CHAPTER

34

The SQL Procedure

Overview

The SQL procedure implements Structured Query Language (SQL) for the SAS System. SQL is a standardized, widely used language that retrieves and updates data in tables and views based on those tables.

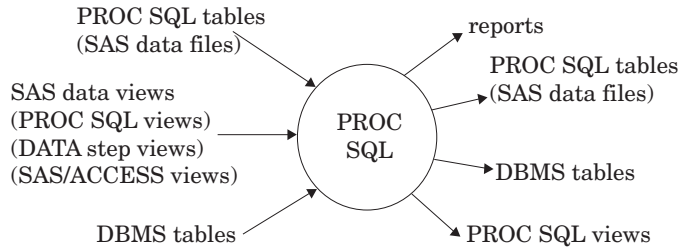
The SAS System's SQL procedure enables you to

- retrieve and manipulate data that are stored in tables or views.
- create tables, views, and indexes on columns in tables.
- create SAS macro variables that contain values from rows in a query's result.

- add or modify the data values in a table's columns or insert and delete rows. You can also modify the table itself by adding, modifying, or dropping columns.
- send DBMS-specific SQL statements to a database management system (DBMS) and to retrieve DBMS data.

Figure 34.1 on page 1023 summarizes the variety of source material that you can use with PROC SQL and what the procedure can produce.

Figure 34.1 PROC SQL Input and Output



What Are PROC SQL Tables?

A PROC SQL *table* is synonymous with a SAS data file and has a member type of DATA. You can use PROC SQL tables as input into DATA steps and procedures.

You create PROC SQL tables from SAS data files, from SAS data views, or from DBMS tables using PROC SQL's Pass-Through Facility. The Pass-Through Facility is described in "Connecting to a DBMS Using the SQL Procedure Pass-Through Facility" on page 1095.

In PROC SQL terminology, a *row* in a table is the same as an *observation* in a SAS data file. A *column* is the same as a *variable*.

What Are Views?

A *SAS data view* defines a virtual data set that is named and stored for later use. A view contains no data but describes or defines data that are stored elsewhere. There are three types of SAS data views:

- PROC SQL views
- SAS/ACCESS views
- DATA step views.

You can refer to views in queries as if they were tables. The view derives its data from the tables or views that are listed in its FROM clause. The data accessed by a view are a subset or superset of the data in its underlying table(s) or view(s).

A PROC SQL view is a SAS data set of type VIEW created by PROC SQL. A PROC SQL view contains no data. It is a stored query expression that reads data values from its underlying files, which can include SAS data files, SAS/ACCESS views, DATA step views, other PROC SQL views, or DBMS data. When executed, a PROC SQL view's output can be a subset or superset of one or more underlying files.

SAS/ACCESS views and DATA step views are similar to PROC SQL views in that they are both stored programs of member type VIEW. SAS/ACCESS views describe data in DBMS tables from other software vendors. DATA step views are stored DATA step programs.

You can update data through a PROC SQL or SAS/ACCESS view with certain restrictions. See “Updating PROC SQL and SAS/ACCESS Views” on page 1097.

You can use all types of views as input to DATA steps and procedures.

Note: In this chapter, the term *view* collectively refers to PROC SQL views, DATA step views, and SAS/ACCESS views, unless otherwise noted. △

SQL Procedure Coding Conventions

Because PROC SQL implements Structured Query Language, it works somewhat differently from other base SAS procedures, as described here:

- You do not need to repeat the PROC SQL statement with each SQL statement. You need only to repeat the PROC SQL statement if you execute a DATA step or another SAS procedure between statements.
- SQL procedure statements are divided into clauses. For example, the most basic SELECT statement contains the SELECT and FROM clauses. Items within clauses are separated with commas in SQL, not with blanks as in the SAS System. For example, if you list three columns in the SELECT clause, the columns are separated with commas.
- The SELECT statement, which is used to retrieve data, also outputs the data automatically unless you specify the NOPRINT option in the PROC SQL statement. This means you can display your output or send it to a list file without specifying the PRINT procedure.
- The ORDER BY clause sorts data by columns. In addition, tables do not need to be presorted by a variable for use with PROC SQL. Therefore, you do not need to use the SORT procedure with your PROC SQL programs.
- A PROC SQL statement runs when you submit it; you do not have to specify a RUN statement. If you follow a PROC SQL statement with a RUN statement, the SAS System ignores the RUN statement and submits the statements as usual.

Procedure Syntax

Tip: Supports the Output Delivery System. (See Chapter 2, "Fundamental Concepts for Using Base SAS Procedures" for information on the Output Delivery System.)

Reminder: See Chapter 3, "Statements with the Same Function in Multiple Procedures," for details. You can also use any global statements as well. See Chapter 2, "Fundamental Concepts for Using Base SAS Procedures," for a list.

Note:

Regular type indicates the name of a component that is described in “Component Dictionary” on page 1056.

view-name indicates a SAS data view of any type.

PROC SQL <*option(s)*>;
  ALTER TABLE *table-name*
      <*constraint-clause*> <,*constraint-clause*>…>;
  <**ADD** column-definition <,column-definition>…>
  <**MODIFY** column-definition
      <,column-definition>…>

<**DROP** *column* <,*column*>…>;

CREATE <**UNIQUE**> **INDEX** *index-name*
ON *table-name (column* <,*column*>…);

CREATE TABLE *table-name* (column-definition <,column-definition>…);
(column-specification , ...<*constraint-specification* > ,...) ;

CREATE TABLE *table-name* **LIKE** *table-name*;

CREATE TABLE *table-name* **AS** query-expression
<**ORDER BY** *order-by-item* <,*order-by-item*>…>;

CREATE VIEW *proc-sql-view* **AS** query-expression
<**ORDER BY** *order-by-item* <,*order-by-item*>…>;
<**USING** *libname-clause*<, *libname-clause*>...>;

DELETE
FROM *table-name*|*proc-sql-view* |*sas*/*access-view* <**AS** *alias*>
<**WHERE** sql-expression>;

DESCRIBE TABLE*table-name*<,*table-name*>… ;

DESCRIBE TABLE CONSTRAINTS *table-name* <, *table-name*>… ;

DESCRIBE VIEW *proc-sql-view* <,*proc-sql-view*>… ;

DROP INDEX *index-name* <,*index-name*>…
FROM *table-name*;

DROP TABLE *table-name* <,*table-name*>…;

DROP VIEW *view-name* <,*view-name*>…;

INSERT INTO *table-name*|*sas*/*access-view*|*proc-sql-view* <(*column*<,*column*>…) >
SET *column*=sql-expression
<,*column*=sql-expression>…
<**SET** *column*=sql-expression
<,*column*=sql-expression>…>;

INSERT INTO *table-name*|*sas*/*access-view*|*proc-sql-view*<(*column*<,*column*>…)>
VALUES (*value*<,*value*>…)
<**VALUES** (*value* <,*value*>…)>…;

INSERT INTO *table-name*|*sas*/*access-view*|*proc-sql-view*
<(*column*<,*column*>…)> query-expression;

RESET <*option(s)*>;

SELECT <DISTINCT> *object-item* <,*object-item*>…
<**INTO** :*macro-variable-specification*
<, :*macro-variable-specification*>…>
FROM *from-list*
<**WHERE** sql-expression>
<**GROUP BY** *group-by-item*
<,*group-by-item*>…>
<**HAVING** sql-expression>
<**ORDER BY** *order-by-item*
<,*order-by-item*>…>;

UPDATE *table-name*|*sas*/*access-view*|*proc-sql-view* <**AS** *alias*>
SET *column*=sql-expression
<,*column*=sql-expression>…
<**SET***column*=sql-expression
<,*column*=sql-expression>…>
<**WHERE** sql-expression>;

VALIDATEquery-expression;

To connect to a DBMS and send it a DBMS-specific nonquery SQL statement, use this form:

PROC SQL;
  <**CONNECT TO** *dbms-name* <AS *alias*><
    <(*connect-statement-argument-1=value*
    …<*connect-statement-argument-n=value*>)>>
    <(*dbms-argument-1=value*
    …<*dbms-argument-n=value*>)>>;
  EXECUTE (*dbms-SQL-statement*)
    BY *dbms-name*|*alias*;
  <**DISCONNECT FROM** *dbms-name*|*alias*;>
<**QUIT**;>

To connect to a DBMS and query the DBMS data, use this form:

PROC SQL;
  <**CONNECT TO** *dbms-name* <AS *alias*><
    <(*connect-statement-argument-1=value*
    …<*connect-statement-argument-n=value*>)>>
    <(*dbms-argument-1=value*
    …<*dbms-argument-n=value*>)>>;
  SELECT *column-list*
    FROM CONNECTION TO *dbms-name*|*alias*
      (*dbms-query*)
      optional PROC SQL clauses;
  <**DISCONNECT FROM** *dbms-name*|*alias*;>
<**QUIT**;>

To do this	Use this statement
Modify, add, or drop columns	ALTER TABLE
Establish a connection with a DBMS	CONNECT
Create an index on a column	CREATE INDEX
Create a PROC SQL table	CREATE TABLE
Create a PROC SQL view	CREATE VIEW
Delete rows	DELETE
Display a definition of a table or view	DESCRIBE
Terminate the connection with a DBMS	DISCONNECT
Delete tables, views, or indexes	DROP
Send a DBMS-specific nonquery SQL statement to a DBMS	EXECUTE
Add rows	INSERT
Reset options that affect the procedure environment without restarting the procedure	RESET

To do this	Use this statement
Select and execute rows	SELECT
Query a DBMS	CONNECTION TO
Modify values	UPDATE
Verify the accuracy of your query	VALIDATE

PROC SQL Statement

PROC SQL <*option(s)*>;

To do this	Use this option
Control output	
Double-space the report	DOUBLE|NODOUBLE
Write a statement to the SAS log that expands the query	FEEDBACK|NOFEEDBACK
Flow characters within a column	FLOW|NOFLOW
Include a column of row numbers	NUMBER|NONUMBER
Specify whether PROC SQL prints the query's result	PRINT|NOPRINT
Specify whether PROC SQL should display sorting information	SORTMSG|NOSORTMSG
Specify a collating sequence	SORTSEQ=
Control execution	
Allow PROC SQL to use names other than SAS names	DQUOTE=
Specify whether PROC SQL should stop executing after an error	ERRORSTOP|NOERRORSTOP
Specify whether PROC SQL should execute statements	EXEC|NOEXEC
Restrict the number of input rows	INOBS=
Restrict the number of output rows	OUTOBS=
Restrict the number of loops	LOOPS=
Specify whether PROC SQL prompts you when a limit is reached with the INOBS=, OUTOBS=, or LOOPS= options	PROMPT|NOPROMPT

To do this	Use this option
Specify whether PROC SQL writes timing information to the SAS log	STIMER|NOSTIMER
Specify how PROC SQL handles updates when there is an interruption	UNDO_POLICY=

Options

DOUBLE|NODOUBLE
double-spaces the report.

