DBMS, June 1996

DBMS online

SQL for Smarties

Joe Celko

When Good Data Goes Bad

Lying grade school teachers, proof that the census is wrong, and the fourth part of a three-part series.

I like to have a theme for my column when I can. This month it will be bad data. Nobody is in favor of bad data; they are just opposed to making the effort required to get good data. That sounds a little silly at first, but consider a few examples.

Bad Programming = Bad Data

You might remember being told in grade school that there are 365.25 days per year and that the accumulation of the fractional day creates a leap year every four years. Once more, your teachers lied to you; there are really 365.2422 days per year and every 400 years the fraction of the fractional day that was left over from the leap years accumulates, too. Because most of us are not over 400 years old, we have not had to worry about this until now. The correct test for leap years in Pascal is:

```
FUNCTION isleapyear (year: INTEGER): BOOLEAN;

BEGIN
```

```
IF ((year MOD 400) = 0)

THEN isleapyear := TRUE

ELSE IF ((year MOD 100) = 0)

THEN isleapyear := FALSE

ELSE IF ((year MOD 4) = 0)

THEN isleapyear := TRUE

ELSE isleapyear := FALSE;

END;
```

A lot of programs were written by people who did not know this algorithm. I do not mean just Cobol legacy programs; I mean packaged programs for which you paid good money. MS Excel for Windows, version 4 shows correctly that the next day after 2000-02-28 is February 29. However, it thinks that the next day after 1900-02-28 is also February 29 instead of March 1. MS Excel for Macintosh doesn't handle the years 1900-1903.

With networked systems, this is already a serious problem. All you need is one program on one node in the network to mess up on one day in the year and the whole network is useless for that day. The Wacovia Bank in Atlanta messed up its automatic payroll deposits on 1996 February 29 because one of its programs could not handle the leap year correctly. Want to guess how many people in the Southeast get paid on the last day of each month via automatic payroll deposit?

I am not going into the "millennium" or "Year 2000" problem in this column, but you might want to read Judith Hurwitz's "Year 2000: Crisis or Opportunity?" on page 10 in this issue.

Bad Data Collection

In late 1991, Professor James P. Allen of California State University at Northridge and Professor David Heer of USC both independently found that the 1990 census for Los Angeles was wrong. The census showed a rise in Afro-American Hispanics in South Central Los Angeles from 17,000 in 1980 to almost 60,000 in 1990. Professor Allen attempted to confirm this growth with field interviews, but he could not find any Afro-American Hispanic children in the schools when he went to the bilingual coordinator for the district's schools.

Professor Heer used the Census Bureau's database to show that the census was wrong. The census questionnaire asked for ethnicity as White, Afro-American, or Asian, but not Hispanic. Most Latinos would not answer this question, considering themselves to be a separate ethnic group. Professor Heer found that the Census Bureau program assigned ethnic group codes when it was faced with missing data. The algorithm was to look at the neighbors and assume that the missing ethnicity data was the same.

The Afro-American population in Los Angeles has actually dropped ("Has Los Angeles Lost Its Census," Buzz magazine, May 1994, by Jill Stewart) as Afro-Americans move up the economic ladder and also move to the suburbs. If the Census Bureau had used SQL, it could have used nulls for the missing data and it might have been saved from embarrassment.

Trees -- Part IV

Yes, I know that the piece on trees was supposed to be a three-part article. I lied. Well, actually, there are a few things I need to

say about optimizing the nested set model of trees. Some of you have already discovered these tricks. Ted Holt of Midrange Computing and several others pointed out that the where clause in queries such as:

```
SELECT DISTINCT B1.emp, (rgt - lft) AS height

FROM Personnel AS B1, Personnel AS E1

WHERE E1.lft BETWEEN B1.lft AND B1.rgt -- required

AND E1.rgt BETWEEN B1.lft AND B1.rgt; -- redundant
```

does not need the second between predicate on the rgt columns of an employee. If their lft value is between the values of the lft and rgt columns of their bosses further up in the hierarchy, then the rgt value also has to be contained within the boss. Be careful; this trick will not work if you drop the first between predicate, which is based on the rgt value.

This is a little hard to see the first time, so you might want to draw a picture and convince yourself. Because both comparisons are being performed on the same rows, there is little or no performance hit, but the extra predicate is redundant.

The constraints that I have shown on the tables are easy to implement in almost all SQL-92 products, but they are not complete. For example, you have nothing that prevents two sets from overlapping instead of nesting. A simple query to locate such problems is:

```
SELECT P1.emp, ' overlaps ', P2.emp

FROM Personnel AS P1, Personnel AS P2

WHERE P2.lft BETWEEN P1.lft AND P1.rgt

AND P1.rgt BETWEEN P2.lft AND P2.rgt

AND P1.emp < P2.emp;
```

In a full implementation of SQL-92, this could be converted into the subquery of a not exists( ) predicate in a check( ) clause on the table.

