
'Wilson'	'F-17 Fighter'

```
CREATE TABLE Hangar
(plane_name CHAR(15) NOT NULL PRIMARY KEY);
```

Hangar

plane_name
'B-1 Bomber'
'B-52 Bomber'
'F-14 Fighter'

Pilot_Skills DIVIDED BY Hangar

pilot_name
'Smith'
'Wilson'

In this example, Smith and Wilson are the two pilots who can fly everything in the hangar. Notice that Higgins and Celko know how to fly a Piper Cub, but we don't have one right now. In Codd's original definition of relational division, having more rows than are called for is not a problem.

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 like a multiplication operator.

34.1 Division with a 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 pilot_name can fly more planes than just those we have in the hangar, this is fine with us. The query can be written as

```
SELECT DISTINCT pilot_name
  FROM Pilot_Skills AS PS1
 WHERE NOT EXISTS
       (SELECT *
          FROM Hangar
         WHERE NOT EXISTS
               (SELECT *
                  FROM Pilot_Skills AS PS2
                 WHERE (PS1.pilot_name = PS2.pilot_name)
                   AND (PS2.plane_name = Hangar.plane_name)));
```

The quickest way to explain what is happening in this query is to imagine a World War II movie where a cocky pilot_name has just walked into the hangar, looked over the fleet, and announced, "There ain't no plane in this hangar that I can't fly!" We want to find the pilots for whom there does not exist a plane in the hangar for which they have no skills. 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 PS1.pilot_name
  FROM Pilot_Skills AS PS1, Hangar AS H1
 WHERE PS1.plane_name = H1.plane_name
 GROUP BY PS1.pilot_name
HAVING COUNT(PS1.plane_name)
       = (SELECT COUNT(plane_name) FROM Hangar);
```

There is a serious difference in the two methods. Burn down the hangar, so that the divisor is empty. Because of the NOT EXISTS() predicates in Date's query,

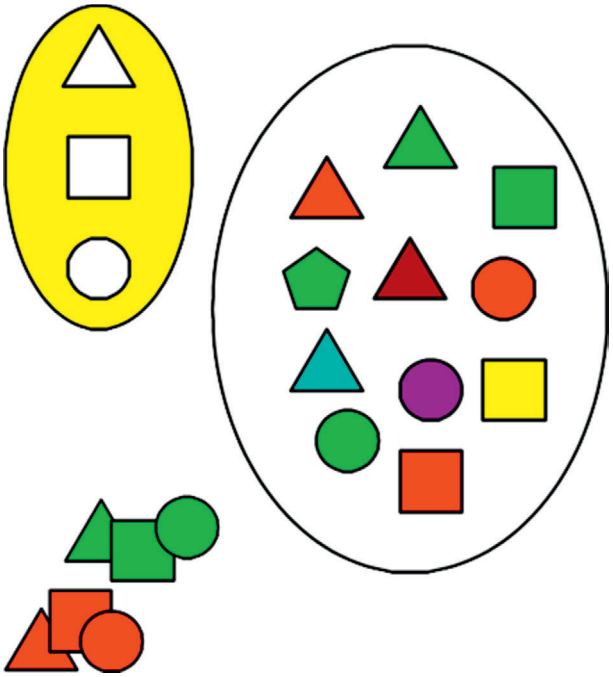

Figure 34.1 Relational Division.

all pilots are returned from a division by an empty set. Because of the COUNT() functions in my query, no pilots are returned from a division by an empty set.

In the sixth edition of his book, INTRODUCTION TO DATABASE SYSTEMS (Addison-Wesley; 1995; ISBN 0-191-82458-2), Chris Date defined another operator (DIVIDEBY … PER) which produces the same results as my query, but with more complexity (Figure 34.1).

34.2 Exact Division

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

```
SELECT PS1.pilot_name
  FROM Pilot_Skills AS PS1
       LEFT OUTER JOIN
       Hangar AS H1
```

```
      ON PS1.plane_name = H1.plane_name
 GROUP BY PS1.pilot_name
HAVING COUNT(PS1.plane_name) = (SELECT COUNT(plane_name) FROM Hangar)
   AND COUNT(H1.plane_name) = (SELECT COUNT(plane_name) FROM Hangar);
```

This says that a pilot must have the same number of certificates as there are planes in the hangar and these certificates all match to a plane in the hangar, not something else. The "something else" is shown by a created NULL from the LEFT OUTER JOIN.

Please do not make the mistake of trying to reduce the HAVING clause with a little algebra to:

```
HAVING COUNT(PS1.plane_name) = COUNT(H1.plane_name)
```

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

34.3 Note on Performance

The nested EXISTS() predicates version of relational division was made popular by Chris Date's textbooks, while the author is associated with popularizing the COUNT(*) version of relational division. The Winter 1996 edition of DB2 ON-LINE MAGAZINE (http://www.db2mag.com/96011ar:htm) 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 as well. The COUNT method completed in well under one second, where as 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.

34.4 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_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'

sup_nbr	part_nbr
'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, SP1.supplier
  FROM JobParts AS JP1, SupParts AS SP1
 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 = SP1.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), (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);
```

34.5 Division with JOINs

Standard SQL has several JOIN operators that can be used to perform a relational division. To find the pilots, who can fly the same planes as Higgins, use this query:

```
SELECT SP1.pilot_name
  FROM (((SELECT plane_name FROM Hangar) AS H1
         INNER JOIN
         (SELECT pilot_name, plane_name FROM Pilot_Skills) AS SP1
         ON H1.plane_name = SP1.plane_name)
       INNER JOIN (SELECT *
                     FROM Pilot_Skills
                    WHERE pilot_name = 'Higgins') AS H2
       ON H2.plane_name = H1.plane_name)
 GROUP BY Pilot
HAVING COUNT(*) >= (SELECT COUNT(*)
                      FROM Pilot_Skills
                     WHERE pilot_name = 'Higgins');
```