Default: NODOUBLE

Featured in: Example 5 on page 1108

DQUOTE=ANSI|SAS
specifies whether PROC SQL treats values within double-quotes as variables or strings. With DQUOTE=ANSI, PROC SQL treats a quoted value as a variable. This enables you to use the following as table names, column names, or aliases:

- reserved words such as AS, JOIN, GROUP, and so on.
- DBMS names and other names not normally permissible in SAS.

The quoted value can contain any character.

With DQUOTE=SAS, values within quotes are treated as strings.

Default: SAS

ERRORSTOP|NOERRORSTOP
specifies whether PROC SQL stops executing if it encounters an error. In a batch or noninteractive session, ERRORSTOP instructs PROC SQL to stop executing the statements but to continue checking the syntax after it has encountered an error.

NOERRORSTOP instructs PROC SQL to execute the statements and to continue checking the syntax after an error occurs.

Default: NOERRORSTOP in an interactive SAS session; ERRORSTOP in a batch or noninteractive session

Interaction: This option is useful only when the EXEC option is in effect.

Tip: ERRORSTOP has an effect only when SAS is running in the batch or noninteractive execution mode.

Tip: NOERRORSTOP is useful if you want a batch job to continue executing SQL procedure statements after an error is encountered.

EXEC | NOEXEC
specifies whether a statement should be executed after its syntax is checked for accuracy.

Default: EXEC

Tip: NOEXEC is useful if you want to check the syntax of your SQL statements without executing the statements.

See also: ERRORSTOP on page 1028 option

FEEDBACK|NOFEEDBACK
specifies whether PROC SQL displays a statement after it expands view references or makes certain transformations on the statement.

This option expands any use of an asterisk (for example, **`SELECT *`**) into the list of qualified columns that it represents. Any PROC SQL view is expanded into the

underlying query, and parentheses are shown around all expressions to further indicate their order of evaluation.

Default: NOFEEDBACK

FLOW<=*n* <*m*>>|NOFLOW
specifies that character columns longer than *n* are flowed to multiple lines. PROC SQL sets the column width at *n* and specifies that character columns longer than *n* are flowed to multiple lines. When you specify FLOW=*n m*, PROC SQL floats the width of the columns between these limits to achieve a balanced layout. FLOW is equivalent to FLOW=12 200.

Default: NOFLOW

INOBS=*n*
restricts the number of rows (observations) that PROC SQL retrieves from any single source.

Tip: This option is useful for debugging queries on large tables.

LOOPS=*n*
restricts PROC SQL to *n* iterations through its inner loop. You use the number of iterations reported in the SQLOOPS macro variable (after each SQL statement is executed) to discover the number of loops. Set a limit to prevent queries from consuming excessive computer resources. For example, joining three large tables without meeting the join-matching conditions could create a huge internal table that would be inefficient to execute.

See also: "Using Macro Variables Set by PROC SQL" on page 1096

NODOUBLE
See DOUBLE|NODOUBLE on page 1028.

NOERRORSTOP
See ERRORSTOP|NOERRORSTOP on page 1028.

NOEXEC
See EXEC|NOEXEC on page 1028.

NOFEEDBACK
See FEEDBACK|NOFEEDBACK on page 1028.

NOFLOW
See FLOW|NOFLOW on page 1029.

NONUMBER
See NUMBER|NONUMBER on page 1029.

NOPRINT
See PRINT|NOPRINT on page 1030.

NOPROMPT
See PROMPT|NOPROMPT on page 1030.

NOSORTMSG
See SORTMSG|NOSORTMSG on page 1030.

NOSTIMER
See STIMER|NOSTIMER on page 1030.

NUMBER|NONUMBER
specifies whether the SELECT statement includes a column called ROW, which is the row (or observation) number of the data as they are retrieved.

Default: NONUMBER

Featured in: Example 4 on page 1106

OUTOBS=*n*
restricts the number of rows (observations) in the output. For example, if you specify OUTOBS=10 and insert values into a table using a query-expression, the SQL procedure inserts a maximum of 10 rows. Likewise, OUTOBS=10 limits the output to 10 rows.

PRINT|NOPRINT
specifies whether the output from a SELECT statement is printed.

Default: PRINT

Tip: NOPRINT is useful when you are selecting values from a table into macro variables and do not want anything to be displayed.

PROMPT|NOPROMPT
modifies the effect of the INOBS=, OUTOBS=, and LOOPS= options. If you specify the PROMPT option and reach the limit specified by INOBS=, OUTOBS=, or LOOPS=, PROC SQL prompts you to stop or continue. The prompting repeats if the same limit is reached again.

Default: NOPROMPT

SORTMSG|NOSORTMSG
Certain operations, such as ORDER BY, may sort tables internally using PROC SORT. Specifying SORTMSG requests information from PROC SORT about the sort and displays the information in the log.

Default: NOSORTMSG

SORTSEQ=*sort-table*
specifies the collating sequence to use when a query contains an ORDER BY clause. Use this option only if you want a collating sequence other than your system's or installation's default collating sequence.

See also: SORTSEQ= option in *SAS Language Reference: Dictionary*.

STIMER|NOSTIMER
specifies whether PROC SQL writes timing information to the SAS log for each statement, rather than as a cumulative value for the entire procedure. For this option to work, you must also specify the SAS system option STIMER. Some operating environments require that you specify this system option when you invoke SAS. If you use the system option alone, you receive timing information for the entire SQL procedure, not on a statement-by-statement basis.

Default: NOSTIMER

UNDO_POLICY=NONE|OPTIONAL|REQUIRED
specifies how PROC SQL handles updated data if errors occur while you are updating data. You can use UNDO_POLICY= to control whether your changes will be permanent:

NONE
keeps any updates or inserts.

OPTIONAL
reverses any updates or inserts that it can reverse reliably.

REQUIRED
undoes all inserts or updates that have been done to the point of the error. In some cases, the UNDO operation cannot be done reliably. For example, when a program uses a SAS/ACCESS view, it may not be able to reverse the effects of the

INSERT and UPDATE statements without reversing the effects of other changes at the same time. In that case, PROC SQL issues an error message and does not execute the statement. Also, when a SAS data set is accessed through a SAS/SHARE server and is opened with the data set option CNTLLEV=RECORD, you cannot reliably reverse your changes.

This option may enable other users to update newly inserted rows. If an error occurs during the insert, PROC SQL can delete a record that another user updated. In that case, the statement is not executed, and an error message is issued.

Default: REQUIRED

Note: Options can be added, removed, or changed between PROC SQL statements with the RESET statement. △

ALTER TABLE Statement

Adds columns to, drops columns from, and changes column attributes in an existing table. Adds, modifies, and drops integrity constraints from an existing table.

Restriction: You cannot use any type of view in an ALTER TABLE statement.

Restriction: You cannot use ALTER TABLE on a table that is accessed via an engine that does not support UPDATE processing.

Featured in: Example 3 on page 1104

ALTER TABLE *table-name*
<*constraint-clause*> <, *constraint-clause*>...>;

<**ADD** column-definition <,column-definition>…>

<**MODIFY** column-definition
<,column-definition>…>

<**DROP** *column* <,*column*>…>;

where each *constraint-clause* is one of the following:

ADD <CONSTRAINT *constraint-name*> *constraint*

DROP CONSTRAINT *constraint-name*

DROP FOREIGN KEY *constraint-name* [Note: This is a DB2 extension.]

DROP PRIMARY KEY [Note: This is a DB2 extension.]

where *constraint* can be one of the following:

NOT NULL (*column*)

CHECK (*WHERE-clause*)

PRIMARY KEY (*columns*)

DISTINCT (*columns*)

UNIQUE (*columns*)

FOREIGN KEY (*columns*)
REFERENCES *table-name*
<ON DELETE *referential-action* > <ON UPDATE *referential-action*>

Arguments

column
names a column in *table-name*.

column-definition
See “column-definition” on page 1059.

constraint-name
specifies the name for the constraint being specified.

referential-action
specifies the type of action to be performed on all matching foreign key values.

RESTRICT
occurs only if there are matching foreign key values. This is the default referential action.

SET NULL
sets all matching foreign key values to NULL.

table-name
refers to the name of table containing the primary key referenced by the foreign key.

WHERE-clause
specifies a SAS WHERE-clause.

Specifying Initial Values of New Columns

When the ALTER TABLE statement adds a column to the table, it initializes the column’s values to missing in all rows of the table. Use the UPDATE statement to add values to the new column(s).

Changing Column Attributes

If a column is already in the table, you can change the following column attributes using the MODIFY clause: length, informat, format, and label. The values in a table are either truncated or padded with blanks (if character data) as necessary to meet the specified length attribute.

You cannot change a character column to numeric and vice versa. To change a column’s data type, drop the column and then add it (and its data) again, or use the DATA step.

Note: You cannot change the length of a numeric column with the ALTER TABLE statement. Use the DATA step instead. △

Renaming Columns

To change a column’s name, you must use the SAS data set option RENAME=. You cannot change this attribute with the ALTER TABLE statement. RENAME= is described in the section on SAS data set options in *SAS Language Reference: Dictionary*.

Indexes on Altered Columns

When you alter the attributes of a column and an index has been defined for that column, the values in the altered column continue to have the index defined for them. If you drop a column with the ALTER TABLE statement, all the indexes (simple and

composite) in which the column participates are also dropped. See "CREATE INDEX Statement" on page 1033 for more information on creating and using indexes.

