

Clinical Trial Data Validation: Using SAS® PROC SQL Effectively

Balakrishna Dandamudi, SFBC New Drug Services

OBJECTIVE:

The main objective of CDM division of a CRO or Pharmaceutical Company is to develop clinical trial databases that are comprehensive and superior in quality to meet the study's objectives and comply with regulatory standards.

Data validation plays a key role in achieving this object. One of the common procedures adopted to validate a clinical trial database is, batch validation. In batch validation, one executes a series of checks that are developed to validate the clinical trial database.

When it comes to programming languages, we always left with more than one option. However, the universally adopted language for querying a database is SQL (Structured Query Language).

This paper explores the PROC SQL, SAS® version of SQL, from clinical data validation perspective. In particular this paper provides a comprehensive and cohesive overview of the PROC SQL features in the form of examples, which can be effectively used in the process of building a best tool for clinical data validation.

INTRODUCTION:

Procedure SQL is the SAS® version of SQL that can be used to define, access, manipulate and query the data. To better understand, Procedure SQL, consider reviewing SQL and its command structure. To keep things simple no relational data base concepts reviewed. This paper assumes some prior SQL and relational data base concepts familiarity. For more information refer to the papers listed in the references section. This paper will cover the following topics:

- SECTION 1: This section glance through the basic concepts of SQL and its features.
- SECTION 2: This section refers to both SQL and SAS® specific features of Procedure SQL.
- SECTION 3: This section refers to Clinical Data Validation using Procedure SQL. The examples presented in this section can be used with little or no modification in any clinical trial database.

SECTION 1: THE 5 Ws OF SQL

What is SQL?

The original version of SQL was called SEQUEL (**Str**uctured **E**nglish **QUE**ry **L**anguage), better known as SQL (and pronounced "**se**quel" or "**ess**-cue-el"). SQL is a set of commands that allow accessing a relational database.

SQL is the standard query language for **Relational Data Base Management Systems** (RDBMSs). SQL is used by most of the leading RDBMS products such as Oracle and SQL Server. Indeed, every relational database management system - and many non-relational DBMS products - supports SQL as the method for accessing data.

SQL has a simple command structure for data definition, access, and manipulation. SQL is set oriented. You can perform a command on a group of data rows or one row. SQL is non-procedural. When you use SQL you specify what you want to be done, not how to do it. To access data you need only to name a table and the columns; you do not have to describe an access method.

Why is it so important?

SQL's features make it the most widely used language for relational databases. The primary advantages of SQL are:

Acceptance - SQL has approval and support of ANSI, ISO and the U.S. Department of Defense. The American National Standards Institute (ANSI) and the International Standards Organization (ISO) define software standards, including standards for the SQL language.

Power - SQL is powerful. SQL command language is simple to use and broad in the scope of functions available with flexibility to do complex operations.

High-level, English like structure/Ease of use - Most SQL statements "say what they mean" and can be read as regular sentences. SQL statements describe the *data* to be retrieved, rather than specifying *how* to find the data. This makes SQL relatively easy to learn and use compared to other programming languages.

Vendor independence/Portability - Many database systems use SQL as the method for retrieving and analyzing data. SQL programs are adoptable from one DBMS to other DBMS with minimal conversion effort and personnel training. The vendor independence is one of the most important reasons for its portability.

Where to find SQL?

You cannot find SQL as stand alone product. Instead, SQL is an integral part of a database management system, a language and a tool for communicating with the DBMS.

Example: PL/SQL of ORACLE, TSQL of MS SQL, and PROC SQL of SAS®.

Who uses SQL?

Database Administrators: The database administrators use SQL to define the database structure and control access to the stored data.

Developers and/or Programmers: Developers and/or Programmers write programs containing SQL commands to connect, access, and manipulate data in a database.

End users: End users issue SQL commands to retrieve, insert, update, or delete data either through an interactive command interface or a client application.

When to use SQL?

SQL has no boundaries in the relational data base world. SQL is more than a query tool. SQL has emerged as a standard tool for managing relational database systems. Standard SQL statements can be subdivided into four distinct groups based on usage, which is shown below. However, in this session the main focus is on SQL's Data Query Language commands.

Data Definition Language (DDL): The Data Definition Language (DDL) permits database tables to be created or deleted. Also define indexes (keys), specify links between tables, and impose constraints between database tables. The common DDL statements are CREATE TABLE, ALTER TABLE, DROP TABLE, CREATE VIEW, CREATE INDEX, and DROP INDEX

Data Manipulation Language (DML): SQL's Data Manipulation Language commands allows to read and manipulate the data by adding new data, removing old data, and modifying previously stored data. Examples: INSERT, DELETE, and UPDATE.

Data Query Language (DQL): SQL's Data Query Language commands allow building complex queries with relational operators. A query can use a join to pull data from different tables and correlate it by matching a common row that is in all the tables. Example: SELECT

Data Control Language (DCL): SQL can be used to restrict a user's ability to retrieve, add, and modify data, protecting stored data against unauthorized access. Example: REVOKE, GRANT

SECTION 2: PROCEDURE SQL

Procedure SQL is the SAS® version of SQL, which is part of base SAS. The terms *table* and *data set* are used interchangeably. A *row* in a table is the same as an *observation* and a *column* is same as a *variable* in a SAS data set.

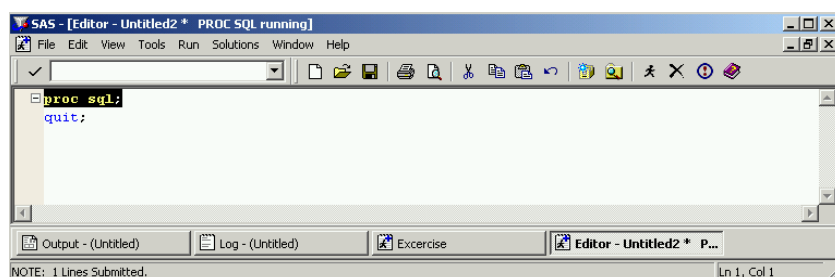


Figure: Invoking Procedure SQL

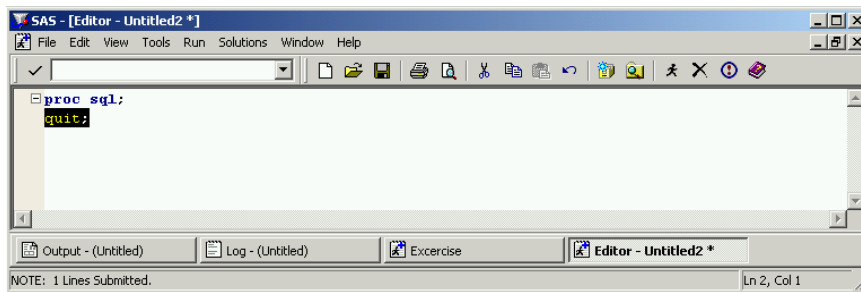


Figure: Stopping Procedure SQL

PROC SQL differs from most other SAS procedures. To invoke SQL procedure submit "PROC SQL;" statement. When you submit the PROC SQL step, the status line in the active Enhanced Editor or Program Editor window displays the message *PROC SQL running*. Unlike most other procedures, the SQL procedure continues to run after you submit a statement.

