**Matrix Math in SQL**

Relational Databases have tables as data structures, not arrays. This makes it tricky and slow to do matrix operations, but it doesn't mean it is impossible to do. Joe gives the Celko Slant on how to go about doing Matrix Math in SQL.

I've just finished teaching a math class at a local school at night, and we had a section on matrix math and determinants. A matrix is not quite the same thing as an array. Matrices are mathematical structures with particular properties that we cannot take the time to discuss here. You can find that information in a college freshman algebra book. This is similar to subtle differences between “hierarchies and trees”.

Whereas an array is merely a data structure who elements are accessed by a numeric value called an index, a matrix is an array with mathematical operations defined on it. A matrix can be one, two, three or more dimensional structures. The most common mathematical convention has be to use the letters i, j and k for the subscripts.

SQL had neither arrays or matrices because the only data structure is a table. Just as there is no obvious way to model trees and hierarchies in SQL, we need ways to model multi-dimensional arrays in SQL.

**Arrays via Named Columns**

An array in procedural programming languages has a name, and subscripts by which the array elements are referenced. The array elements are all of the same data type and the subscripts are all sequential integers. Some languages start subscripts at zero, some start at one, and some let the user set the upper and lower bounds of each subscript. Algol 60 even allowed arrays to be declared with dynamic subscripts using expressions in the declarations when program control enters a block; the array is deallocated on exciting the block. Most languages are not that complex. For example, a Pascal array declaration would look like this:

foobar : ARRAY [1..5] OF REAL;

Array foobar has five elements foobar[1], foobar[2], foobar[3], foobar[4], and foobar[5]. This is most often mapped into an SQL declaration as

CREATE TABLE Foobar1

(element1 REAL NOT NULL,

 element2 REAL NOT NULL,

 element3 REAL NOT NULL,

 element4 REAL NOT NULL,

 element5 REAL NOT NULL));

The elements cannot be accessed by the use of a subscript in this table as they can in a true array. That is, to set the array elements equal to zero in Pascal takes one statement with a FOR loop in it:

FOR i := 1 TO 5 DO foobar[i] := 0;

The same action in SQL would be performed with the statement ....

UPDATE Foobar1

SET element1 = 0,

element2 = 0,

element3 = 0,

element4 = 0,

element5 = 0;

... because there is no subscript which can be iterated in a loop. Any access has to be based on column names and not on subscripts. These “pseudo-subscripts" lead to building column names on the fly in dynamic SQL, giving code that is both slow and dangerous. In short this is the wrong way to do this. Try again.

**Arrays via Subscript Columns**

Another approach of faking a multi-dimensional array is to map arrays into a table with an integer column for each subscript, thus:

CREATE TABLE Foobar2

(i INTEGER NOT NULL PRIMARY KEY

   CONSTRAINT valid\_array\_index

   CHECK(i BETWEEN 1 AND 5),

 element\_value REAL NOT NULL);

This looks more complex than the first approach, but it is closer to what the original Pascal declaration was doing behind the scenes. Subscripts resolve to unique physical addresses, so it is not possible to have two values for foobar[i]; hence, i is a key. The Pascal compiler will check to see that the subscripts are within the declared range; hence the CHECK() clause.

The first advantage of this approach is that multidimensional arrays are easily handled by adding another column for each subscript. The Pascal declaration ...

ThreeD : ARRAY [1..3, 1..4, 1..5] OF REAL;

... is mapped over to ...

CREATE TABLE ThreeD

(element\_value REAL DEFAULT 0.00 NOT NULL,

 i INTEGER NOT NULL

  CONSTRAINT valid\_i

  CHECK(i BETWEEN 1 AND 3),

j INTEGER NOT NULL

 CONSTRAINT valid\_j

 CHECK(j BETWEEN 1 AND 4),

k INTEGER NOT NULL

 CONSTRAINT valid\_k

 CHECK(k BETWEEN 1 AND 5),

PRIMARY KEY (i, j, k));

If the original 'one element\_value/one column' approach were used, the table declaration would have 120 columns, named "element\_value\_111" through "element\_value\_345". This would be too many names to handle in any reasonable way.