Integrity Constraints

Use ALTER TABLE to modify integrity constraints for existing tables. Use the CREATE TABLE statement to attach integrity constraints to new tables. For more information on integrity constraints, see the section on SAS files in *SAS Language Reference: Concepts*.

CONNECT Statement

Establishes a connection with a DBMS that is supported by SAS/ACCESS software.

Requirement: SAS/ACCESS software is required. For more information on this statement, refer to your SAS/ACCESS documentation.

See also: "Connecting to a DBMS Using the SQL Procedure Pass-Through Facility" on page 1095

CONNECT TO *dbms-name* <AS *alias*> <(<*connect-statement-arguments*> <*database-connection-arguments*>)>;

Arguments

alias
specifies an alias that has 1 to 32 characters. The keyword AS must precede *alias*. Some DBMSs allow more than one connection. The optional AS clause enables you to name the connections so that you can refer to them later.

connect-statement-arguments
specifies arguments that indicate whether you can make multiple connections, shared or unique connections, and so on to the database. These arguments are optional, but if they are included, they must be enclosed in parentheses.

database-connection-arguments
specifies the DBMS-specific arguments that are needed by PROC SQL to connect to the DBMS. These arguments are optional for most databases, but if they are included, they must be enclosed in parentheses.

dbms-name
identifies the DBMS that you want to connect to (for example, ORACLE or DB2).

CREATE INDEX Statement

Creates indexes on columns in tables.

Restriction: You cannot use CREATE INDEX on a table accessed via an engine that does not support UPDATE processing.

CREATE <UNIQUE> INDEX *index-name*
ON *table-name (column <, column>…)*;

Arguments

column
specifies a column in *table-name*.

index-name
names the index that you are creating. If you are creating an index on one column only, *index-name* must be the same as *column*. If you are creating an index on more than one column, *index-name* cannot be the same as any column in the table.

table-name
specifies a PROC SQL table.

Indexes in PROC SQL

An *index* stores both the values of a table's columns and a system of directions that enable access to rows in that table by index value. Defining an index on a column or set of columns enables SAS, under certain circumstances, to locate rows in a table more quickly and efficiently. Indexes enable PROC SQL to execute the following classes of queries more efficiently:

- comparisons against a column that is indexed
- an IN subquery where the column in the inner subquery is indexed
- correlated subqueries, where the column being compared with the correlated reference is indexed
- join-queries, where the join-expression is an equals comparison and all the columns in the join-expression are indexed in one of the tables being joined.

SAS maintains indexes for all changes to the table, whether the changes originate from PROC SQL or from some other source. Therefore, if you alter a column's definition or update its values, the same index continues to be defined for it. However, if an indexed column in a table is dropped, the index on it is also dropped.

You can create simple or composite indexes. A *simple index* is created on one column in a table. A simple index must have the same name as that column. A *composite index* is one index name that is defined for two or more columns. The columns can be specified in any order, and they can have different data types. A composite index name cannot match the name of any column in the table. If you drop a composite index, the index is dropped for all the columns named in that composite index.

UNIQUE Keyword

The UNIQUE keyword causes the SAS System to reject any change to a table that would cause more than one row to have the same index value. Unique indexes guarantee that data in one column, or in a composite group of columns, remain unique for every row in a table. For this reason, a unique index cannot be defined for a column that includes NULL or missing values.

Managing Indexes

You can use the CONTENTS statement in the DATASETS procedure to display a table's index names and the columns for which they are defined. You can also use the

DICTIONARY tables INDEXES, TABLES, and COLUMNS to list information about indexes. See "DICTIONARY tables" on page 1062.

See the section on SAS files in *SAS Language Reference: Dictionary* for a further description of when to use indexes and how they affect SAS statements that handle BY-group processing.

CREATE TABLE Statement

Creates PROC SQL tables.

Featured in: Example 1 on page 1101 and Example 2 on page 1103

❶**CREATE TABLE** *table-name* (column-definition <,column-definition>…);

(column-specification ,…<*constraint-specification*> ,…) ;

where *column-specification* is

column-definition <*column-attribute*>

where *constraint-specification* is

CONSTRAINT *constraint-name constraint*

column-attribute is one of the following:

UNIQUE

DISTINCT [Note: This is a DB2 extension. DISTINCT is the same as UNIQUE.]

NOT NULL

CHECK (*WHERE-clause*)

PRIMARY KEY

REFERENCES *table-name*
<**ON DELETE** *referential-action* > <**ON UPDATE** *referential-action* >

constraint is one of the following:

NOT NULL (*column*)

CHECK (*WHERE-clause*)

PRIMARY KEY (*columns*)

DISTINCT (*columns*)

UNIQUE (*columns*)

FOREIGN KEY (*columns*)
REFERENCES *table-name*
<ON DELETE *referential-action*> <ON UPDATE *referential-action*>

❷**CREATE TABLE***table-name* **LIKE** *table-name*;

❸**CREATE TABLE** *table-name* **AS** query-expression
<**ORDER BY** *order-by-item* <,*order-by-item*>…>;

Arguments

column-definition
See “column-definition” on page 1059.

constraint-name
is the name for the constraint being specified.

order-by-item
See ORDER BY Clause on page 1053.

query-expression
See “query-expression” on page 1075.

referential-action
specifies the type of action to be performed on all matching foreign key values.

RESTRICT
occurs only if there are matching foreign key values. This is the default referential action.

SET NULL
sets all matching foreign key values to NULL.

table-name
is the name of the table containing the primary key referenced by the foreign key.

WHERE clause
specifies a SAS WHERE-clause.

Creating a Table without Rows

1 The first form of the CREATE TABLE statement creates tables that automatically map SQL data types to those supported by the SAS System. Use this form when you want to create a new table with columns that are not present in existing tables. It is also useful if you are running SQL statements from an SQL application in another SQL-based database.

2 The second form uses a LIKE clause to create a table that has the same column names and column attributes as another table. To drop any columns in the new table, you can specify the DROP= data set option in the CREATE TABLE statement. The specified columns are dropped when the table is created. Indexes are not copied to the new table.

Both of these forms create a table without rows. You can use an INSERT statement to add rows. Use an ALTER statement to modify column attributes or to add or drop columns.

Creating a Table from a Query Expression

3 The third form of the CREATE TABLE statement stores the results of any query-expression in a table and does not display the output. It is a convenient way to create temporary tables that are subsets or supersets of other tables.

When you use this form, a table is physically created as the statement is executed. The newly created table does not reflect subsequent changes in the underlying tables (in the query-expression). If you want to continually access the

most current data, create a view from the query expression instead of a table. See "CREATE VIEW Statement" on page 1037.

Integrity Constraints

You can attach integrity constraints when you create a new table. To modify integrity constraints, use the ALTER TABLE statement. For more information on integrity constraints, see the section on SAS files in *SAS Language Reference: Concepts*.

CREATE VIEW Statement

Creates a PROC SQL view from a query-expression.

See also: "What Are Views?" on page 1023
Featured in: Example 8 on page 1116

CREATE VIEW *proc-sql-view* **AS** query-expression
<**ORDER BY** *order-by-item* <,*order-by-item*>...>
<**USING** *statement*<, *libname-clause*> ... > ;

where each *libname-clause* is one of the following:

LIBNAME *libref* <*engine*> *'SAS-data-library'* <*option(s)*> <*engine-host-option(s)*>

LIBNAME *libref SAS*|*ACCESS-engine-name* <*SAS*|*ACCESS-engine-connection-option(s)*> <*SAS*|*ACCESS-engine-LIBNAME-option(s)*>

Arguments

order-by-item
See ORDER BY Clause on page 1053.

query-expression
See "query-expression" on page 1075.

proc-sql-view
specifies the name for the PROC SQL view that you are creating. See "What Are Views?" on page 1023 for a definition of a PROC SQL view.

Sorting Data Retrieved by Views

PROC SQL allows you to specify the ORDER BY clause in the CREATE VIEW statement. Every time a view is accessed, its data are sorted and displayed as specified by the ORDER BY clause. This sorting on every access has certain performance costs, especially if the view's underlying tables are large. It is more efficient to omit the ORDER BY clause when you are creating the view and specify it as needed when you reference the view in queries.

Note: If you specify the NUMBER option in the PROC SQL statement when you create your view, the ROW column appears in the output. However, you cannot order by

the ROW column in subsequent queries. See the description of the NUMBER option on page 1030. △

Librefs and Stored Views

You can refer to a table name alone (without the libref) in the FROM clause of a CREATE VIEW statement if the table and view reside in the same SAS data library, as in this example:

```
create view proclib.view1 as
   select *
      from invoice
      where invqty>10;
```

In this view, VIEW1 and INVOICE are stored permanently in the SAS data library referenced by PROCLIB. Specifying a libref for INVOICE is optional.

Updating Views

You can update a view's underlying data with some restrictions. See "Updating PROC SQL and SAS/ACCESS Views" on page 1097.

Embedded LIBNAME Statements

The USING clause allows you to store DBMS connection information in a view by *embedding* the SAS/ACCESS LIBNAME statement inside the view. When PROC SQL executes the view, the stored query assigns the libref and establishes the DBMS connection using the information in the LIBNAME statement. The scope of the libref is local to the view, and will not conflict with any identically named librefs in the SAS session. When the query finishes, the connection to the DBMS is terminated and the libref is deassigned.

The USING clause must be the last clause in the SELECT statement. Multiple LIBNAME statements can be specified, separated by commas. In the following example, a connection is made and the libref ACCREC is assigned to an ORACLE database.

```
create view proclib.view1 as
   select *
      from accrec.invoices as invoices
      using libname accrec oracle
         user=username pass=password
         path='dbms-path';
```

For more information on the SAS/ACCESS LIBNAME statement, see the SAS/ACCESS documentation for your DBMS.

You can also embed a SAS LIBNAME statement in a view with the USING clause. This enables you to store SAS libref information in the view. Just as in the embedded SAS/ACCESS LIBNAME statement, the scope of the libref is local to the view, and it will not conflict with an identically named libref in the SAS session.

```
create view work.tableview as
   select * from proclib.invoices
      using libname proclib 'sas-data-library';
```

DELETE Statement

Removes one or more rows from a table or view that is specified in the FROM clause.

Restriction: You cannot use DELETE FROM on a table accessed via an engine that does not support UPDATE processing.