PROC SQL statements consist of clauses. Items within clauses are separated with commas, not with blanks as in data step and other PROCEDURES in the SAS system. Each statement is executed before the next statement is interpreted. It executes immediately without looking for a RUN statement. RUN statement has no effect in PROC SQL world.

After initial call up you can submit a series of SQL statements without repeating "PROC SQL" statement. However any calls to data step or any other SAS procedure makes PROC SQL engine to stop. The other way of quitting the PROC SQL is by submitting the "QUIT;" statement.

PROC SQL is an efficient one step stand-in to few SAS procedures and most of the data step tasks. PROC SQL supports SAS system options, data set options, global statements, functions, informats, formats and macro variables.

With this knowledge let's advance further to get our feet wet. Let's have a closer look at basic PROC SQL statements.

ALTER TABLE	CREATE TABLE	DELETE	DESCRIBE TABLE	DROP TABLE
INSERT INTO	RESET	SELECT	UPDATE	VALIDATE

Querying Data

The most widely used statement in PROC SQL is, the one we use to query tables, the SELECT statement. We find SELECT statement usage all over, including in other PROC SQL statements. Without knowing SELECT statement, we cannot discuss every available feature of many of the PROC SQL statements like CREATE TABLE, INSERT INTO, ALTER. So let's start with SELECT statement first.

The Simple SELECT Statement

At its simplest, SELECT statement can be used to select data from a single table. When executed, the SELECT statement returns one or more rows, called the result set. If we don't specify a WHERE clause, it simply returns one row for every row in the table.

EXAMPLE 1-1: Using SELECT * ...

The select *, gets all the columns of a table. You cannot list the columns individually, so you cannot give the columns an alias or specify an order for the columns. The columns are listed in the order in which they appear in the table.

```
select * from demog; /* you should interpret asterisk as "all the fields" */
```

Note 1: The above code is missing PROC SQL step and QUIT step. Through out this paper we will use the same convention. You need to add those before running this code.

```
proc sql;
    select * from demog; /* you should interpret asterisk as "all the fields" */
quit;
```

Note 2: The table names are not associated with any qualifier, meaning they are from work library in SAS. You can add qualifier as we do in normal data step and PROC programming. However, one needs to be cautious while working with DDL and DML statements of PROC SQL.

OUTPUT: The Simple SELECT ... Statement output

Protocol ID	Birth Date (C)	Birth Date (N)	Site ID	Subject ID	Subject Initials	Visit ID	Visit Date (C)	Visit Date (N)	Date Informed Consent Signed (C)	Date Informed Consent Signed (N)	Time Informed Consent Signed (C)	Time Informed Consent Signed (N)
			Gender	Race	Other Race	Race - Specify	CRF Page Number					
NESUG-2004-HM06	22JUN1972	22JUN72	001	0001	AAA	2	1	14JUL2003	14JUL03	14JUL2003	14JUL03	15:37
NESUG-2004-HM06	14DEC1969	14DEC69	001	0002	BBB	2	1	17JUL2003	17JUL03	17JUL2003	17JUL03	13:42
NESUG-2004-HM06	01MAY1962	01MAY62	001	0003	CCC	1	1	17JUL2003	17JUL03	17JUL2003	17JUL03	16:15
NESUG-2004-HM06	13OCT1972	13OCT72	001	0004	DDD	1	1	18JUL2003	18JUL03	18JUL2003	18JUL03	13:45
NESUG-2004-HM06	20MAR1962	20MAR62	002	0001	BBB	5	1	29OCT2003	29OCT03	29OCT2003	29OCT03	12:00

The query results are displayed in SAS log as soon as PROC SQL finishes executing the query as shown above. The results appear in the SAS log as they are shown in the above figure. Column labels are at the top with the columns shown in the order in which they were specified in the SELECT statement or by the order in which they were defined when the table was created, if asterisk is used in SELECT statement. If more columns are specified in the SELECT statement than can fit in the log window, they are split up on two or more lines.

EXAMPLE 1-2: SELECT list of columns ...

To select certain columns in a table specify these columns as column list in SELECT statement. You can rename these columns, giving them a *column alias*. The output in SAS log appears in the order they are specified in the SELECT clause.

```
select PROTCLID, SUBJINIT, SITEID, SUBJID, VISDT, VISID, CSDT, BIRTHDT, GENDER, RACE, OTHRACE
from demog;
```

OUTPUT: Simple SELECT [specified column output]

Protocol ID	Subject Initials	Site ID	Subject ID	Visit Date (N)	Visit ID	Date Informed Consent Signed (N)	Birth Date (N)	Gender	Race	Other Race	Race - Specify
NESUG-2004-HM06	AAA	001	0001	14JUL03	1	14JUL03	22JUN72	1	2		
NESUG-2004-HM06	BBB	001	0002	17JUL03	1	17JUL03	14DEC69	2	3		
NESUG-2004-HM06	CCC	001	0003	17JUL03	1	17JUL03	01MAY62	1	1		
NESUG-2004-HM06	DDD	001	0004	18JUL03	1	18JUL03	13OCT72	1	1		
NESUG-2004-HM06	BBB	002	0001	29OCT03	1	29OCT03	20MAR62	2	5		

EXAMPLE 1-3: SELECT ... DISTINCT ...

The SELECT statement with distinct clause eliminates all the duplicate rows from the result table. Two rows are duplicates if they have identical values in every column of the result table. If even one column is different, then they do not match and they are not duplicates. As an example, you need to list distinct subjects in the medical history table.

```
select distinct siteid, subjid from medhx;
```

OUTPUT: SELECT ... DISTINCT

Site ID	Subject ID
001	0001
001	0002
001	0003
001	0004
002	0001

EXAMPLE 1-4: Defining New Columns

You can create a column by giving column alias to the existing column. You can also assign the result of an expression or the literal by using the keyword **AS**. This changes the heading that appears in the result table. It does not have any permanent effect on the actual table.

```
select 'NESUG-2004-HW06' as Protocol,
       siteid, subjid, birthdt, csdt, floor((csdt - birthdt)/365.25)+1 as AGE
from   demog;
```

OUTPUT: Defining New Columns

Protocol	Site ID	Subject ID	Birth Date (N)	Date Informed Consent Signed (N)	AGE
NESUG-2004-HW06	001	0001	22JUN72	14JUL03	32
NESUG-2004-HW06	001	0002	14DEC69	17JUL03	34
NESUG-2004-HW06	001	0003	01MAY62	17JUL03	42
NESUG-2004-HW06	001	0004	13OCT72	18JUL03	31
NESUG-2004-HW06	002	0001	20MAR62	29OCT03	42

The column named protocol is defined by a literal. The column named AGE is a new column that is derived from Birth Date and Consent Signed Date.