Another handy validation trick is a view of the numbers currently being used in the lft and rgt columns of an Organization table:

```
CREATE VIEW OrgLftRgt (num)

AS SELECT lft FROM Organization

UNION

SELECT rgt FROM Organization;
```

Some SQL products will still not allow you to use a union in a view, so this might not work for you. Following are some queries that you can use to check the status of the tree:

```
SELECT 'Tree has duplicate node numbers = ', num
```

```
FROM OrgLftRgt

GROUP BY num

HAVING COUNT(*) > 1;

SELECT 'Tree has gaps in node numbers'

FROM OrgLftRgt

GROUP BY num

HAVING COUNT(*) <> MAX(num);
```

Another tip is to put the tree structure information in one table and the node information in a second table. The tree table will be quite small and you can change the structure or the nodes independently of one another. For example, in the examples in this series I used a simplified Personnel table that looked like the following:

```
CREATE TABLE Personnel

(emp_id CHAR(10) PRIMARY KEY,

salary DECIMAL(8, 2) NOT NULL CHECK (salary >= 0.00),

lft INTEGER NOT NULL UNIQUE CHECK (lft > 0),

rgt INTEGER NOT NULL UNIQUE CHECK (rgt > 0),

CHECK (lft < rgt));
```

What you would have had in a real schema, which would have far more data items, is several tables; one for the tree structure itself and one for each of the entities involved in a node (in this example, that would be employees and the job positions):

```
CREATE TABLE Organization

(position CHAR(10) NOT NULL PRIMARY KEY,

emp_id CHAR(10) NOT NULL REFERENCES Personnel (emp_id),

lft INTEGER NOT NULL UNIQUE CHECK (lft > 0),

rgt INTEGER NOT NULL UNIQUE CHECK (rgt > 1),

CHECK (lft < rgt));

CREATE TABLE Personnel

(emp_id CHAR(10) NOT NULL PRIMARY KEY,

emp_name CHAR(30) NOT NULL,
```

```
emp_address CHAR(30) NOT NULL,

salary DECIMAL(8, 2) NOT NULL CHECK (salary >= 0.00),

...

);

CREATE TABLE Positions

(position CHAR(10) NOT NULL PRIMARY KEY,

job_title CHAR(20) NOT NULL,

high_salary DECIMAL(10, 2) NOT NULL,

low_salary DECIMAL(10, 2) NOT NULL,

...

);
```

You then join the tables to see all of the details of who holds which position. Notice that the way we have this set up, the same person can hold multiple positions within an organization. If you wish to disallow this, put a unique constraint on the emp_id column in the Organization table. Generally speaking, a organizational chart disallows multiple roles and a parts explosion has lots of them.

As an aside, the adjacency matrix model has problems in separating the hierarchical structure and the node data. For example, consider the sample table:

Personnel

```
emp        boss     salary

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

'Jerry'    NULL    1000.00

'Bert'    'Jerry'   900.00

'Chuck'   'Jerry'   900.00

'Donna'   'Chuck'   800.00
```

```
'Eddie'   'Chuck'   700.00

'Fred '   'Chuck'   600.00
```

Employee Chuck decides that he will now be called Charles instead, in fitting with his new promotion. This means that you must change a primary key in the table, which does happen in the real world. Oops, we also have to change Chuck to Charles in the boss column, so that Donna, Eddie, and Fred answer to the correct boss. This means that the table declaration should be:

CREATE TABLE Personnel

(emp CHAR(20) PRIMARY KEY REFERENCES Personnel(boss)

ON UPDATE CASCADE

ON DELETE CASCADE,

boss CHAR(20) REFERENCES Personnel(emp)

ON UPDATE CASCADE

ON DELETE CASCADE,

salary DECIMAL(6,2) NOT NULL);

Let's say you decide to fire Charles, but first you need to reassign Donna, Eddie, and Fred to a new boss or terminate them and their subordinates because the referential constraint will not let us just drop Charles from the table.

Puzzle

Sissy Kubu sent me a strange question on CompuServe. She has a table like this:

CREATE TABLE Inventory

(goods CHAR(10) NOT NULL PRIMARY KEY,

pieces INTEGER NOT NULL CHECK (pieces >= 0)

);

She wants to deconsolidate the table; that is, get a view or table with one row for each pieces. For example, given a row with ('cd-rom', 3) in the original table, she would like to get three rows with ('cd-rom', 1) in them. Before you ask me, I have no idea why she wants to do this; consider it a training exercise.

Because SQL has no "un-count(*) ... de-group by.." operators, you will have to use a cursor or the vendor's 4GL to do this. Frankly, I would do this in a report program instead of a SQL query because the results will not be a table with a key.