Featured in: Example 5 on page 1108

DELETE
FROM *table-name*|*sas/access-view*|*proc-sql-view* <**AS** *alias*>
<**WHERE**sql-expression>;

Arguments

alias
assigns an alias to *table-name*, *sas/access-view*, or *proc-sql-view*.

sas/access-view
specifies a SAS/ACCESS view that you are deleting rows from.

proc-sql-view
specifies a PROC SQL view that you are deleting rows from.

sql-expression
See "sql-expression" on page 1081.

table-name
specifies the table that you are deleting rows from.

Deleting Rows Through Views

You can delete one or more rows from a view's underlying table, with some restrictions. See "Updating PROC SQL and SAS/ACCESS Views" on page 1097.

CAUTION:
If you omit a WHERE clause, the DELETE statement deletes all the rows from the specified table or the table described by a view. △

DESCRIBE Statement

Displays a PROC SQL definition in the SAS log.

Restriction: PROC SQL views are the only type of view allowed in a DESCRIBE VIEW statement.

Featured in: Example 6 on page 1111

DESCRIBE TABLE *table-name* <,*table-name*>... ;

DESCRIBE VIEW *proc-sql-view* <,*proc-sql-view*>... ;

DESCRIBE TABLE CONSTRAINTS *table-name* <, *table-name*>... ;

Arguments

table-name
specifies a PROC SQL table.

proc-sql-view
specifies a PROC SQL view.

Details

- The DESCRIBE TABLE statement writes a CREATE TABLE statement to the SAS log for the table specified in the DESCRIBE TABLE statement, regardless of how the table was originally created (for example, with a DATA step). If applicable, SAS data set options are included with the table definition. If indexes are defined on columns in the table, CREATE INDEX statements for those indexes are also written to the SAS log.

  When you are transferring a table to a DBMS that is supported by SAS/ACCESS software, it is helpful to know how it is defined. To find out more information on a table, use the FEEDBACK option or the CONTENTS statement in the DATASETS procedure.
- The DESCRIBE VIEW statement writes a view definition to the SAS log. If you use a PROC SQL view in the DESCRIBE VIEW statement that is based on or derived from another view, you may want to use the FEEDBACK option in the PROC SQL statement. This option displays in the SAS log how the underlying view is defined and expands any expressions that are used in this view definition. The CONTENTS statement in DATASETS procedure can also be used with a view to find out more information.
- The DESCRIBE TABLE CONSTRAINTS statement lists the integrity constraints that are defined for the specified table(s).

DISCONNECT Statement

Ends the connection with a DBMS that is supported by a SAS/ACCESS interface.

Requirement: SAS/ACCESS software is required. For more information on this statement, refer to your SAS/ACCESS documentation.

See also: "Connecting to a DBMS Using the SQL Procedure Pass-Through Facility" on page 1095

DISCONNECT FROM *dbms-name*|*alias*;

Arguments

alias
specifies the alias that is defined in the CONNECT statement.

dbms-name
specifies the DBMS from which you want to end the connection (for example, DB2 or ORACLE). The name you specify should match the name that is specified in the CONNECT statement.

Details

- An implicit COMMIT is performed before the DISCONNECT statement ends the DBMS connection. If a DISCONNECT statement is not submitted, implicit DISCONNECT and COMMIT actions are performed and the connection to the DBMS is broken when PROC SQL terminates.
- PROC SQL continues executing until you submit a QUIT statement, another SAS procedure, or a DATA step.

DROP Statement

Deletes tables, views, or indexes.

Restriction: You cannot use DROP TABLE or DROP INDEX on a table accessed via an engine that does not support UPDATE processing.

DROP TABLE *table-name* <,*table-name*>...;

DROP VIEW *view-name* <,*view-name*>...;

DROP INDEX *index-name* <,*index-name*>...
FROM *table-name*;

Arguments

index-name
specifies an index that exists on *table-name*.

table-name
specifies a PROC SQL table.

view-name
specifies a SAS data view of any type: PROC SQL view, SAS/ACCESS view, or DATA step view.

Details

- If you drop a table that is referenced in a view definition and try to execute the view, an error message is written to the SAS log stating that the table does not

exist. Therefore, remove references in queries and views to any table(s) and view(s) that you drop.

- If you drop a table with indexed columns, all the indexes are automatically dropped. If you drop a composite index, the index is dropped for all the columns that are named in that index.
- You cannot use the DROP statement to drop a table or view in an external database that is described by a SAS/ACCESS view.

EXECUTE Statement

Sends a DBMS-specific SQL statement to a DBMS that is supported by a SAS/ACCESS interface.

Requirement: SAS/ACCESS software is required. For more information on this statement, refer to your SAS/ACCESS documentation.

See also: "Connecting to a DBMS Using the SQL Procedure Pass-Through Facility" on page 1095 and the SQL documentation for your DBMS.

EXECUTE (*dbms-SQL-statement*)
BY *dbms-name*|*alias*;

Arguments

alias
specifies an optional alias that is defined in the CONNECT statement. Note that *alias* must be preceded by the keyword BY.

dbms-name
identifies the DBMS to which you want to direct the DBMS statement (for example, ORACLE or DB2).

dbms-SQL-statement
is any DBMS-specific SQL statement, except the SELECT statement, that can be executed by the DBMS-specific dynamic SQL.

Details

- If your DBMS supports multiple connections, you can use the alias that is defined in the CONNECT statement. This alias directs the EXECUTE statements to a specific DBMS connection.
- Any return code or message that is generated by the DBMS is available in the macro variables SQLXRC and SQLXMSG after the statement completes.

INSERT Statement

Adds rows to a new or existing table or view.

Restriction: You cannot use INSERT INTO on a table accessed via an engine that does not support UPDATE processing.

Featured in: Example 1 on page 1101

❶**INSERT INTO***table-name*|*sas*/*access-view*|*proc-sql-view*
<(*column*<,*column*>…)><,*user-name*>...;
SET *column*=sql-expression
<,*column*=sql-expression>…
<**SET** *column*=sql-expression
<,*column*=sql-expression>…>;

❷**INSERT INTO** *table-name*|*sas*/*access-view*|*proc-sql-view* <(*column*<,*column*>…)>
VALUES (*value* <, *value*>…)
<**VALUES** (*value* <, *value*>…)>…;

❸**INSERT INTO** *table-name*|*sas*/*access-view*|*proc-sql-view*
<(*column*<,*column*>…)> query-expression;

Arguments

column
specifies the column into which you are inserting rows.

sas/access-view
specifies a SAS/ACCESS view into which you are inserting rows.

proc-sql-view
specifies a PROC SQL view into which you are inserting rows.

sql-expression
See “sql-expression” on page 1081.

table-name
specifies a PROC SQL table into which you are inserting rows.

value
is a data value.

Methods for Inserting Values

1 The first form of the INSERT statement uses the SET clause, which specifies or alters the values of a column. You can use more than one SET clause per INSERT statement, and each SET clause can set the values in more than one column. Multiple SET clauses are not separated by commas. If you specify an optional list of columns, you can set a value only for a column that is specified in the list of columns to be inserted.

2 The second form of the INSERT statement uses the VALUES clause. This clause can be used to insert lists of values into a table. You can either give a value for

each column in the table or give values just for the columns specified in the list of column names. One row is inserted for each VALUES clause. Multiple VALUES clauses are not separated by commas. The order of the values in the VALUES clause matches the order of the column names in the INSERT column list or, if no list was specified, the order of the columns in the table.

3 The third form of the INSERT statement inserts the results of a query-expression into a table. The order of the values in the query-expression matches the order of the column names in the INSERT column list or, if no list was specified, the order of the columns in the table.

Note: If the INSERT statement includes an optional list of column names, only those columns are given values by the statement. Columns that are in the table but not listed are given missing values. △

Inserting Rows through Views

You can insert one or more rows into a table through a view, with some restrictions. See "Updating PROC SQL and SAS/ACCESS Views" on page 1097.

Adding Values to an Indexed Column

If an index is defined on a column and you insert a new row into the table, that value is added to the index. You can display information about indexes with

- the CONTENTS statement in the DATASETS procedure. See "CONTENTS Statement" on page 346.
- the DICTIONARY.INDEXES table. See "DICTIONARY tables" on page 1062 for more information.

For more information on creating and using indexes, see "CREATE INDEX Statement" on page 1033.

RESET Statement

Resets PROC SQL options without restarting the procedure.

Featured in: Example 5 on page 1108

RESET <*option(s)*>;

The RESET statement enables you to add, drop, or change the options in PROC SQL without restarting the procedure. See "PROC SQL Statement" on page 1027 for a description of the options.

SELECT Statement

Selects columns and rows of data from tables and views.

See also: "table-expression" on page 1094, "query-expression" on page 1075

SELECT <DISTINCT> *object-item* <,*object-item*>...
<**INTO** :*macro-variable-specification*
<, :*macro-variable-specification*>...>
FROM *from-list*
<**WHERE** sql-expression>
<**GROUP BY** *group-by-item*
<, *group-by-item*>...>
<**HAVING** sql-expression>
<**ORDER BY** *order-by-item*
<,*order-by-item*>...>;

SELECT Clause

Lists the columns that will appear in the output.

See Also: "column-definition" on page 1059

Featured in: Example 1 on page 1101 and Example 2 on page 1103

SELECT <DISTINCT> *object-item* <,*object-item*>...

- *object-item* is one of the following:

  *

  case-expression <**AS** *alias*>

  column-name <**AS** *alias*>
  <column-modifier <column-modifier>...>

  sql-expression <**AS** *alias*>
  <column-modifier <column-modifier>...>

  table-name.*

  table-alias.*

  view-name.*

  view-alias.*

Arguments

case-expression
See "CASE expression" on page 1058.

column-modifier
See "column-modifier" on page 1060.

column-name
See "column-name" on page 1061.

DISTINCT
eliminates duplicate rows.

Featured in: Example 13 on page 1126

sql-expression
See "sql-expression" on page 1081.

table-alias
is an alias for a PROC SQL table.

table-name
specifies a PROC SQL table.

view-name
specifies any type of SAS data view.

view-alias
specifies the alias for any type of SAS data view.