SELECT ... FROM ... WHERE

Now let's look at how to specify which rows to retrieve based on search conditions. You can do this by using WHERE Clause of your SELECT statement. Search conditions include comparison operators, ranges, lists, string matching, unknown values, combinations, and negations of these conditions.

EXAMPLE 1-5: Comparison Operators

The common comparison operators are: =, EQ, <>, NE, >, GT, <, LT, <=, LE, >=, GE

```
select siteid, subjid, birthdt, csdt, floor((csdt - birthdt)/365.25)+1 as AGE
from   demog
where  calculated age < 40; /* alternate: where calculated age LT 40;*/
```

OUTPUT:

Site ID	Subject ID	Birth Date (N)	Date Informed Consent Signed (N)	AGE
001	0001	22JUN72	14JUL03	32
001	0002	14DEC69	17JUL03	34
001	0004	13OCT72	18JUL03	31

In the above example, you might notice the usage of new key word **CALCULATED**. This enables you to use the data value of a calculated column AGE, in the immediate **WHERE** clause. It is valid only when used to refer to columns that are calculated in the immediate query expression. Now let's look at other aspect of the example, this query is listing those subjects whose age is less than 40.

The following example, subsets DEMOG table by including only subjects whose age equal to 32:

```
select siteid, subjid, birthdt, csdt, floor((csdt - birthdt)/365.25)+1 as AGE
from   demog
where  calculated age = 32; /* alternate: where calculated age EQ 32;*/
```

OUTPUT:

Site ID	Subject ID	Birth Date (N)	Date Informed Consent Signed (N)	AGE
001	0001	22JUN72	14JUL03	32

EXAMPLE 1-6: SELECT ... FROM ... WHERE ... BETWEEN ... AND

You can retrieve rows based on a range of values using the BETWEEN AND keyword. As in the example:

```
select siteid, subjid,birthdt,csdt, floor((csdt - birthdt)/365.25)+1 as AGE
from demog
where calculated age between 32 and 42; /* alternate: where calculated age GE 32
and calculated age LE 42;*/
```

OUTPUT:

Site ID	Subject ID	Birth Date (N)	Date Informed Consent Signed (N)	AGE
001	0001	22JUN72	14JUL03	32
001	0002	14DEC69	17JUL03	34
001	0003	01MAY62	17JUL03	42
002	0001	20MAR62	29OCT03	42

EXAMPLE 1-7: SELECT ... FROM ... WHERE ... NOT BETWEEN ... AND

You can retrieve rows based on a range of values using the NOT BETWEEN ... AND keyword. As in the example:

```
select siteid, subjid,birthdt,csdt, floor((csdt - birthdt)/365.25)+1 as AGE
from demog
where calculated age not between 32 and 42;
```

OUTPUT:

Site ID	Subject ID	Birth Date (N)	Date Informed Consent Signed (N)	AGE
001	0004	13OCT72	18JUL03	31

Each condition in the WHERE clause has both a positive form and a negative form. The negative form is always the exact opposite of the positive form. For example, the **is not null** condition is true for every row for which the **is null** condition is false. And the **not between** condition is true for every row where the **between** condition is false.

EXAMPLE 1-8: SELECT ... FROM ... WHERE ... IN ...

You can retrieve rows with values that match those in a list using the IN keyword. If the values are of character data type you must enclose these in quotation marks.

```
select siteid, subjid,birthdt,csdt, floor((csdt - birthdt)/365.25)+1 as AGE
from demog
where calculated age in (32,42); /* Negative form: where calculated age not in (32,42);*/
```

OUTPUT:

Site ID	Subject ID	Birth Date (N)	Date Informed Consent Signed (N)	AGE
001	0001	22JUN72	14JUL03	32
001	0003	01MAY62	17JUL03	42
002	0001	20MAR62	29OCT03	42

Using the ORDER BY Clause

To sort rows by the values of specific columns, you can use the ORDER BY clause in the SELECT statement. Specify the keywords ORDER BY, followed by one or more column names separated by commas.

EXAMPLE 1-9: SELECT ... FROM ... ORDER BY ...

You can use column names and ordinal numbers together in the ORDER BY clause. You can also specify whether you want the result sorted in ascending (ASC) or descending (DESC) order. Default is ASC. We tried to cover all key feature of ORDER BY clause in one example:

```
select siteid, subjid,birthdt,csdt
  from demog
 order by siteid, 4 desc;
```

OUTPUT:

Site ID	Subject ID	Birth Date (N)	Date Informed Consent Signed (N)
001	0004	13OCT72	18JUL03
001	0003	01MAY62	17JUL03
001	0002	14DEC69	17JUL03
001	0001	22JUN72	14JUL03
002	0001	20MAR62	29OCT03

Data Correlation: Retrieving Data from Multiple Tables

So far, we've been looking at queries that retrieve data from single table at a time. Now we will look at implementing joins to retrieve data from two or more tables. The results will appear as a single table with columns from all the tables specified in the SELECT column list and meeting the search criteria. The data tables we use at this time are DEMOG and VITALS as listed below.

OUTPUT:

Protocol: NESUG-2004-HW06 CTD Validation: Using PROC SQL Effectively Wednesday, September 8, 2004

Obs	SITEID	SUBJID	SUBJINIT	VISID	VISDT	SYSTBP	DIASBP	TEMP	PULSE	CRFPGNO
1	001	0001	AAA	1	14JUL2003	118	72	98.8	71	4
2	001	0006	BBB	1	17JUL2003	114	60	98.5	63	4
3	001	0003	CCC	1	17JUL2003	113	82	98.8	67	4
4	001	0004	DDD	1	18JUL2003	135	94	97.5	65	4
5	002	0001	BBB	1	29OCT2003	92	58	99.3	86	4
6	001	0001	AAA	2	21JUL2003	119	78	99.7	71	5
7	001	0006	BBB	2	23JUL2003	115	76	.	64	5
8	001	0003	CCC	2	22JUL2003	112	66	98.6	65	5
9	001	0004	DDD	2	24JUL2003	.	65	98.7	70	5
10	002	0001	BBB	2	04NOV2003	118	74	97.9	69	5

Figure: Vitals Table with 10 rows.

