Paper 084-2008

# Existential Moments in Database Programming: SAS® PROC SQL EXISTS and NOT EXISTS Quantifiers, and More

Sigurd W. Hermansen and Stanley E. Legum, Westat, Rockville, MD, USA

## ABSTRACT

Existential and universal quantifiers have a key role in SAS/Base Version 9.1.3 PROC SQL programs--especially UPDATE's—yet, tend to baffle even experienced database programmers. The IN, EXISTS, and ALL operators put complex table look-up operations in the basic framework of WHERE clause logical conditions. Difficult problems for SAS Data step programmers, such as searches for data patterns in related tables or in different rows within the same table, become easier for those who understand these operators. (Even a caveman can learn to use of these logical quantifiers.) Carefully chosen examples illustrate both the need for existential quantifiers in database programming and methods for implementing them in SAS programs.

## INTRODUCTION

All database programming systems feature automatic looping through sets of records. The SAS System offers two complementary methods: looping through a physical sequence of observations in a Data step; and, SAS SQL scanning of each row (tuple) in an abstract table (relation). The former generalizes record processing in system files to INPUT and SET statements. The latter gives the SAS SQL compiler a great deal of latitude in deciding how to go about processing each row in a table.

```
Visits DATASET              Data Step                  SQL
ID type date  measureX      DATA D;                    CREATE TABLE D AS
03  B   0705   15             SET Visits;              SELECT id, date,  measureX,
02  A   0702   30               IF measureX > 20                 CASE WHEN value > 20
05  B   0707   20                   THEN type = 'C'                            THEN 'C' ELSE
type
02  A   0708   21                                         END AS type FROM Visits
04  A   0707   19               IF date > 0703;        WHERE date > 0703;
```

Looking initially at automated looping alone, A SAS Data step evaluates logical conditions, such as a subsetting IF, or IF … THEN … ELSE, or CASE statement, for each row separately (save only for RETAIN'd or lagged values carried over from rows accessed earlier). This leads many SAS programmers early in their careers to believe that it works best to stuff many variables related to a single entity, say a person, into one row. Statisticians and analysts tend to encourage that way of thinking. Only when the complexity of relations among data elements reaches a critical point, when management of variable names and data types becomes overwhelming or a relational database architect normalizes a crucial data source, does a flat file programmer feel compelled to think outside the row. In the case of dataset D (above), a programming task that requires information about ID 02 in row 5 when an automated loop reaches row 2 creates an existential crisis. For example, does the earliest date for an ID correspond to the largest value for measureX?

Logicians, who developed the basic logic that we use daily in computer programs, actually reached much the same crisis point long ago, when logical sentences (*propositions*) did not have enough power to answer questions about an object that ranged over many sentences; for example, considering each row in Visits as a sentence and the variable date as an object, for all instances of date in Visits, what is the earliest date value for ID ‘02’? No one sentence (row) has the answer to that question. One must “quantify” an answer over all sentences in Visits. Any question that asks “For all instances of date in Visits where ID=’02’, is date=’0702’ the earliest?”, or the equivalent, “Does there exist a date in visits where ID=’02’ and date less than ‘0702’?”, goes beyond *propositional* logic and into the realm of *predicate logic*. A typical SAS Data step solution SORT’s Visits BY ID date in ascending order, then SET’s Visits BY ID. The FIRST.ID automatic variable then has a True value when the automated loop counter reaches the first row containing ID ‘02’. True to the origins of SQL and other set-logic programming languages, SAS Proc SQL Version 9.1.3 offers methods much closer to the usual “For all …” (∀x P(x)) and “There exists …” (∃x P(x)) constructs of predicate logic. The “There exists …” predicate proves essential to a generic Proc SQL Update statement, as we shall demonstrate. The predicate logic quantifiers also expand the horizons of database programs beyond the realm of whatever row happens to be next in physical sequence and the variable values that it happens to contain.

Database programmers familiar with table look-ups, SAS formats, and pointer indexes will immediately see the close analogy between predicate logic and its power to reference sentences across a “universe of discourse”, and advanced programming methods that link data items in different arrays and objects. The SQL implementations of quantifiers in predicate logic give programmers more general and flexible ways to think and act outside the row. Though somewhat obscure and tricky at first glance, the quantifiers quickly become easy to apply and extend, more than can be said of many software engineering alternatives.

## PREDICATE LOGIC IN SAS PROC SQL

Beginning with SAS System Version 9.1.3, Proc SQL now supports universal quantification with the ALL operator as well as existential quantification with the EXISTS operator included in earlier versions. Logical conditions in WHERE clauses may include ALL or EXISTS clauses of the form

- WHERE ….VarX <boolean operator> ALL (SELECT VarX FROM <relation> WHERE <logical condition>); or
- WHERE …. EXISTS (SELECT 1 FROM <relation> WHERE <logical condition>).