The obvious procedural way to do this would be to write a routine in your SQL's 4GL that reads a row from the Inventory table, and then writes the value of good to a second table in a loop driven by the value of pieces. This will be pretty slow, because it

requires (select sum(pieces) from Inventory) single-row insertions into the working table. Can you do better? (See Puzzle Answer on page 22.)

Puzzle Answer

I always stress the need to think in terms of sets in SQL. The way to build a better solution is to perform repeated self-insertion operations, using a technique based on the "Russian peasant's algorithm," which was used for multiplication and division in early computers. You can look it up in a history of mathematics or a computer science book -- it is based on binary arithmetic and can be implemented with right and left shift operators in assembly languages.

You will still need a 4GL to do this, but it won't be so bad. First, create two working tables and one for the final answer:

```
CREATE TABLE WorkingTable1

(goods CHAR(10) NOT NULL,

pieces INTEGER NOT NULL);

CREATE TABLE WorkingTable2

(goods CHAR(10) NOT NULL,

pieces INTEGER NOT NULL);

CREATE TABLE Answer

(goods CHAR(10) NOT NULL,

pieces INTEGER NOT NULL);
```

Now start by loading the goods that have only one piece in inventory into the answer table:

```
INSERT INTO Answer

SELECT * FROM Inventory WHERE pieces = 1;
```

Now put the rest of the data into the first working table:

```
INSERT INTO WorkingTable1

SELECT * FROM Inventory WHERE pieces > 1;
```

The following block of code will load the second working table with pairs of rows that each have half (or half plus one) the piece counts of those in the first working table:

```
INSERT INTO WorkingTable2

SELECT goods, FLOOR(pieces/2.0)
```

```
FROM WorkingTable1

WHERE pieces > 1

UNION ALL

SELECT goods, CEILING(pieces/2.0)

FROM WorkingTable1

WHERE pieces > 1;
```

The floor(x) and ceiling(x) functions return, respectively, the greatest integer that is lower than x and smallest integer that is higher than x. If your SQL does not have them, you can write them with rounding and truncation functions. It is also important to divide by (2.0) and not by 2 because this will make the result into a decimal number.

Now harvest the rows that have gotten down to a piece count of one and clear out the first working table:

```
INSERT INTO Answer

SELECT *

FROM WorkingTable2

WHERE pieces = 1;

DELETE FROM WorkingTable1;
```

Exchange the roles of WorkingTable1 and WorkingTable2, and repeat the process until both working tables are empty. That is simple, straightforward procedural coding. The ways that the results shift from table to table are interesting to follow. Think of these diagrams as an animated cartoon:

Step One: Load the first working table, harvesting any goods that already had a piece count of one.

```
WorkingTable1              WorkingTable2

goods      pieces          goods      pieces

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

Alpha        4

Beta         5

Delta       16
```

```
Gamma       50
```

The row (Epsilon, 1) goes immediately to Answer table.

Step Two: Halve the piece counts and double the rows in the second working table. Empty the first working table.

```
WorkingTable1               WorkingTable2

goods      pieces           goods      pieces

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

                            Alpha        2

                            Alpha        2

                            Beta         2

                            Beta         3

                            Delta        8

                            Delta        8

                            Gamma       25

                            Gamma       25
```

Step Three: Repeat the process until both working tables are empty.

```
WorkingTable1               WorkingTable2

goods      pieces           goods      pieces

========== =======          ========== =======
```

```
Alpha        1

Alpha        1

Alpha        1

Alpha        1

Beta         1  Alpha and Beta are ready to harvest

Beta         1

Beta         1

Beta         1

Beta         1

 --------------

Delta        4

Delta        4

Delta        4

Delta        4

Gamma       12

Gamma       12

Gamma       13
```

```
Gamma       13
```

The cost of completely emptying a table is usually very low. Likewise, the cost of copying sets of rows (which are in physical blocks of disk storage that can be moved as whole buffers) from one table to another is much lower than inserting one row at a time.

The code could have been written to leave the results in one of the working tables, but instead this approach allows the working tables to get smaller and smaller so that you get better buffer usage. This algorithm uses (select sum(pieces) from Inventory) rows of storage and (log2((select max(pieces) from Inventory)) + 1) moves, which is pretty good on both counts. -- Joe Celko

---

Joe Celko is a member of the ANSI X3H2 Database Standards Committee, a widely published author, and a consultant with OSoft Development Corp. in Atlanta. Joe is also a frequent contributor to the DBMS Forum on CompuServe. You can email him at 71062.1056@compuserve.com.

Table of Contents - June 1996 | Home Page

Please send questions or comments to mfrank@mfi.com
Updated Saturday, June 22, 1996