OUTPUT:

Protocol: NESUG-2004-HW06 CTD Validation: Using PROC SQL Effectively 10:20 Wednesday, September 8, 2004

Site ID	Subject ID	Subject Initials	Visit ID	Visit Date (N)	Date Informed Consent Signed (N)	Birth Date (N)	Gender	Race	Other Race - Specify	CRF Page Number
001	0001	AAA	1	14JUL2003	14JUL2003	22JUN1972	1	2		1
001	0002	BBB	1	17JUL2003	17JUL2003	14DEC1969	2	3		1
001	0003	CCC	1	17JUL2003	17JUL2003	01MAY1962	1	1		1
001	0004	DDD	1	18JUL2003	18JUL2003	13OCT1972	1	1		1
002	0001	BBB	1	29OCT2003	29OCT2003	20MAR1962	2	5		1

Figure: Demog Table with 5 rows.

Cartesian Product: Cartesian product is the result of a query created using more than one table without using a where clause. The example used below generates a Cartesian product of 50 records. Cartesian product of large tables is huge. Two tables of one thousand rows each produce a Cartesian product of one million rows. In most cases this type of result is unwanted unless one intended to generate every possible combination result set.

EXAMPLE 1-10: Cartesian Product

```
select demog.SITEID, demog.SUBJID, vitals.VISID, demog .gender,vitals.systbp
  from demog, vitals;
quit;
```

Inner Joins: An inner join returns a query result for all of the rows in a table that have one or more matching rows in another table. The most common types of inner joins are equijoins and natural joins. Equijoin compares column values for equality and displays redundant columns in the result. Natural joins compares column values for equality but eliminates the redundant columns from the result.

The WHERE clause in both natural join and equijoin subsets the rows from the DEMOG table, that matches rows in the VITALS table. SITEID and SUBJID are the matching data points.

Natural Join:

EXAMPLE 1-11: Natural Join

```
select b.siteid,b.subjid,a.gender,a.race,b.visitid,b.systbp,b.diasbp,b.temp, b.pulse
  from demog a, vitals b
 where a.siteid = b.siteid
       and a.subjid = b.subjid;
```

OUTPUT:

Row	Site ID	Subject ID	Gender	Race	Visit ID	Systolic Blood Pressure (mmHg)	Diastolic Blood Pressure (mmHg)	Temperature (oF)	Pulse (bpm)
1	001	0001	1	2	1	118	72	98.8	71
2	001	0003	1	1	1	113	82	98.8	67
3	001	0004	1	1	1	135	94	97.5	65
4	002	0001	2	5	1	92	58	99.3	86
5	001	0001	1	2	2	119	78	99.7	71
6	001	0003	1	1	2	112	66	98.6	65
7	001	0004	1	1	2	.	65	98.7	70
8	002	0001	2	5	2	118	74	97.9	69

Figure: Inner Join Output (Natural Join)

One other interesting feature we addressed in the above examples is, **using table aliases**. Using keyword **AS** between table name and alias name is optional. We use table aliases in joins. Qualifying a column name with alias (abbreviated name) name is more convenient than using full name of the table. This feature is most useful in self-joins, where both joining tables are of same name.

Equijoin: For clarity some of the columns have been dropped from both DEMOG and VITALS tables. In this example the redundant columns SITEID, SUBJID are listed in the output.

EXAMPLE 1-11: Equijoin

```
select *
  from demog (keep= SITEID SUBJID VISID GENDER ), vitals (keep= SITEID SUBJID VISID
SYSTBP)
 where demog.siteid = vitals.siteid
       and demog.subjid = vitals.subjid;
```

OUTPUT:

Row	Site ID	Subject ID	Visit ID	Gender	Site ID	Subject ID	Visit ID	Systolic Blood Pressure (mmHg)
1	001	0001	1	1	001	0001	1	118
2	001	0003	1	1	001	0003	1	113
3	001	0004	1	1	001	0004	1	135
4	002	0001	1	2	002	0001	1	92
5	001	0001	1	1	001	0001	2	119
6	001	0003	1	1	001	0003	2	112
7	001	0004	1	1	001	0004	2	.
8	002	0001	1	2	002	0001	2	118

Figure: Inner Join (EquiJoin)

Self Join:

To show comparative relation ship between data values in a table, it is necessary to join columns within the same table. Joining a table to it self is called a self-Join, or reflexive join.

For example in table VITAL we need to derive the number of days between consecutive visits. You can do this by using self join technique as:

EXAMPLE 1-12: Self Join

```
Select a.siteid, a.subjid, a.visdt as visit1 label = 'First Visit',
       b.visdt as visit2 label = 'Second Visit',
       b.visdt - a.visdt as NDAYS label = 'No. of Days Between Visits'
from   vitals a, vitals b
where  a.siteid = b.siteid
       and a.subjid = b.subjid
       and a.visid = b.visid - 1;
```

OUTPUT:

Site ID	Subject ID	First Visit	Second Visit	No. of Days Between Visits
001	0001	14JUL2003	21JUL2003	7
001	0006	17JUL2003	23JUL2003	6
001	0003	17JUL2003	22JUL2003	5
001	0004	18JUL2003	24JUL2003	6
002	0001	29OCT2003	04NOV2003	6

Figure: Self Join

Outer Joins:

An inner join retrieves only those records that satisfy the join condition. An outer join does the same thing but with the addition of returning records for one table in which there were no matching records in the other table.

There are three types of outer joins: left outer, right outer, and full outer. All outer joins retrieve records from both tables, just as an inner join does. However, an outer join retrieves all of the records from one of the tables. A column in the result is NULL if the corresponding input table did not contain a matching record.

LEFT OUTER JOIN

The left outer join retrieves records from both tables, retrieving all the records from the left table and any records from the right table where the condition values match. If there are no matching values in from the right table, the join still retrieves all the records from the left table. Any columns from the right table that are unmatched are left NULL. Consequently, the resulting record set often appears to have incomplete records.

EXAMPLE 1-13: LEFT OUTER JOIN

```
select  a.*, b.gender, b.race, b.csdt, b.birhtdt
from    vitals a left join demog b
on      a.siteid = b.siteid
       and a.subjid = b.subjid;
```

The code above generates a query result keeping all the rows and columns of VITALS table and adds corresponding GENDER, RACE, CSDT, BIRTHDT data values form DEMOG table to it.

SubQueries:

A subquery is a query within a query. The results of the subquery are used by the higher-level query that contains the subquery. Depending on the clause that contains it, a subquery can return a single value or multiple values. In the simplest forms of a subquery, the subquery appears within the WHERE or HAVING clause of another SQL statement.

EXAMPLE 1-14: SubQueries

```
select  siteid, subjid
from    demog
where   subjid||siteid||subjinit not in(select distinct (subjid||siteid||subjinit) from
                                         endstudy);
```

Above query identifies those subjects, who are in DEMOG table and not in ENDSTUDY table. IN key word uses the subquery result, which is a multiple value list.