Asterisk(*) Notation

The asterisk (*) represents all columns of the table(s) listed in the FROM clause. When an asterisk is not prefixed with a table name, all the columns from all tables in the FROM clause are included; when it is prefixed (for example, *table-name.** or *table-alias.**), all the columns from that table only are included.

Column Aliases

A column alias is a temporary, alternate name for a column. Aliases are specified in the SELECT clause to name or rename columns so that the result table is clearer or easier to read. Aliases are often used to name a column that is the result of an arithmetic expression or summary function. An alias is one word only. If you need a longer column name, use the LABEL= column-modifier, as described in "column-modifier" on page 1060. The keyword AS is required with a column alias to distinguish the alias from other column names in the SELECT clause.

Column aliases are optional, and each column name in the SELECT clause can have an alias. After you assign an alias to a column, you can use the alias to refer to that column in other clauses.

If you use a column alias when creating a PROC SQL view, the alias becomes the permanent name of the column for each execution of the view.

INTO Clause

Stores the value of one or more columns for use later in another PROC SQL query or SAS statement.

Restriction: An INTO clause cannot be used in a CREATE TABLE statement.

See also: "Using Macro Variables Set by PROC SQL" on page 1096

INTO :*macro-variable-specification*
<, :*macro-variable-specification*>…

- :*macro-variable-specification* is one of the following:

  :*macro-variable* <SEPARATED BY '*character*' <NOTRIM>>;

:*macro-variable-1* – :*macro-variable-n* <NOTRIM>;

Arguments

macro-variable
specifies a SAS macro variable that stores the values of the rows that are returned.

NOTRIM
protects the leading and trailing blanks from being deleted from the macro variable value when the macro variables are created.

SEPARATED BY '*character*'
specifies a character that separates the values of the rows.

Details

- Use the INTO clause only in the outer query of a SELECT statement and not in a subquery.
- You can put multiple rows of the output into macro variables. You can check the PROC SQL macro variable SQLOBS to see the number of rows produced by a query-expression. See "Using Macro Variables Set by PROC SQL" on page 1096 for more information on SQLOBS.

Examples

These examples use the PROCLIB.HOUSES table:

```
                              The SAS System                              1

                          Style       SqFeet
                          ------------------
                          CONDO          900
                          CONDO         1000
                          RANCH         1200
                          RANCH         1400
                          SPLIT         1600
                          SPLIT         1800
                          TWOSTORY      2100
                          TWOSTORY      3000
```

With the *macro-variable-specification*, you can do the following:

- You can create macro variables based on the first row of the result.

```
proc sql noprint;
   select style, sqfeet
      into :style, :sqfeet
      from proclib.houses;

%put &style &sqfeet;
```

The results are written to the SAS log:

```
1   proc sql noprint;
2      select style, sqfeet
3         into :style, :sqfeet
4         from proclib.houses;
5
6   %put &style &sqfeet;
CONDO         900
```

- You can create one new macro variable per row in the result of the SELECT statement. This example shows how you can request more values for one column than for another. The hyphen (-) is used in the INTO clause to imply a range of macro variables. You can use either the keywords THROUGH or THRU instead of a hyphen.

  The following PROC SQL step puts the values from the first four rows of the PROCLIB.HOUSES table into macro variables:

```
proc sql noprint;
select distinct Style, SqFeet
   into :style1 - :style3, :sqfeet1 - :sqfeet4
   from proclib.houses;

%put &style1 &sqfeet1;
%put &style2 &sqfeet2;
%put &style3 &sqfeet3;
%put &sqfeet4;
```

  The %PUT statements write the results to the SAS log:

```
1   proc sql noprint;
2   select distinct style, sqfeet
3      into :style1 - :style3, :sqfeet1 - :sqfeet4
4      from proclib.houses;
5
6   %put &style1 &sqfeet1;
CONDO 900
7   %put &style2 &sqfeet2;
CONDO 1000
8   %put &style3 &sqfeet3;
CONDO 1200
9   %put &sqfeet4;
1400
```

- You can concatenate the values of one column into one macro variable. This form is useful for building up a list of variables or constants.

```
proc sql;
   select distinct style
      into :s1 separated by ','
      from proclib.houses;

%put &s1;
```

The results are written to the SAS log:

```
3    proc sql;
4       select distinct style
5          into :s1 separated by ','
6          from proclib.houses;
7
8   %put &s1

CONDO,RANCH,SPLIT,TWOSTORY
```

- The leading and trailing blanks are trimmed from the values before the macro variables are created. If you do not want the blanks to be trimmed, add NOTRIM, as shown in the following example:

```
proc sql noprint;
   select style, sqfeet
      into :style1 - :style4 notrim,
           :sqfeet separated by ',' notrim
      from proclib.houses;

%put *&style1* *&sqfeet*;
%put *&style2* *&sqfeet*;
%put *&style3* *&sqfeet*;
%put *&style4* *&sqfeet*;
```

The results are written to the SAS log, as shown in Output 34.1 on page 1049.

Output 34.1 Macro Variable Values

```
3   proc sql noprint;
4      select style, sqfeet
5         into :style1 - :style4 notrim,
6              :sqfeet separated by ',' notrim
7         from proclib.houses;
8
9   %put *&style1* *&sqfeet*;
*CONDO   * *     900,    1000,    1200,    1400,    1600,    1800,    2100,
 3000*
10  %put *&style2* *&sqfeet*;
*CONDO   * *     900,    1000,    1200,    1400,    1600,    1800,    2100,
 3000*
11  %put *&style3* *&sqfeet*;
*RANCH   * *     900,    1000,    1200,    1400,    1600,    1800,    2100,
 3000*
12  %put *&style4* *&sqfeet*;
*RANCH   * *     900,    1000,    1200,    1400,    1600,    1800,    2100,
 3000*
```

FROM Clause

Specifies source tables or views.

Featured in: Example 1 on page 1101, Example 4 on page 1106, Example 9 on page 1118, and Example 10 on page 1121

FROM *from-list*

□ *from-list* is one of the following:

table-name <<**AS**> *alias*>

view-name <<**AS**> *alias*>

joined-table

(query-expression) <<AS> alias
<(*column* <,*column*>...)>>

CONNECTION TO

Arguments

column
names the column that appears in the output. The column names that you specify are matched by position to the columns in the output.

CONNECTION TO
See "CONNECTION TO" on page 1062.

joined-table
See "joined-table" on page 1068.

query-expression
See "query-expression" on page 1075.

table-name
specifies a PROC SQL table.

view-name
specifies any type of SAS data view.

Table Aliases

A table alias is a temporary, alternate name for a table that is specified in the FROM clause. Table aliases are prefixed to column names to distinguish between columns that are common to multiple tables. Table aliases are always required when joining a table with itself. Column names in other kinds of joins must be prefixed with table aliases or table names unless the column names are unique to those tables.

The optional keyword AS is often used to distinguish a table alias from other table names.

In-Line Views

The FROM clause can itself contain a query-expression that takes an optional table alias. This kind of nested query-expression is called an *in-line view*. An in-line view is any query-expression that would be valid in a CREATE VIEW statement. PROC SQL can support many levels of nesting, but it is limited to 32 tables in any one query. The 32–table limit includes underlying tables that may contribute to views that are specified in the FROM clause.

An in-line view saves you a programming step. Rather than creating a view and referring to it in another query, you can specify the view *in-line* in the FROM clause.

Characteristics of in-line views include the following:

□ An in-line view is not assigned a permanent name, although it can take an alias.

- An in-line view can be referred to only in the query in which it is defined. It cannot be referenced in another query.
- You cannot use an ORDER BY clause in an in-line view.
- The names of columns in an in-line view can be assigned in the object-item list of that view or with a parenthesized list of names following the alias. This syntax can be useful for renaming columns. See Example 10 on page 1121 for an example.

WHERE Clause

Subsets the output based on specified conditions.

Featured in: Example 4 on page 1106 and Example 9 on page 1118

WHERE sql-expression

Argument

sql-expression
See "sql-expression" on page 1081.

Details

- When a condition is met (that is, the condition resolves to true), those rows are displayed in the result table; otherwise, no rows are displayed.
- You cannot use summary functions that specify only one column. For example:

```
where max(measure1) > 50;
```

  However, this WHERE clause will work:

```
where max(measure1,measure2) > 50;
```

Writing Efficient WHERE Clauses

Here are some guidelines for writing efficient WHERE clauses that enable PROC SQL to use indexes effectively:

- Avoid using LIKE predicates that begin with % or _:

```
/* inefficient:*/ where country like '%INA'
/* efficient: */  where country like 'A%INA'
```

- Avoid using arithmetic expressions in a predicate:

```
/* inefficient:*/ where salary>12*4000
/* efficient: */  where salary>48000
```

- First put the expression that returns the fewest number of rows. In the following query, there are fewer rows where miles>3800 than there are where boarded>100.

```
where miles>3800 and boarded>100
```

GROUP BY Clause

Specifies how to group the data for summarizing.

Featured in: Example 8 on page 1116 and Example 12 on page 1124

GROUP BY *group-by-item <,group-by-item>…*

- *group-by-item* is one of the following:

  integer

  column-name

  sql-expression

Arguments

integer
equates to a column's position.

column-name
See "column-name" on page 1061.

sql-expression
See "sql-expression" on page 1081.

Details

- You can specify more than one *group-by-item* to get more detailed reports. Both the grouping of multiple items and the BY statement of a PROC step are evaluated in similiar ways. If more than one *group-by-item* is specified, the first one determines the major grouping.
- Integers can be substituted for column names (that is, SELECT object-items) in the GROUP BY clause. For example, if the *group-by-item* is 2, the results are grouped by the values in the second column of the SELECT clause list. Using integers can shorten your coding and enable you to group by the value of an unnamed expression in the SELECT list.
- The data do not have to be sorted in the order of the group-by values because PROC SQL handles sorting automatically. You can use the ORDER BY clause to specify the order in which rows are displayed in the result table.
- If you specify a GROUP BY clause in a query that does not contain a summary function, your clause is transformed into an ORDER BY clause and a message to that effect is written to the SAS log.
- A *group-by-item* cannot be a summary function. For example, the following GROUP BY clause is not valid:

```
group by sum(x)
```

HAVING Clause

Subsets grouped data based on specified conditions.

Featured in: Example 8 on page 1116 and Example 12 on page 1124

HAVING sql-expression

Argument

sql-expression
See "sql-expression" on page 1081.