But this is still not right. The first problem is that this allows holes in the array. We might actually want this when we have a lot of missing values and want to save storage. But most of the time, we need to have the all the cells of the array modeled in the table.

INSERT INTO ThreeD (i,j,k)

SELECT I.seq, J.seq. K.seq

  FROM Series AS I, Series AS J, Series AS K

 WHERE I.i BETWEEN 1 AND 3

AND J.j BETWEEN 1 AND 4

AND K.k BETWEEN 1 AND 5;

I have assumed you know the idiom ...

CREATE TABLE Series (seq INTEGER NOT NULL PRIMARY KEY);

... which holds sequence of integer from 1 to some value (n). You will see this same thing called “Tally”, “Numbers” or other names.

**Matrix Operations in SQL**

Though it is possible to do many matrix operations in SQL, it is not a good idea, because such queries and operations will eat up resources and run much too long. SQL was never meant to be a language for calculations. Having said that, here are some tricks for you to do.

The presence of NULLs is not defined in linear algebra and I have no desire to invent a three-valued linear algebra of my own. Another problem is that a matrix has rows and columns that are not the same as the rows and columns of an SQL table; as you read the rest of this article, be careful not to confuse the two. Let's try some basic operations.

If you want to have flashbacks to college math, then look at the matrix calculator at <http://www.bluebit.gr/matrix-calculator/>

**Matrix Equality**

This test for matrix equality is from the article "SQL Matrix Processing" by Mrdalj, Vujovic, and Jovanovic in 1996 July issue of DATABASE PROGRAMING & DESIGN. Two matrices are equal if their cardinalities and the cardinality of the their intersection are all equal.

SELECT COUNT(\*) FROM MatrixA

UNION

SELECT COUNT(\*) FROM MatrixB

UNION

SELECT COUNT(\*)

  FROM MatrixA AS A, MatrixB AS B

 WHERE A.i = B.i

AND A.j = B.j

AND A.element\_value = B.element\_value;

You have to decide how to use this query in your context. If it returns one number, they are the same and otherwise they are different.

This was written before we had the rest of the basic set operations:

NOT EXISTS

((SELECT \* FROM MatrixA

  INTERSECT

  SELECT \* FROM MatrixB)

 EXCEPT

 (SELECT \* FROM MatrixA

  UNION

  SELECT \* FROM MatrixB))

There are other ways to write this.

**Matrix Addition**

Matrix addition and subtraction are possible only between matrices of the same dimensions. The obvious way to do the addition is simply:

SELECT A.i, A.j,

(A.element\_value + B.element\_value) AS element\_tot

  FROM MatrixA AS A, MatrixB AS B

 WHERE A.i = B.i

AND A.j = B.j;

But properly, you ought to add some checking to be sure the matrices match.

SELECT A.i, A.j, (A.element\_value + B.element\_value) AS total

  FROM MatrixA AS A, MatrixB AS B

 WHERE A.i = B.i

AND A.j = B.j

AND (SELECT COUNT(\*) FROM MatrixA)

= (SELECT COUNT(\*) FROM MatrixB)

AND (SELECT MAX(i) FROM MatrixA)

= (SELECT MAX(i) FROM MatrixB)

AND (SELECT MAX(j) FROM MatrixA)

= (SELECT MAX(j) FROM MatrixB)

AND (SELECT MIN(i) FROM MatrixA)

= (SELECT MIN(i) FROM MatrixB)

AND (SELECT MIN(j) FROM MatrixA)

= (SELECT MIN(j) FROM MatrixB));

All of the checking for subscripts was done for you under the covers in procedural languages. But SQL has to do it explicitly. Ugh!

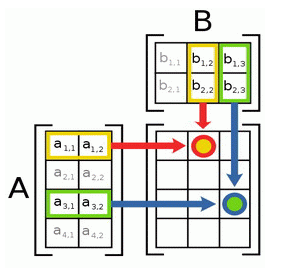
**Matrix Multiplication**

Multiplication by a scalar constant is direct and easy:

UPDATE MyMatrix

SET element\_value = element\_value \* @in\_multiplier;

Matrix multiplication is not as big a mess as might be expected. The first matrix must have the same number of rows as the second matrix has columns. Matrix multiplication is defined as A[i, k] \* B[k, j] = C[i, j]. This illustration from Wikipedia explains it in an intuitive way.