The ‘1’ following SELECT represents a minimal column value that the query nested in parentheses yields if and only if its WHERE condition evaluates as true. The EXISTS condition evaluates to True unless the nested query yields an empty relation; that is, no rows. As a primary example, consider the Proc SQL UPDATE.

### PROC SQL UPDATE

In this example we have a number of empty LastName attributes in the database table **Subject** and names to fill in from a temporary table **Names**.

**Subject**

| ID | LastName |
|---|---|
| 1 | |
| 2 | |
| 3 | Applebaum |
| 4 | |
| 5 | |
| 6 | |
| 7 | |
| 8 | |
| 11 | |

**Names**

| ID | LastName |
|---|---|
| 8 | Beauregaard |
| 5 | Perkins |
| 3 | Applebee |
| 1 | Flintstone |
| 4 | Gonzalez-Rubio |
| 2 | Bart |
| 6 | He |
| 7 | Mosby |
| 11 | Zeller |

| ID | LastName |
|---|---|
| 12 | |
| 13 | |
| 0 | |

| ID | LastName |
|---|---|
| 10 | Houston |

Here's what happens if we try to perform this update without using an existential quantifier. A Proc SQL UPDATE program automatically loops through each row of Subject and, provided that ID's match, overwrites NULL values of LastName in Subject with a LastName value:

```
proc sql ;
 UPDATE Subject as t1
 SET LastName=(select LastName from Names as t2
               WHERE t1.LastName IS NULL AND t1.ID=t2.ID
               )
 ;
quit;
```

All appears to have worked as expected until we look closely at what happened to rows in Subject during the UPDATE.

| ID | LastName |
|---|---|
| 1 | Flintstone |
| 2 | Bart |
| 3 | |
| 4 | Gonzalez-Rubio |
| 5 | Perkins |
| 6 | He |
| 7 | Mosby |
| 8 | Beauregaard |
| 11 | Zeller |
| 12 | |
| 13 | |
| 0 | |

The LastName attribute of ID=3 has had its value replaced with a NULL value! What happened?

The SQL SET operator behaves the same as an assignment operator ('=' in a SAS Data step and in other programming languages). The assignments of value proceed row by row through the table. The logical condition in the subquery specifies that the ID in Subject has to equal the ID in Names and the LastName attribute in Subject has to have a NULL value for the select of LastName in Names to take place. When the LastName attribute in the SUBJECT table is not null, the SELECT statement returns a null value and the SET assignment executes, replacing the existing name with null. Therefore the subquery yields a NULL value, and the SET clause dutifully assigns the NULL as the column value, replacing the name 'Applebaum'. The SET clause also replaces NULL values with NULL values for ID's 12, 13, and 0, as indicated by this note in the log produced by Proc SQL:

```
NOTE: 12 rows were updated in WORK.SUBJECT.
```

An existential quantifier, specified with the EXISTS operator, restricts the set of rows in Subject being UPDATE'd to those that have both a NULL value of LastName in Subject and matching ID's:

```
proc sql;
 UPDATE Subject as t1
 SET LastName=(select LastName from Names as t2
               WHERE t1.ID=t2.ID
               )
 where exists (select 1 from Names as t2
               WHERE t1.LastName IS NULL AND t1.ID=t2.ID
               )
 ;
quit;
```

It specifies the UPDATE's correctly:

| ID | LastName |
|---|---|
| 1 | Flintstone |
| 2 | Bart |
| 3 | Applebaum |
| 4 | Gonzalez-Rubio |
| 5 | Perkins |
| 6 | He |
| 7 | Mosby |
| 8 | Beauregaard |
| 11 | Zeller |
| 12 | |
| 13 | |
| 0 | |

The note in the log now indicates that only 10 rows were updated, which was the intent of the query.

Often a SAS Macrovariable containing the same logical condition can make the program less verbose, especially when UPDATE'ing more than one attribute in the same table:

```
%let __cond = %str( t1.LastName IS NULL AND t1.ID=t2.ID);
proc sql ;
 UPDATE Subject as t1
 SET LastName =  (select LastName from Names as t2 where &__cond),
     FirstName = (select FirstName from Names as t2 where &__cond)
     /* etc. */
 WHERE exists (select 1 from Names as t2 where &__cond )
 ;
```

```
quit;
```

Redundancy of conditions in row and dataset logical conditions adds very little to execution time.