SubQueries in the FROM Clause of SELECT:

When a SubQuery is used in a FROM clause instead of a Table name, PROC SQL executes the subquery and then uses the resulting rows as a view in the FROM clause. The below code explains this:

EXAMPLE 1-15: SubQueries (contd.)

```
select  a.*, b.gender, b.race, b.csdt, b.birthdt
from    vitals a left join (select siteid, subjid, gender,
                             csdt, birthdt, race from demog) b
on      a.siteid = b.siteid
and     a.subjid = b.subjid;
```

CREATE TABLE statement:

CREATE TABLE is the command used to create new tables. The CREATE TABLE command is part of the DDL. In PROC SQL we can use CREATE TABLE command in three different forms.

The one form is to create a complete new table, the other form is to copy the structure of an existing table without data and the last form is to copy both structure and data from an existing table.

PROC SQL automatically maps any SQL data types used, to SAS supported data types. This feature is handy while building SAS data sets from other relational databases.

EXAMPLE 2.1: Creating a new Empty Table

The CREATE TABLE statement creates an empty table-one with no records. The parameters that you must supply are name of the table, a list of the columns in the table and a description of the columns (data type, size, format, informat and label). A valid table must have at least one column.

```
create table DEMO
(
  PROTCLID  Varchar(20) label = 'Protocol ID',
  SITEID    Varchar(3)   label = 'Site ID',
  SUBJID    Varchar(4)   label = 'Subject ID',
  SUBJINIT  Varchar(3)   label = 'Subject Initials',
  VISID     Integer      label = 'Visit ID',
  VISDTC    Char(9)      label = 'Visit Date (C)',
  VISDT     Date         label = 'Visit Date (N)',
  CSDTC     Char(9)      label = 'Date Informed Consent Signed (C)',
  CSDT      Date         label = 'Date Informed Consent Signed (N)',
  CSTMC     Char(4)      label = 'Time Informed Consent Signed (C)',
  CSTM      Date format = time5. label = 'Time Informed Consent Signed (N)',
  BIRTHDTC  Char(9)      label = 'Birth Date (C)',
  BIRTHDT   Date         label = 'Birth Date (N)',
  GENDER     Num(8)      label = 'Gender',
  RACE       Num(8)      label = 'Race',
  OTHRACE    Char(20)    label = 'Other Race - Specify',
  CRFPGNO    Num(8)      label = 'CRF Page Number');
```

LOG: NOTE: Table WORK.DEMO created, with 0 rows and 17 columns.

EXAMPLE 2.2: Using DESCRIBE Statement

Use DESCRIBE statement to know the underlying CREATE TABLE statement of a table. DESCRIBE statement writes the output to the SAS log.

```
describe table demo;
```

Log: NOTE: SQL table WORK.DEMO was created like:

```
create table WORK.DEMO( bufsize=12288 )
  (PROTCLID char(20) label='Protocol ID',
   SITEID char(3) label='Site ID',
   SUBJID char(4) label='Subject ID',
   SUBJINIT char(3) label='Subject Initials',
   VISID num label='Visit ID',
   VISDTC char(9) label='Visit Date (C)',
   VISDT num format=DATE. informat=DATE. label='Visit Date (N)',
   CSDTC char(9) label='Date Informed Consent Signed (C)',
   CSDT num format=DATE. informat=DATE. label='Date Informed Consent Signed (N)',
   CSTMC char(4) label='Time Informed Consent Signed (C)',
   CSTM num format=TIME5. informat=DATE. label='Time Informed Consent Signed (N)',
   BIRTHDTC char(9) label='Birth Date (C)',
   BIRTHDT num format=DATE. informat=DATE. label='Birth Date (N)',
   GENDER num label='Gender',
   RACE num label='Race',
   OTHRACE char(20) label='Other Race - Specify',
   CRFPGNO num label='CRF Page Number');
```

Use data set options in conjunction with DESCRIBE statement, to build subset of the CREATE TABLE statement of an existing table.

```
describe table demo(drop = VISDTC CSDTC BIRTHDTC);
```

Log:

```
create table WORK.DEMO( bufsize=12288 )
  (PROTCLID char(20) label='Protocol ID',
   SITEID char(3) label='Site ID',
   SUBJID char(4) label='Subject ID',
   SUBJINIT char(3) label='Subject Initials',
   VISID num label='Visit ID',
   VISDT num format=DATE. informat=DATE. label='Visit Date (N)',
   CSDT num format=DATE. informat=DATE. label='Date Informed Consent Signed (N)',
   CSTMC char(4) label='Time Informed Consent Signed (C)',
   CSTM num format=TIME5. informat=DATE. label='Time Informed Consent Signed (N)',
   BIRTHDT num format=DATE. informat=DATE. label='Birth Date (N)',
   GENDER num label='Gender',
   RACE num label='Race',
   OTHRACE char(20) label='Other Race - Specify',
   CRFPGNO num label='CRF Page Number');
```

EXAMPLE 2.3: Creating an Empty Table from an Existing Table

Using LIKE clause in conjunction with CREATE TABLE statement is a quick and simple one step answer for this task. You are creating a table that has the same structure and attributes of another table referred.

```
create table DEMOG like DEMO;
```

Log: NOTE: Table WORK.DEMOG created, with 0 rows and 17 columns.

For instance in the above-referred table DEMOG you would like to change column name BIRTHDT to BDATE. Use SAS data set option RENAME. You can use this data set option on both referenced and created tables. There won't be any special message in SAS log except like the one in the above example.

```
create table DEMOG like DEMO(rename = (crfpgno = crfpg));
create table DEMOG(rename = (crfpgno = crfpg)) like DEMO;
```

EXAMPLE 2.4: Creating a Table with Data from an Existing Table

One other way of creating a table is based on the result of a query. This is a most effective way of creating a table, which is a direct result of the SQL query.

For example, for some reason you need to create a table with all male subjects participated in a study. You can simply create this table by using the SQL command:

EXAMPLE 2.4.1:

```
create table M_SUBJECTS as
  select * from DEMO
  where gender = 1;
```

LOG: NOTE: Table WORK.DEMOG created, with 3 rows and 17 columns.

Now as you did in other examples, you can modify the above SQL command by converting SQL WHERE clause to SAS data set option. This way you are filtering the data at source level.

EXAMPLE 2.4.2:

```
create table M SUBJECTS as
  select * from DEMO (where = (gender = 1));
```

LOG: NOTE: Table WORK.DEMOG created, with 3 rows and 17 columns.

For example you need to exclude CSTMC and CSTM variables in the new table DEMOG. You can do so by using SAS data set option in CREATE TABLE statement.