Subsetting Grouped Data

The HAVING clause is used with at least one summary function and an optional GROUP BY clause to summarize groups of data in a table. A HAVING clause is any valid SQL expression that is evaluated as either true or false for each group in a query. Or, if the query involves remerged data, the HAVING expression is evaluated for each row that participates in each group. The query must include one or more summary functions.

Typically, the GROUP BY clause is used with the HAVING expression and defines the group(s) to be evaluated. If you omit the GROUP BY clause, the summary function and the HAVING clause treat the table as one group.

The following PROC SQL step uses the PROCLIB.PAYROLL table (shown in Example 2 on page 1103) and groups the rows by SEX to determine the oldest employee of each sex. In SAS, dates are stored as integers. The lower the birthdate as an integer, the greater the age. The expression **`birth=min(birth)`**is evaluated for each row in the table. When the minimum birthdate is found, the expression becomes true and the row is included in the output.

```
proc sql;
   title 'Oldest Employee of Each Gender';
   select *
      from proclib.payroll
      group by sex
      having birth=min(birth);
```

Note: This query involves remerged data because the values returned by a summary function are compared to values of a column that is not in the GROUP BY clause. See "Remerging Data" on page 1090 for more information about summary functions and remerging data. △

ORDER BY Clause

Specifies the order in which rows are displayed in a result table.

See also: "query-expression" on page 1075
Featured in: Example 11 on page 1122

ORDER BY *order-by-item* <,*order-by-item*>…;

- *order-by-item* is one of the following:

  integer <ASC|DESC>

  column-name <ASC|DESC>

  sql-expression <ASC|DESC>

Arguments

ASC
orders the data in ascending order. This is the default order.

column-name
See “column-name” on page 1061.

DESC
orders the data in descending order.

integer
equates to a column’s position.

sql-expression
See “sql-expression” on page 1081.

Details

- The ORDER BY clause sorts the result of a query expression according to the order specified in that query. When this clause is used, the default ordering sequence is ascending, from the lowest value to the highest. You can use the SORTSEQ= option to change the collating sequence for your output. See “PROC SQL Statement” on page 1027.
- If an ORDER BY clause is omitted, the SAS System’s default collating sequence and your operating environment determine the order of a result table’s rows. Therefore, if you want your result table to appear in a particular order, use the ORDER BY clause.
- Using an ORDER BY clause has certain performance costs, as does any sorting procedure. If you are querying large tables, and the order of their results is not important, your queries will run faster without an ORDER BY clause.
- If more than one *order-by-item* is specified (separated by commas), the first one determines the major sort order. For example, if the *order-by-item* is 2 (an integer), the results are ordered by the values of the second column. If a query-expression includes a set operator (for example, UNION), use integers to specify the order. Doing so avoids ambiguous references to columns in the table expressions.
- In the ORDER BY clause, you can specify any column of a table or view that is specified in the FROM clause of a query-expression, regardless of whether that column has been included in the query’s SELECT clause. For example, this query produces a report ordered by the descending values of the population change for each country from 1990 to 1995:

```
proc sql;
   select country
```

```
      from census
      order by pop95-pop90 desc;


NOTE: The query as specified involves
      ordering by an item that
      doesn't appear in its SELECT clause.
```

- You can order the output by the values that are returned by a function, for example:

```
proc sql;
   select *
      from measure
      order by put(pol_a,fmt_a.);
```

UPDATE Statement

Modifies a column's values in existing rows of a table or view.

Restriction: You cannot use UPDATE on a table accessed via an engine that does not support UPDATE processing.

Featured in: Example 3 on page 1104

UPDATE *table-name*|*sas/access-view*|*proc-sql-view* <**AS** *alias*>

SET *column*=sql-expression
<,*column*=sql-expression>…

<**SET***column*=sql-expression
<,*column*=sql-expression>…>

<**WHERE**sql-expression>;

Arguments

alias
assigns an alias to *table-name*, *sas/access-view*, or *proc-sql-view*.

column
specifies a column in *table-name*, *sas/access-view*, or *proc-sql-view*.

sas/access-view
specifies a SAS/ACCESS view.

sql-expression
See "sql-expression" on page 1081.

table-name
specifies a PROC SQL table.

proc-sql-view
specifies a PROC SQL view.

Updating Tables through Views

You can update one or more rows of a table through a view, with some restrictions. See "Updating PROC SQL and SAS/ACCESS Views" on page 1097.

Details

- Any column that is not modified retains its original values, except in certain queries using the CASE expression. See "CASE expression" on page 1058 for a description of CASE expressions.
- To add, drop, or modify a column's definition or attributes, use the ALTER TABLE statement, described in "ALTER TABLE Statement" on page 1031.
- In the SET clause, a column reference on the left side of the equal sign can also appear as part of the expression on the right side of the equal sign. For example, you could use this expression to give employees a $1,000 holiday bonus:

```
set salary=salary + 1000
```

- If you omit the WHERE clause, all the rows are updated. When you use a WHERE clause, only the rows that meet the WHERE condition are updated.
- When you update a column and an index has been defined for that column, the values in the updated column continue to have the index defined for them.

VALIDATE Statement

Checks the accuracy of a query-expression's syntax without executing the expression.

VALIDATE query-expression;

Argument

query-expression
See "query-expression" on page 1075.

Details

- The VALIDATE statement writes a message in the SAS log that states that the query is valid. If there are errors, VALIDATE writes error messages to the SAS log.
- The VALIDATE statement can also be included in applications that use the macro facility. When used in such an application, VALIDATE returns a value that indicates the query-expression's validity. The value is returned through the macro variable SQLRC (a short form for SQL return code). For example, if a SELECT statement is valid, the macro variable SQLRC returns a value of 0. See "Using Macro Variables Set by PROC SQL" on page 1096 for more information.

Component Dictionary

This section describes the components that are used in SQL procedure statements. *Components* are the items in PROC SQL syntax that appear in roman type.

Most components are contained in clauses within the statements. For example, the basic SELECT statement is composed of the SELECT and FROM clauses, where each clause contains one or more components. Components can also contain other components.

For easy reference, components appear in alphabetical order, and some terms are referred to before they are defined. Use the index or the "See Also" references to refer to other statement or component descriptions that may be helpful.

BETWEEN condition

Selects rows where column values are within a range of values.

sql-expression <NOT> **BETWEEN** sql-expression
AND sql-expression

- sql-expression is described in “sql-expression” on page 1081.

Details

- The sql-expressions must be of compatible data types. They must be either all numeric or all character types.
- Because a BETWEEN condition evaluates the boundary values as a range, it is not necessary to specify the smaller quantity first.
- You can use the NOT logical operator to exclude a range of numbers, for example, to eliminate customer numbers between 1 and 15 (inclusive) so that you can retrieve data on more recently acquired customers.
- PROC SQL supports the same comparison operators that the DATA step supports. For example:

```
x between 1 and 3
x between 3 and 1
1<=x<=3
x>=1 and x<=3
```

CALCULATED

Refers to columns already calculated in the SELECT clause.

CALCULATED *column-alias*

- *column-alias* is the name assigned to the column in the SELECT clause.

Referencing a CALCULATED Column

CALCULATED enables you to use the results of an expression in the same SELECT clause or in the WHERE clause. It is valid only when used to refer to columns that are calculated in the immediate query expression.

CASE expression

Selects result values that satisfy specified conditions.

Featured in: Example 3 on page 1104 and Example 13 on page 1126

CASE <*case-operand*>
WHEN *when-condition* **THEN** *result-expression*
<**WHEN** *when-condition* **THEN** *result-expression*>...
<**ELSE** *result-expression*>
END

- *case-operand*, *when-condition*, and *result-expression* must be valid sql-expressions. See "sql-expression" on page 1081.

Details

The CASE expression selects values if certain conditions are met. A CASE expression returns a single value that is conditionally evaluated for each row of a table (or view). Use the WHEN-THEN clauses when you want to execute a CASE expression for some but not all of the rows in the table that is being queried or created. An optional ELSE expression gives an alternative action if no THEN expression is executed.

When you omit *case-operand*, *when-condition* is evaluated as a Boolean (true or false) value. If *when-condition* returns a nonzero, nonmissing result, the WHEN clause is true. If *case-operand* is specified, it is compared with *when-condition* for equality. If *case-operand* equals *when-condition*, the WHEN clause is true.

If the *when-condition* is true for the row being executed, the *result-expression* following THEN is executed. If *when-condition* is false, PROC SQL evaluates the next *when-condition* until they are all evaluated. If every *when-condition* is false, PROC SQL executes the ELSE expression, and its result becomes the CASE expression's result. If no ELSE expression is present and every *when-condition* is false, the result of the CASE expression is a missing value.

You can use CASE expressions in the SELECT, UPDATE, and INSERT statements.

Example

The following two PROC SQL steps show two equivalent CASE expressions that create a character column with the strings in the THEN clause. The CASE expression in the second PROC SQL step is a shorthand method that is useful when all the comparisons are with the same column.

Example Code 34.1

```
proc sql;
   select *, case
               when degrees > 80 then 'Hot'
               when degrees < 40 then 'Cold'
               else 'Mild'
               end
     from temperatures;

proc sql;
   select *, case Degrees
               when > 80 then 'Hot'
               when < 40 then 'Cold'
               else 'Mild'
               end
     from temperatures;
```

column-definition

Defines PROC SQL's data types and dates.

See also: "column-modifier" on page 1060
Featured in: Example 1 on page 1101

column CHARACTER|VARCHAR <(*width*)>
<column-modifier <column-modifier>...>

column INTEGER|SMALLINT
<column-modifier <column-modifier>...>

column DECIMAL|NUMERIC|FLOAT <(*width*<,*ndec*>)>
<column-modifier <column-modifier>...>

column REAL|DOUBLE PRECISION
<column-modifier <column-modifier>...>

column DATE <column-modifier>

- column-modifier is described in "column-modifier" on page 1060.
- *ndec* is the number of decimals. PROC SQL ignores *ndec*. It is included for compatibility with SQL from other software.
- *width* is the width of the column. The *width* field on a character column specifies the width of that column; it defaults to eight characters. PROC SQL ignores a *width* field on a numeric column. All numeric columns are created with the maximum precision allowed by the SAS System. If you want to create numeric columns that use less storage space, use the LENGTH statement in the DATA step.