What does the 1 in "select 1 …" mean? It may help to think of the 1 as a SAS logical value of "True", but in truth the in-line query yields an empty relation when the logical condition does not hold, and a relation with one attribute and row (cell) value(s) containing 1 when it does. The EXISTS operation merely tests for a non-empty relation in the yield of the in-line query. Note that the choice of 1 as the value to return is arbitrary. Any non-null value would work. (When the in-line query returns an empty relation, it is analogous to C. J. Date's "Table_Dum"),.

### THE UNIVERSAL QUANTIFIER, ALL

Whereas the existential quantifier requires only that at least one instance in its argument meet a logical condition, the universal quantifier ALL requires (as the name suggests) that all instances of its relation argument meet a logical condition. A universal quantifier makes declaration of "outside the row" SQL solutions simple. Say that we are looking for medical study subjects that have survived very high systolic blood pressure. We have vital status (VS) and blood pressure (BP) for patients:

```
data patientInfo;
   input ID VS $char5. BP;
cards;
10 Alive 130
06 Alive 115
07 Dead  120
22 Dead  140
16 Alive 150
20 Dead  125
03 Alive 125
;
run;
```

The inner query in the program below compiles a column vector of BP for deceased subjects. If a patient has a BP value greater than any in the vector, the program selects his or her patientInfo:

```
proc sql;
  select * from patientInfo
  where VS="Alive"
    and BP >ALL (select BP from patientInfo
                 where VS="Dead"
                 )
  ;
quit;
```

In this example, the query returns only the record for ID=16, who is Alive and has a BP greater than any of those listed as dead.

### TO EXIST OR TO NOT EXIST

Often the question comes down to whether something doesn't exist, which, as we all know from basic logic and mathematics, turns out to be more difficult to prove than whether something does exist. One of the more common SQL rookie errors brings down the system when someone changes an equality AND condition to an inequality form,

… SELECT * FROM t1 inner join t2 WHERE t1.ID ^= t2.ID AND t1.Site ^= t2.Site ;

and the machine generates hundreds of millions of rows of data. Why so? The trap lies in the mistaken idea that the WHERE compiler understands when a programmer means to evaluate a logical expression over a "universe of discourse" or a scope limited to one comparison of tuples (rows) at a time. While in the universe of t1 and t2 the complement of "some true" could be "some false", the WHERE clause calls for evaluation individually of each pair of rows of t1 and t2. Since very few of these pairs will have equivalent values, the query selects almost as many pairs of rows as the number of rows (cardinality) of t1 times the number of rows in t2.

### THE LOGICAL IN OPERATOR

Many SAS programmers who know little about universal or existential quantifiers may be making frequent use of a closely related quantifier in predicate logic, the SQL IN operator:

```
  … where x IN (y) …
```

In the analogous data step statement, the programmer replaces "y" with an explicit list of values to compare x against. That will work in a SQL statement too, but SQL offers an additional option. An argument "y" can be any SQL statement that returns a relation with a single column. The value of x is compared to each of the values in the column represented by y just like the comparison that is made in the data step operation. In each case, if the value of x is found in list y, "x in (y)" evaluates to "True".

The IN operator illustrates two crucial concepts in relational database programming. First, all queries, whether top level or nested, resolve to relations. This "closure" property supports the use of queries as values of arguments of operators such as IN. Second, some contexts in SQL programs require a specific type of instance of a relation. The argument of an IN operator, for example, has to be of "relvar type" *attribute* (column).

## CONCLUSION

Thinking outside the row represents a pivotal stage in the evolution of a database programmer. Existential quantifiers in set-logic programming languages such as SQL extend more basic methods based on propositional logic to the realm of predicate logic. In doing so, these languages allow programmers to select data based on whether at least one instance exists (or whether all instances) in a database (relation) meet a condition. Here we have introduced concepts and provided basic examples. Effective use of predicate logic constructs in SAS PROC SQL may require different methods of implementation, but the concepts remain general and crucial for database work.

## REFERENCES

SAS Institute Inc. 2006. *Base SAS® 9.1.3 Procedures Guide, Second Edition, Volumes 1, 2, 3, and 4*. Cary, NC: SAS Institute Inc..

SAS Institute Inc., 2004. *SAS® 9.1 SQL Procedure User's Guide*. Cary, NC: SAS Institute Inc.

## RECOMMENDED READING

Date, C. J. 2005. *Database in Depth: Relational Theory for Practioners*. Sebastopol, CA: O'Reilly Media, Inc.

## DISCLAIMERS

The contents of this paper are the work of the author and do not necessarily represent the opinions, recommendations, or practices of Westat.

## CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Name Sigurd W. Hermansen
Enterprise Westat
Address 1650 Research Blvd
City, State ZIP Rockville, MD 20850
Work Phone: 301.251.4268
E-mail: hermans1@westat.com