Regular Method: The normal SQL syntax looks like

EXAMPLE 2.4.3:

```
create table DEMOG as
  select PROTCLID, SITEID, SUBJID, SUBJINIT, VISID, VISDTC, VISDT, CSDTC, CSDT
         CSTMC, CSTM, BIRTHDTC, BIRTHDT, GENDER, RACE, OTHRACE, CRFPGNO
  from DEMO;
```

LOG: NOTE: Table WORK.DEMOG created, with 5 rows and 15 columns.

Using SAS DATA SET OPTION DROP:

EXAMPLE 2.4.4:

```
create table DEMOG as
  select *
  from DEMO (drop = CSTMC CSTM);
```

LOG: NOTE: Table WORK.DEMOG created, with 5 rows and 15 columns.

EXAMPLE 3: Adding Data to an Existing Table

Now that we have created DEMOG table, let's enter some data into it. You can add data to an existing table by using INSERT INTO statement. The INSERT INTO statement is part of the Data Manipulation Language.

In **PROC SQL** we can use INSERT statement in three different ways.

1. Inserting values using INSERT INTO statement with SET clause.
2. Inserting values using INSERT INTO statement with VALUES clause.
3. Inserting values using INSERT INTO statement with SQL Query.

Using INSERT INTO ... SET ...

EXAMPLE 3.1:

```
insert into DEMOG
  set visid = 2,
    protclid = 'NESUG-2004-HW06',
    subjid = '0001',
    siteid = '001'
  set protclid = 'NESUG-2004-HW06',
    visid = 2,
    subjid = '0001',
    siteid = '002';
```

Log: NOTE: 2 rows were inserted into WORK.DEMOG.

In this form of INSERT INTO statement you need to specify table name, column name and data value for each column. To do so you need to know about the structure of table into which you are entering data. The simplest method of finding a table attributes including column names is, using the DESCRIBE statement on the table you want to insert the data.

Using INSERT INTO ... VALUES ...

EXAMPLE 3.2.1:

```
insert into DEMOG
  values('NESUG-2004-HW06','001','0002','','2','','','','','','','','','')
  values('NESUG-2004-HW06','001','0001','','2','','','','','','','','','');
```

Log:

```
315 proc sql;
316     insert into DEMOG
317         values('NESUG-2004-HW06','001','0002','','2','','','','','','','','','')
318         values('NESUG-2004-HW06','001','0001','','2','','','','','','','','','');
ERROR: VALUES clause 2 attempts to insert more columns than specified after the INSERT table name.
ERROR: Value 16 of VALUES clause 2 does not match the data type of the corresponding column in the
      object-item list (in the SELECT clause).
ERROR: Value 17 of VALUES clause 2 does not match the data type of the corresponding column in the
      object-item list (in the SELECT clause).
319 quit;
320
```

NOTE: The SAS System stopped processing this step because of errors.

In this form of INSERT INTO statement you must specify the table name and data values only. However, you need to specify the data value of each related column in the order of its physical position in the table. Each missing data value must be provided with an appropriate blank or period. In the above example there is an error in the second value clause this forced SAS to stop processing the entire insert statement and there won't be any rows included to DEMOG table.

EXAMPLE 3.2.2:

```
insert into DEMOG
  values('NESUG-2004-HW06','001','0002','','2','','','','','','','','','');
insert into DEMOG
  values('NESUG-2004-HW06','001','0001','','2','','','','','','','','','');
```

Log:

```
320 proc sql;
321     insert into DEMOG
322         values('NESUG-2004-HW06','001','0002','','2','','','','','','','','','');
NOTE: 1 row was inserted into WORK.DEMOG.
323     insert into DEMOG
324         values('NESUG-2004-HW06','001','0001','','2','','','','','','','','','');
```

ERROR: VALUES clause 1 attempts to insert more columns than specified after the INSERT table name.
ERROR: Value 16 of VALUES clause 1 does not match the data type of the corresponding column in the object-item list (in the SELECT clause).
ERROR: Value 17 of VALUES clause 1 does not match the data type of the corresponding column in the object-item list (in the SELECT clause).

325 quit;

NOTE: The SAS System stopped processing this step because of errors.

This time we have two INSERT INTO statements with one values clause in each of them. SAS system processed the insert statement, the one without errors, and stopped processing the one with errors.

```
insert into DEMOG (PROTCLID, SITEID, SUBJID, VISID)
  values      ('NESUG-2004-HW06', '001', '0003', 3);
```

This time the data values listed in values clause are related to those selected column names as listed in the column list.

Using INSERT INTO ... query-expression

The INSERT INTO statement can be used to add more than one row at a time to a table. To do this, the VALUES clause must be replaced with a SELECT statement that retrieves the required rows from a second table. As an example, suppose we create a table called F_SUBJECTS, which is just like table DEMOG, to hold all female subjects demographic information. The following query will populate it:

EXAMPLE 3.3.1:

```
create table F_SUBJECTS like DEMOG;
insert into F_SUBJECTS
  select * from DEMOG
  where gender = 2;
```

Log:

```
490 proc sql;
491   create table F_SUBJECTS like DEMOG;
NOTE: Table WORK.F_SUBJECTS created, with 0 rows and 17 columns.
492   insert into F_SUBJECTS
493     select * from DEMOG
494     where gender = 2;
NOTE: 1 row was inserted into WORK.F_SUBJECTS.
```

EXAMPLE 3.3.2:

The INSERT INTO statement with SELECT clause can be used with column names list to insert only selected column's data values. In the example below data set option DROP is used to create column names list.

```
create table F_SUBJECTS like DEMOG;
insert into F_SUBJECTS (drop = VISDTC CSDTC CSTMC BIRTHDTC)
  select * from DEMOG(drop = VISDTC CSDTC CSTMC BIRTHDTC)
  where gender = 2;
```

EXAMPLE 4: Dropping an Existing Table

In PROC SQL, using DROP statement you can remove a table. The drop statement is part of Data Definition Language. Since the DROP TABLE statement removes a table permanently, it is important to ensure that the table should not be accessed by any VIEWS. You should always exercise caution in using DROP TABLE statement.

As an example to delete the F_SUBJECTS table:

```
drop table F_SUBJECTS;
```

Log: NOTE: Table WORK.F_SUBJECTS has been dropped.

EXAMPLE 5: Removing Rows From Tables

You might want to delete some of the data rows from tables. SQL allows you to remove data by using the DELETE statement. This statement is part of Data Manipulation Language. DELETE statement allows you to remove one or more rows from tables. This statement works on an entire row, you cannot work on an individual column value with this statement.

When used without a conditional clause, DELETE removes all the rows from a table. To clear the DEMOG table of all data:

EXAMPLE 5.1:

```
delete from DEMOG;
```

LOG: NOTE: 5 rows were deleted from WORK.DEMOG.