Details

- SAS supports many but not all of the data types that SQL-based databases support. The SQL procedure defaults to the SAS data types NUM and CHAR.
- The CHARACTER, INTEGER, and DECIMAL data types can be abbreviated to CHAR, INT, and DEC, respectively.
- A column declared with DATE is a SAS numeric variable with a date informat or format. You can use any of the column-modifiers to set the appropriate attributes for the column being defined. See *SAS Language Reference: Dictionary* for more information on dates.

column-modifier

Sets column attributes.

See also: "column-definition" on page 1059 and SELECT Clause on page 1045
Featured in: Example 1 on page 1101 and Example 2 on page 1103

<**INFORMAT**=*informatw.d*>

<**FORMAT**=*formatw.d*>

<**LABEL**=*'label'*>

<**LENGTH=***length*>

Specifying Informats for Columns (INFORMAT=)

INFORMAT= specifies the informat to be used when SAS accesses data from a table or view. You can change one permanent informat to another by using the ALTER statement. PROC SQL stores informats in its table definitions so that other SAS procedures and the DATA step can use this information when they reference tables created by PROC SQL.

Specifying Formats for Columns (FORMAT=)

FORMAT= determines how character and numeric values in a column are displayed by the query-expression. If the FORMAT= modifier is used in the ALTER, CREATE TABLE, or CREATE VIEW statements, it specifies the permanent format to be used when SAS displays data from that table or view. You can change one permanent format to another by using the ALTER statement.

See *SAS Language Reference: Dictionary* for more information on informats and formats.

Specifying Labels for Columns (LABEL=)

LABEL= associates a label with a column heading. If the LABEL= modifier is used in the ALTER, CREATE TABLE, or CREATE VIEW statements, it specifies the permanent label to be used when displaying that column. You can change one permanent label to another by using the ALTER statement.

If you refer to a labeled column in the ORDER BY or GROUP BY clause, you must use either the column name (not its label), the column's alias, or its ordering integer

(for example, **ORDER BY 2**). See the section on SAS statements in *SAS Language Reference: Dictionary* for more information on labels.

A label can begin with the following characters: a through z, A through Z, 0 through 9, an underscore (_), or a blank space. If you begin a label with any other character, such as pound sign (#), that character is used as a split character and it splits the label onto the next line wherever it appears. For example:

```
select dropout label=
'#Percentage of#Students Who#Dropped Out'
   from educ(obs=5);
```

If you need a special character to appear as the first character in the output, precede it with a space or a forward slash (/).

You can omit the LABEL= part of the column-modifier and still specify a label. Be sure to enclose the label in quotes. For example:

```
select empname "Names of Employees"
   from sql.employees;
```

If you need an apostrophe in the label, type it twice so that the SAS System reads the apostrophe as a literal. Or, you can use single and double quotes alternately (for example, “Date Rec’d”).

column-name

Specifies the column to select.

See also: “column-modifier” on page 1060 and SELECT Clause on page 1045

column-name is one of the following:

column

table-name.column

table-alias.column

view-name.column

view-alias.column

Qualifying Column Names

A column can be referred to by its name alone if it is the only column by that name in all the tables or views listed in the current query-expression. If the same column name exists in more than one table or view in the query expression, you must *qualify* each use of the column name by prefixing a reference to the table that contains it. Consider the following examples:

```
SALARY       /* name of the column */
EMP.SALARY   /* EMP is the table or view name */
E.SALARY     /* E is an alias for the table
                or view that contains the
                SALARY column */
```

CONNECTION TO

Retrieves and uses DBMS data in a PROC SQL query or view.

Tip: You can use CONNECTION TO in the SELECT statement's FROM clause as part of the from-list.

See also: "Connecting to a DBMS Using the SQL Procedure Pass-Through Facility" on page 1095 and your SAS/ACCESS documentation.

CONNECTION TO *dbms-name (dbms-query)*

CONNECTION TO *alias (dbms-query)*

- *alias* specifies an alias, if one was defined in the CONNECT statement.
- *dbms-name* identifies the DBMS you are using.
- *dbms-query* specifies the query to send to a DBMS. The query uses the DBMS's dynamic SQL. You can use any SQL syntax that the DBMS understands, even if that is not valid for PROC SQL. However, your DBMS query cannot contain a semicolon because that represents the end of a statement to the SAS System.

  The number of tables that you can join with *dbms-query* is determined by the DBMS. Each CONNECTION TO component counts as one table toward the 32-table PROC SQL limit for joins.

CONTAINS condition

Tests whether a string is part of a column's value.

Restriction: The CONTAINS condition is used only with character operands.

Featured in: Example 7 on page 1112

sql-expression<NOT> **CONTAINS** sql-expression

For more information, see "sql-expression" on page 1081.

DICTIONARY tables

Retrieve information about elements associated with the current SAS session.

Restriction: You cannot use SAS data set options with DICTIONARY tables.

Restriction: DICTIONARY tables are read-only objects.

Featured in: Example 6 on page 1111

DICTIONARY. *table-name*

- *table-name* is one of the following:

CATALOGS	**MEMBERS**
COLUMNS	OPTIONS
EXTFILES	TABLES
INDEXES	TITLES
MACROS	VIEWS

Querying DICTIONARY Tables

The DICTIONARY tables component is specified in the FROM clause of a SELECT statement. DICTIONARY is a reserved libref for use only in PROC SQL. Data from DICTIONARY tables are generated at run time.

You can use a PROC SQL query to retrieve or subset data from a DICTIONARY table. You can save that query as a PROC SQL view for use later. Or, you can use the existing SASHELP views that are created from the DICTIONARY tables.

To see how each DICTIONARY table is defined, submit a DESCRIBE TABLE statement. After you know how a table is defined, you can use its column names in a subsetting WHERE clause to get more specific information. For example:

```
proc sql;
   describe table dictionary.indexes;
```

The results are written to the SAS log:

```
1   proc sql;
2      describe table dictionary.indexes;
NOTE: SQL table DICTIONARY.INDEXES was created like:

create table DICTIONARY.INDEXES
  (
   libname char(8) label='Library Name',
   memname char(32) label='Member Name',
   memtype char(8) label='Member Type',
   name char(32) label='Column Name',
   idxusage char(9) label='Column Index Type',
   indxname char(32) label='Index Name',
   indxpos num label='Position of Column in Concatenated Key',
   nomiss char(3) label='Nomiss Option',
   unique char(3) label='Unique Option'
  );
```

You specify a DICTIONARY table in a PROC SQL query or view to retrieve information about its objects. For example, the following query returns a row for each index in the INDEXES DICTIONARY table:

```
proc sql;
   title 'DICTIONARY.INDEXES Table';
   select * from dictionary.indexes;
```

Subsetting Data from DICTIONARY Tables

DICTIONARY tables are often large. Therefore, if you are looking for specific information, use a WHERE clause to retrieve a subset of the rows in a DICTIONARY table. In the following example, only the rows with the member name **ADBDBI** are displayed from the DICTIONARY.CATALOGS table:

```
proc sql ;
title 'Subset of the DICTIONARY.CATALOGS Table';
title2 'Rows with Member Name ADBDBI ';
   select * from dictionary.catalogs
      where memname ='ADBDBI';
```

Creating PROC SQL Views from DICTIONARY Tables

To use DICTIONARY tables in other SAS procedures or in the DATA step, use PROC SQL views that are based on the DICTIONARY tables.

You can either create a PROC SQL view on a DICTIONARY table or you can use the SASHELP views, as described in "Accessing DICTIONARY Tables with SASHELP Views" on page 1065. You can then use the view in a DATA or PROC step. The following example creates a PROC SQL view on the DICTIONARY.OPTIONS table. Output 34.2 on page 1064 displays the view with PROC PRINT:

```
options linesize=120 nodate pageno=1;

proc sql;
   create view work.options as
      select * from dictionary.options;


proc print data=work.options(obs=10) noobs;
   title 'Listing of the View WORK.OPTIONS';
   title2 'First 10 Rows Only';
run;
```

Output 34.2 DICTIONARY.OPTIONS Table (partial output)

Listing of the View WORK.OPTIONS 1
First 10 Rows Only

optname	setting	optdesc	level
BATCH	NOBATCH	Use the batch set of default values for SAS system options	Portable
BINDING	DEFAULT	Controls the binding edge for duplexed output	Portable
BOTTOMMARGIN		Bottom margin for printed output	Portable
BUFNO	1	Number of buffers for each SAS data set	Portable
BUFSIZE	0	Size of buffer for page of SAS data set	Portable
BYERR	BYERR	Set the error flag if a null data set is input to the SORT procedure	Portable
BYLINE	BYLINE	Print the by-line at the beginning of each by-group	Portable
CAPS	NOCAPS	Translate SAS source and data lines to uppercase	Portable
CARDIMAGE	NOCARDIMAGE	Process SAS source and data lines as 80-byte records	Portable
CATCACHE	0	Number of SAS catalogs to keep in cache memory	Portable

Accessing DICTIONARY Tables with SASHELP Views

You can use the permanent PROC SQL views that are available in the SASHELP data library to access DICTIONARY tables. Table 34.1 on page 1065 lists all of the permanent PROC SQL views in the SASHELP library as well as the CREATE VIEW statement that defines each view. You can reference these views and display their results using a PROC SQL query, other SAS procedure, or the DATA step.