The first JOIN finds all of the planes in the hangar for which we have a pilot_name. The next JOIN takes that set and finds which of those match up with (SELECT * FROM Pilot_Skills WHERE pilot_name = 'Higgins') skills. The GROUP BY clause will then see that the intersection we have formed with the joins has at least as many elements as Higgins has planes. The GROUP BY also means that the SELECT DISTINCT can be replaced with a simple SELECT. If the theta operator in the GROUP BY clause is changed from >=to =, the query finds an exact division. If the theta operator in the GROUP BY clause is changed from >=to <= or <, the query finds those pilots whose skills are a superset or a strict superset of the planes that Higgins flies.

It might be a good idea to put the divisor into a VIEW for readability in this query and as a clue to the optimizer to calculate it once. Some products will execute this form of the division query faster than the nested subquery version, because they will use the PRIMARY KEY information to pre-compute the joins between tables.

34.6 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 pilots-and-hangar-tables example, we would write

```
SELECT DISTINCT pilot_name
  FROM Pilot_Skills AS P1
 WHERE (SELECT plane_name FROM Hangar
        EXCEPT
        SELECT plane_name
          FROM Pilot_Skills AS P2
         WHERE P1.pilot_name = P2.pilot_name) IS NULL;
```

Again, informally, you can imagine that we got a skill list from each pilot_name, walked over to the hangar, and crossed off each plane_name he could fly. If we marked off all the planes in the hangar, 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.

34.7 Romley's Division

This somewhat complicated relational division is due to Richard Romley, a DBA retired from Salomon 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 each department whether is he 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 cover the department or not.

That observation leads to the next changes. 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 power
  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;
```

Another version, using the airplane hangar example:

```
SELECT PS1.pilot_name,
       CASE WHEN COUNT(PS1.plane_name)
                 > (SELECT COUNT(plane_name) FROM Hanger)
                 AND COUNT(H1.plane_name)
                     = (SELECT COUNT(plane_name) FROM Hanger)
            THEN 'more than all'
            WHEN COUNT(PS1.plane_name)
                 = (SELECT COUNT(plane_name) FROM Hanger)
                 AND COUNT(H1.plane_name)
                     = (SELECT COUNT(plane_name) FROM Hanger)
            THEN 'exactly all '
            WHEN MIN(H1.plane_name) IS NULL
            THEN 'none '
            ELSE 'some ' END AS skill_level
```

```
FROM Pilot_Skills AS PS1
     LEFT OUTER JOIN
     Hanger AS H1
     ON PS1.plane_name = H1.plane_name
GROUP BY PS1.pilot_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	power
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

34.8 Boolean Expressions in Relational Division

Given the usual "hangar and pilots" schema, we want to create and store queries that involve Boolean expressions such as "Find the pilots who can fly a Piper Cub and also an F-14 or F-17 Fighter". The trick is to put the expression into the disjunctive canonical form. In English that means a bunch of AND-ed predicates that are then OR-ed together, like this. Any Boolean function can be expressed this way. This form is canonical

when each Boolean variable appears exactly once in each term. When all variables are not required to appear in every term, the form is called a disjunctive normal form. The algorithm to convert any Boolean expression into disjunctive canonical form is a bit complicated, but can be found in a good book on circuit design. Our simple example would convert to this predicate.

('Piper Cub' AND 'F-14 Fighter') OR ('Piper Cub' AND 'F-17 Fighter') which we load into this table:

```
CREATE TABLE Boolean_Expressions
(and_grp INTEGER NOT NULL,
 skill CHAR(10) NOT NULL,
 PRIMARY KEY (and_grp, skill));

INSERT INTO BooleanExpressions
VALUES (1, 'Piper Cub'), (1, 'F-14 Fighter'),
       (2, 'Piper Cub'), (2, 'F-17 Fighter');
```

Assume we have a table of job_nbr candidates:

```
CREATE TABLE Candidates
(candidate_name CHAR(15) NOT NULL,
 skill CHAR(10) NOT NULL,
 PRIMARY KEY (candidate_name, skill));

INSERT INTO Candidates
VALUES ('John', 'Piper Cub'), --winner
       ('John', 'B-52 Bomber'),
       ('Mary', 'Piper Cub'), --winner
       ('Mary', 'F-17 Fighter'),
       ('Larry', 'F-14 Fighter'), --winner
       ('Larry', 'F-17 Fighter'),
       ('Moe', 'F-14 Fighter'), --winner
       ('Moe', 'F-17 Fighter'),
       ('Moe', 'Piper Cub'),
       ('Celko', 'Piper Cub'), -- loser
       ('Celko', 'Blimp'),
       ('Smith', 'Kite'), -- loser
       ('Smith', 'Blimp');
```

The query is simple now:

```
SELECT DISTINCT C1.candidate_name
  FROM Candidates AS C1, BooleanExpressions AS Q1
 WHERE C1.skill = Q1.skill
 GROUP BY Q1.and_grp, C1.candidate_name
HAVING COUNT(C1.skill)
       = (SELECT COUNT(*)
            FROM BooleanExpressions AS Q2
           WHERE Q1.and_grp = Q2.and_grp);
```

You can retain the COUNT() information to rank candidates. For example, Moe meets both qualifications, while other candidates meet only one of the two. This means you can do very complex candidate selections in pure SQL, but it does not mean you should do it.