EXAMPLE 5.2:

DELETE statement allows the use of the WHERE clause to selectively remove rows from a table. In the ENTCRIT table, suppose you want to remove a subject record whose initial is 'BBB' , ID is '0001', and site number is '002'. You can do this by:

```
delete from ENTCRIT
where SITEID = '002' and subjid = '0001' and subjinit = 'BBB';
```

LOG: NOTE: 1 row was deleted from WORK.ENTCRIT.

EXAMPLE 6: Using VALIDATE query-expression

VALIDATE statement in PROC SQL helps in verifying the syntactical errors in the SQL statement without executing the statement. When used in conjunction with macro facility a value is returned through the macro variable SQL Return Code (SQLRC).

EXAMPLE 7: Altering Existing Tables

It's not too late to modify an existing table. We can add, modify, or drop columns, and add or drop the constraints, among other things, using ALTER statement.

Using ALTER TABLE Statement

The ALTER TABLE statement is part of Data Definition Language. With this statement we can change the structure of a table. New columns can be added with the ADD clause. Existing columns can be modified with the MODIFY clause. Columns can be dropped from a table with DROP clause.

Please note that you cannot use data set options in this ALTER TABLE statement as we did in earlier examples. Here are some of the most commonly used ALTER TABLE statements:

Using ALTER TABLE ... ADD ...

EXAMPLE 7.1:

```
alter table demog
add AGE num label = 'Subject Age'; /* you can also specify informat and format */
```

LOG: NOTE: Table WORK.DEMOG has been modified, with 18 columns.

Using ALTER TABLE ... MODIFY ...

EXAMPLE 7.2:

```
alter table demog
modify csdt format = MMDDYY10.,
birthdt format = MMDDYY10.; /* you can include more than one modification
request in one modify clause*/
```

LOG: NOTE: Table WORK.DEMOG has been modified, with 18 columns.

The above SAS log is not close to the point. From the log it's clear that there has been some modification on the DEMOG table and after modification the table DEMOG has 18 columns.

Using ALTER TABLE ... DROP ...

EXAMPLE 7.3:

```
alter table demog
  drop CSTMC, CSTM;
```

In the above example we dropped two columns, related to consent time, CSTMC and CSTM from DEMOG table using ALTER TABLE statement in conjunction with DROP clause.

Using ALTER TABLE ... ADD ... MODIFY ... DROP ...

You can also combine more than one clause in one ALTER TABLE statement like:

EXAMPLE 7.4:

```
alter table demog
  add AGE num label = 'Subject Age'
  modify csdt format = MMDDYY10., birthdt format = MMDDYY10.
  drop CSTMC, CSTM;
```

LOG: NOTE: Table WORK.DEMOG has been modified, with 16 columns.

This time we used all three clauses in one ALTER TABLE statement. We added one column using ADD clause, modified two columns using MODIFY clause, remember we never dropped them, and dropped two columns using DROP clause. At the end we have 16 columns in DEMOG table. This is what SAS log is saying.

EXAMPLE 8: Updating Existing Records

After a record has been entered, we may wish to change it or add more information to it. The SQL UPDATE statement is used to change the existing values of the columns.

Using UPDATE ... SET ...

In its simplest form an UPDATE statement needs a table name, column name and data value to set the column to. Using UPDATE statement we can update one or more columns in a single statement. This statement is part of Data Modification Language. The basic UPDATE statement looks like:

EXAMPLE 8.1:

```
update demog
  set PROTCLID = 'NESUG2004-HW06';
```

LOG: NOTE: 5 rows were updated in WORK.DEMOG.

In the above example, we tried to fix the format of data value of column PROTCLID. By using UPDATE statement this way, we can change the column values of all the rows in the table at once.

EXAMPLE 8.2:

```
alter table demog
  add AGE num label = 'Subject Age' ;
update demog
  set age = floor((csdt - birthdt)/365.25)+1;
```

LOG: NOTE: 5 rows were updated in WORK.DEMOG.

This time we tried a little differently, DEMOG table has been updated with a calculated data value, which is calculated by using consent date and birth date columns of same table. Of course, we altered the DEMOG table first to add AGE column.

Using UPDATE ... SET ... WHERE ...

To update a group of rows within a table conditionally we need to use WHERE clause as:

EXAMPLE 8.3:

```
update demog
  set OTHRACE = 'UNKNOWN'
  where race = 5;
```

Log: NOTE: 1 row was updated in WORK.DEMOG.

Here OTHRACE column of a specific group of rows, whose RACE column is equal to 5, has been updated.

SECTION 3: CLINICAL DATA VALIDATION

After having a glance at SQL and reviewing the basics of PROC SQL, I hope every one is interested see how we can tap in all this clinical data validation. Let's get over it.

Batch reviews, programmed edit checks executable in batch mode, and manual reviews are executed against the clinical database to identify potential errors. The CDM programmer programs the edit checks listed in the data validation document and generates the reports. The data manager reviews each discrepancy and gets resolved by communicating to the study investigator.

A list of these edit checks is as follows:

- Domain checks:
- Missing Value Checks
- Range Checks
- Frequency Count Checks
- Duplicate Record Checks
- Referential Integrity Checks
- Data Consistency Checks
- Unique Value Checks
- Protocol Violation Checks
- Data rule compliance Checks
- Format Consolidation Checks

In the next few pages we will go through some of these checks with code examples and analysis of each code snippet used.

Discrepancy Data Table:

To hold all discrepancies, generated out of programmatic checks, in a clinical trial a table is required. So create a table and move each query result into it. In this clinical data validation part we are using only PROC SQL. The statements uses are CREATE TABLE, INSERT INTO, and SELECT. Of course we have to deal with lots of clauses in each of these statements, glad we reviewed most of them in the earlier sessions.

Program:

```
create table CTDVALID (Label = 'Clinical Trial Data Validation')
  (CHECKNO Char(20) label = 'Unique Check Number',
   PATID Char(10) label = 'Unique Patient ID',
   RECID Char(40) label = 'Unique Record Identifier',
   ERRMSG Char(100) label = 'ERROR MESSAGE'
  );
```

LOG: NOTE: Table WORK.CTDVALID created, with 0 rows and 4 columns.

Analysis:

The previous CREATE TABLE statement is a significantly simplified version, which solves our purpose here.

Domain checks

Domain checks identify the columns having non-possible values.

Check: The patient gender code should be recorded as 1 (Male) or 2 (Female).

Program:

```
insert into CTDVALID
select 'DEMOG-0001' as CHECKNO,
       SITEID||'-'||SUBJID as PATID,
       'Visit:('||put(VISID,z3.)||'); Page:('||put(CRFPGNO,z3.)||')' as RECID,
       'Gender value should be 1 = Male or 2 = Female' as ERRMSG
from   demog
where  gender not in (1,2);
```

LOG: NOTE: 1 row was inserted into WORK.CTDVALID.