Notice that even with square matrices, A\*B and B\*A might not be the same. Let's look at an example of matrix multiplication

CREATE TABLE MatrixA

(i INTEGER NOT NULL

CHECK (i BETWEEN 1 AND 10), -- pick your own bounds

k INTEGER NOT NULL

CHECK (k BETWEEN 1 AND 10), -- must match MatrixB.k range

element\_value INTEGER NOT NULL,

PRIMARY KEY (i, k));

MatrixA

i k element\_value

=============

1 1 2

1 2 -3

1 3 4

2 1 -1

2 2 0

2 3 2

CREATE TABLE MatrixB

(k INTEGER NOT NULL

CHECK (k BETWEEN 1 AND 10), -- must match MatrixA.k range

j INTEGER NOT NULL

CHECK (j BETWEEN 1 AND 4), -- pick your own bounds

element\_value INTEGER NOT NULL,

PRIMARY KEY (k, j));

MatrixB

k j element\_value

=============

1 1  -1

1 2 2

1 3 3

2 1 0

2 2 1

2 3 7

3 1 1

3 2 1

3 3  -2

CREATE VIEW MatrixC(i, j, element\_value)

AS

SELECT i, j, SUM(MatrixA.element\_value \* MatrixB.element\_value)

  FROM MatrixA, MatrixB

 WHERE MatrixA.k = MatrixB.k

 GROUP BY i, j;

This will give us:

-1  2  3

 0  1  7

 1  1 -2

**Matrix Transpose**

The transpose of a matrix swaps the rows and columns. It is easy to do:

CREATE VIEW TransA (i, j, element\_value)

AS SELECT j, i, element\_value FROM MatrixA;

Multiplication by a column or row vector is just a special case of matrix multiplication, but a bit easier. Given the vector V and MatrixA:

SELECT i, SUM(A.element\_value \* V.element\_value)

 FROM MatrixA AS A, VectorV AS V

 WHERE V.j = A.i

 GROUP BY A.i;

The mathematical notation is a superscript T after the matrix name. The only interesting rules are that (AB)T = BTAT and (AT)T = A. But the multiplication commutes: (AB)C = A(BC).

**Square Matrices**

Matrices with the same number of rows as they have columns have some nice properties, so we seek them out. The n by n Identity matrix has the property that A\*I = I\*A = A. The identity matrix has ones in its main diagonal and zeroes everywhere else.

CREATE TABLE Identity\_Matrix

(element\_value INTEGER DEFAULT 0 NOT NULL

  CHECK (element\_value IN (0,1)),

 i INTEGER NOT NULL

  CHECK (i BETWEEN 1 AND 10), -- pick your own bound

 j INTEGER NOT NULL

  CHECK (j BETWEEN 1 AND 10),

 CONSTRAINT Main\_Diagonal

  CHECK (CASE WHEN i = j AND element\_value = 1

THEN 'T' ELSE 'F' END = 'T')

);

INSERT INTO Identity\_Matrix (element\_value, i, j)

SELECT CASE WHEN I.seq = J.seq THEN 1 ELSE 0 END

  I.seq, J.seq

  FROM Series AS I, Series AS j

 WHERE I.seq <= 10

AND J.seq <= 10;

You could put this in a VIEW, but the advantage of a materialized base table is that the Main\_Diagonal constraint is available for the optimizer. Multiplication by one and zero is really easy to optimize.

The inverse of a square matrix, shown by a superscript -1, has the property that AA-1 = A-1A = I, but A-1 might not exist at all.

**Summary**

It is possible to do fancy matrix operations in SQL, but the code becomes so complex, and the execution time so long, that it is simply not worth the effort. However, multiplication and addition can be very handy, specially if the original data is kept in a matrix.

If a reader would like to submit queries for eigenvalues, inverses and determinants, I will be happy to put them in future editions of one of my books. But I do not want to use it in my real work.