Analysis:

The NOT IN keyword is used, in the where clause, to retrieve the rows of those subjects whose gender code is not of 1 or 2. If column gender is of type character then these values must be enclosed in single quotation marks.

Missing Value Checks

These checks identify the columns with missing data.

Check: A valid birth date should be present in patient demography records.

Program:

```
insert into CTDVALID
select 'DEMOG-0002' as CHECKNO,
       SITEID||'-'||SUBJID as PATID,
       'Visit:('||put(VISID,z3.)||'); Page:('||put(CRFPGNO,z3.)||')'
       as RECID,
       'Birth date cannot be missing' as ERRMSG
from   demog
where  birthdt is missing;
```

LOG: NOTE: 1 row was inserted into WORK.CTDVALID.

Analysis:

In the RDBMS world a missing value is not a blank or zero it's an unknown called *null*. In Proc SQL a null value is same as blank or zero value.

The IS MISSING operator enables to identify rows that contain columns with missing values. The above example selects rows of those subjects have a missing value in the BIRTHDT column. The IS NULL operator is interchangeable with the IS MISSING operator.

Range Checks

The range check verifies whether a data value lies between two specified values. These checks identify data out-of-range discrepancies in numeric data. This can be protocol violation or an out-of-range value compared to normal ranges.

Check: Temperature should be between 96 and 100 °F.

Program:

```
insert into CTDVALID
select 'VITALS-0001' as CHECKNO,
       SITEID||'-'||SUBJID as PATID,
       'Visit:('||put(VISID,z3.)||'); Page:('||put(CRFPGNO,z3.)||')' as RECID,
```

```

      'TEMP must be within 96 - 104 F' as ERRMSG
from    vitals
where   int(temp) not between 96 and 104;

```

LOG: NOTE: 2 rows were inserted into WORK.CTDVALID.

Analysis:

In the above example NOT BETWEEN.... AND operators are used to identify rows outside the range 96 and 104 both inclusive.

Frequency Count Checks

These checks allow verifying the number of instances of a value or values across data positions or fields.

Check: In table VITALS each subject should have two rows.

Program:

```

insert into CTDVALID
select 'VITALS-0001' as CHECKNO,
      SITEID||'-'||SUBJID as PATID,
      '' as RECID,
      'Subject has more than two rows' as ERRMSG
from   (select siteid, subjid, count(visit) as NVISITS
        from   vitals
        group by siteid, subjid)
where  nvisits > 2;

```

LOG: NOTE: 2 rows were inserted into WORK.CTDVALID.

Analysis:

In the above example we used a subquery in FROM clause. PROC SQL generates a view from this subquery and executes the higher-level SELECT statement on this view. It's like querying a view.

Now finally we got some data into our discrepancy table CTDVALID. Before making listing of it let's look at its structure. This table has four columns. Column labels explain the purpose of these columns. However, RECID column is there to hold information required, other than site number and subject number, to identify the specific row. This helps the data manager, when he/she looks at the report to generate DCRs based on your data validation report.

Here is the image of the report. To generate this report you can use SQL reporting features. Which we haven't dealt much here, except select statement. Don't compare the data values in this report to demog table. There is no subject having data values other than 1 or 2 in GENDER column. This is to give an idea how report looks like.

The PROC SQL SELECT statement output:

Protocol: NESUG-2004-HW06		CTD Validation: Using PROC SQL Effectively		15:17 Wednesday, September 8, 2004
Unique Check Number	Unique Patient ID	Unique Record Identifier	ERROR MESSAGE	
DEMOG-0001	001-0002	Visit:{001}; Page:{001}	Gender value should be 1 = Male or 2 = Female	
DEMOG-0001	002-0001	Visit:{001}; Page:{001}	Gender value should be 1 = Male or 2 = Female	

The PROC PRINT output:

Protocol: NESUG-2004-HW06		CTD Validation: Using PROC SQL Effectively		Wednesday, September 8, 2004
Obs	CHECKNO	PATID	RECID	ERRMSG
1	DEMOG-0001	001-0002	Visit:{001}; Page:{001}	Gender value should be 1 = Male or 2 = Female
2	DEMOG-0001	002-0001	Visit:{001}; Page:{001}	Gender value should be 1 = Male or 2 = Female

All good things have to come to an end though. In this paper we tried to cover some of the PROC SQL features and a few clinical data validation checks using PROC SQL. Now it's your turn to explore in these lines.

CONCLUSION

The PROC SQL is an excellent and an efficient tool in BASE SAS. Querying clinical data using PROC SQL is less time consuming and efficient. Query programs generated using PROC SQL are of vendor independent and easily portable into all most every data management system available at present, with minimal re-coding effort. However, one needs to exercise caution while using SAS data set options and other system options, in PROC SQL programming, if they intended to use these programs outside the SAS environment.

PROC SQL bridges the gap between SAS and other relational databases. It plays a major role in connecting and accessing data in other Relational Data Base Systems.

Indeed, the PROC SQL is a valuable one step alternative to many of the tasks covered by SAS data step and few other procedures.

All code developed under Windows 2000 Pro system, running release 8.2 of the SAS System.

REFERENCES

1. SAS Institute Inc., *SAS® Procedures Guide, Version 8*, Cary, NC: SAS Institute Inc., 1999. 1729 pp.

ACKNOWLEDGMENTS

The author would like to express his deepest gratitude to Mr. Glenn Adams, Vice President Data Management of SFBC New Drug Services and Mr. Stephen Dorian, Director Data Management of SFBC New Drug Services. Without their encouragement, this paper might not have come alive.

SFBC New Drug Services is a wholly owned subsidiary of SFBC International, Inc. (NASDAQ symbol: SFCC). The SFBC New Drug Services division provides Phase I through Phase IV contract research services to the pharmaceutical industry. The company combines over 25 years of experience and offers services in clinical trial planning and monitoring, project management, data management, biostatistics, medical writing, NDA and ANDA preparation including electronic submissions. In addition, SFBC New Drug Services offers complete product development and strategic consulting and management services. SFBC International, Inc. was founded in 1984 in Miami, Florida as a Phase I clinic, and through organic growth and strategic acquisitions has developed into a world-class contract research organization. SFBC International has been named by Forbes Magazine as one of the 200 best companies in America for 2002.

TRADEMARKS

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

Other brand and product names are trademarks of their respective companies.

CONTACT INFORMATION

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

Balakrishna Dandamudi
Senior SAS Programmer
SFBC New Drug Services
Longwood Corporate Center South
415 McFarlan Road, Suite 201
Kennett Square, PA 19348-2412
Work Phone: 610.444.4722
Fax: 610.444.4663
E-mail: kdandamudi@sfbci.com