Table 34.1 Views in DICTIONARY Tables

PROC SQL Views in the SASHELP LIBRARY	PROC SQL Statements to Create the Views
SASHELP.VCATALG	`create view sashelp.vcatalg as select * from dictionary.catalogs;`
SASHELP.VCOLUMN	`create view sashelp.vcolumn as select * from dictionary.columns;`
SASHELP.VEXTFL	`create view sashelp.vextfl as select * from dictionary.extfiles;`
SASHELP.VINDEX	`create view sashelp.vindex as select * from dictionary.indexes;`
SASHELP.VMACRO	`create view sashelp.vmacro as select * from dictionary.macros;`
SASHELP.VMEMBER	`create view sashelp.vmember as select * from dictionary.members;`
SASHELP.VOPTION	`create view sashelp.voption as select * from dictionary.options;`
SASHELP.VTABLE	`create view sashelp.vtable as select * from dictionary.tables;`
SASHELP.VTITLE	`create view sashelp.vtitle as select * from dictionary.titles;`
SASHELP.VVIEW	`create view sashelp.vview as select * from dictionary.views;`
SASHELP.VSACCES	`create view sashelp.vsacces as select libname, memname from dictionary.members where memtype='ACCESS' order by libname, memname;`
SASHELP.VSCATLG	`create view sashelp.vscatlg as select libname, memname from dictionary.members where memtype='CATALOG' order by libname, memname;`

PROC SQL Views in the SASHELP LIBRARY	PROC SQL Statements to Create the Views
SASHELP.VSLIB	`create view sashelp.vslib as select distinct libname, path from dictionary.members order by libname;`
SASHELP.VSTABLE	`create view sashelp.vstable as select libname, memname from dictionary.members where memtype='DATA' order by libname, memname;`
SASHELP.VSTABVW	`create view sashelp.vstabvw as select libname, memname, memtype from dictionary.members where memtype='VIEW' or memtype='DATA' order by libname, memname;`
SASHELP.VSVIEW	`create view sashelp.vsview as select libname, memname from dictionary.members where memtype='VIEW' order by libname, memname;`

EXISTS condition

Tests if a subquery returns one or more rows.

See also: "Query Expressions (Subqueries)" on page 1083

<NOT> **EXISTS** (query-expression)

- query-expression is described in "query-expression" on page 1075.

Details

The EXISTS condition is an operator whose right operand is a subquery. The result of an EXISTS condition is true if the subquery resolves to at least one row. The result of a NOT EXISTS condition is true if the subquery evaluates to zero rows. For example, the following query subsets PROCLIB.PAYROLL (which is shown in Example 2 on page 1103) based on the criteria in the subquery. If the value for STAFF.IDNUM is on the same row as the value **CT** in PROCLIB.STAFF (which is shown in Example 4 on page 1106), the matching IDNUM in PROCLIB.PAYROLL is included in the output. Thus, the query returns all the employees from PROCLIB.PAYROLL who live in **CT**.

```
proc sql;
  select *
```

```
from proclib.payroll p
where exists (select *
                 from proclib.staff s
                 where p.idnumber=s.idnum
                       and state='CT');
```

IN condition

Tests set membership.

Featured in: Example 4 on page 1106

sql-expression <NOT> **IN** (*constant* <,*constant*>...)

sql-expression <NOT> **IN** (query-expresssion)

- *constant* is a number or a quoted character string (or other special notation) that indicates a fixed value. Constants are also called *literals*.
- query-expression is described in "query-expression" on page 1075.
- sql-expression is described in "sql-expression" on page 1081.

Details

An IN condition tests if the column value that is returned by the sql-expression on the left is a member of the set (of constants or values returned by the query-expression) on the right. If so, it selects rows based upon the column value. That is, the IN condition is true if the value of the left-hand operand is in the set of values that are defined by the right-hand operand.

IS condition

Tests for a missing value.

Featured in: Example 5 on page 1108

sql-expression **IS** <NOT> **NULL**

sql-expression **IS** <NOT> **MISSING**

- sql-expression is described in "sql-expression" on page 1081.

Details

IS NULL and IS MISSING are predicates that test for a missing value. IS NULL and IS MISSING are used in the WHERE, ON, and HAVING expressions. Each predicate resolves to true if the sql-expression's result is missing and false if it is not missing.

SAS stores a numeric missing value as a period (.) and a character missing value as a blank space. Unlike missing values in some versions of SQL, missing values in SAS always appear first in the collating sequence. Therefore, in Boolean and comparison operations, the following expressions resolve to true in a predicate:

```
 3>null
-3>null
 0>null
```

The SAS System way of evaluating missing values differs from that of the ANSI Standard for SQL. According to the Standard, these expressions are NULL. See "sql-expression" on page 1081 for more information on predicates and operators. See "PROC SQL and the ANSI Standard" on page 1098 for more information on the ANSI Standard.

joined-table

Joins a table with itself or with other tables.

Restrictions: Joins are limited to 32 tables.

See also: FROM Clause on page 1049 and "query-expression" on page 1075

Featured in: Example 4 on page 1106, Example 7 on page 1112, Example 9 on page 1118, Example 13 on page 1126, and Example 14 on page 1129

table-name <<**AS**> *alias*>, *table-name* <<**AS**> *alias*>
<, *table-name* <<**AS**> *alias*>...>

table-name <**INNER**> **JOIN** *table-name*
ON sql-expression

table-name **LEFT JOIN** *table-name* **ON** sql-expression

table-name **RIGHT JOIN** *table-name* **ON** sql-expression

table-name **FULL JOIN** *table-name* **ON** sql-expression

- *alias* specifies an alias for *table-name*.
- sql-expression is described in "sql-expression" on page 1081.
- *table-name* can be one of the following:
  - the name of a PROC SQL table.
  - the name of a SAS data view.
  - a query-expression. A query-expression in the FROM clause is usually referred to as an *in-line view*. See FROM Clause on page 1049 for more information on in-line views.
  - a connection to a DBMS in the form of the CONNECTION TO component. See "CONNECTION TO" on page 1062 for more information.

Joining Tables

When multiple tables, views, or query-expressions are listed in the FROM clause, they are processed to form one table. The resulting table contains data from each contributing table. These queries are referred to as *joins*.

Conceptually, when two tables are specified, each row of table A is matched with all the rows of table B to produce an internal or intermediate table. The number of rows in the intermediate table (*Cartesian product*) is equal to the product of the number of rows in each of the source tables. The intermediate table becomes the input to the rest of the query in which some of its rows may be eliminated by the WHERE clause or summarized by a summary function.

A common type of join is an *equijoin*, in which the values from a column in the first table must equal the values of a column in the second table.

Table Limit

PROC SQL can process a maximum of 32 tables for a join. If you are using views in a join, the number of tables on which the views are based count toward the 32-table limit. Each CONNECTION TO component in the Pass-Through Facility counts as one table.

Specifying the Rows to Be Returned

The WHERE clause or ON clause contains the conditions (sql-expression) under which the rows in the Cartesian product are kept or eliminated in the result table. WHERE is used to select rows from inner joins. ON is used to select rows from inner or outer joins.

The expression is evaluated for each row from each table in the intermediate table described earlier in "Joining Tables" on page 1069. The row is considered to be matching if the result of the expression is true (a nonzero, nonmissing value) for that row.

Note: You can follow the ON clause with a WHERE clause to further subset the query result. See Example 7 on page 1112 for an example. △

Table Aliases

Table aliases are used in joins to distinguish the columns of one table from those in the other table(s). A table name or alias must be prefixed to a column name when you are joining tables that have matching column names. See FROM Clause on page 1049 for more information on table aliases.

Joining a Table with Itself

A single table can be joined with itself to produce more information. These joins are sometimes called *reflexive joins*. In these joins, the same table is listed twice in the FROM clause. Each instance of the table must have a table alias or you will not be able to distinguish between references to columns in either instance of the table. See Example 13 on page 1126 and Example 14 on page 1129 for examples.

Inner Joins

An *inner join* returns a result table for all the rows in a table that have one or more matching rows in the other table(s), as specified by the sql-expression. Inner joins can be performed on up to 32 tables in the same query-expression.

You can perform an inner join by using a list of table-names separated by commas or by using the INNER, JOIN, and ON keywords.

The LEFTTAB and RIGHTTAB tables are used to illustrate this type of join:

```
                Left Table - LEFTTAB

Continent  Export    Country
-----------------------------
NA         wheat     Canada
EUR        corn      France
EUR        rice      Italy
AFR        oil       Egypt
            Right Table- RIGHTTAB

Continent  Export    Country
-----------------------------
NA         sugar     USA
EUR        corn      Spain
EUR        beets     Belgium
ASIA       rice      Vietnam
```

The following example joins the LEFTTAB and RIGHTTAB tables to get the *Cartesian product* of the two tables. The Cartesian product is the result of combining every row from one table with every row from another table. You get the Cartesian product when you join two tables and do not subset them with a WHERE clause or ON clause.

```
proc sql;
   title 'The Cartesian Product of';
   title2 'LEFTTAB and RIGHTTAB';
   select *
      from lefttab, righttab;
```

```
                The Cartesian Product of
                  LEFTTAB and RIGHTTAB

 Continent  Export    Country   Continent  Export    Country
 ------------------------------------------------------------
 NA         wheat     Canada    NA         sugar     USA
 NA         wheat     Canada    EUR        corn      Spain
 NA         wheat     Canada    EUR        beets     Belgium
 NA         wheat     Canada    ASIA       rice      Vietnam
 EUR        corn      France    NA         sugar     USA
 EUR        corn      France    EUR        corn      Spain
 EUR        corn      France    EUR        beets     Belgium
 EUR        corn      France    ASIA       rice      Vietnam
 EUR        rice      Italy     NA         sugar     USA
 EUR        rice      Italy     EUR        corn      Spain
 EUR        rice      Italy     EUR        beets     Belgium
 EUR        rice      Italy     ASIA       rice      Vietnam
 AFR        oil       Egypt     NA         sugar     USA
 AFR        oil       Egypt     EUR        corn      Spain
 AFR        oil       Egypt     EUR        beets     Belgium
 AFR        oil       Egypt     ASIA       rice      Vietnam
```

The LEFTTAB and RIGHTTAB tables can be joined by listing the table names in the FROM clause. The following query represents an equijoin because the values of Continent from each table are matched. The column names are prefixed with the table aliases so that the correct columns can be selected.

```
proc sql;
   title 'Inner Join';
```

```
select *
   from lefttab as l, righttab as r
   where l.continent=r.continent;
```

Inner Join

Continent	Export	Country	Continent	Export	Country
NA	wheat	Canada	NA	sugar	USA
EUR	corn	France	EUR	corn	Spain
EUR	corn	France	EUR	beets	Belgium
EUR	rice	Italy	EUR	corn	Spain
EUR	rice	Italy	EUR	beets	Belgium

The following PROC SQL step is equivalent to the previous one and shows how to write an equijoin using the INNER JOIN and ON keywords.

```
proc sql;
   title 'Inner Join';
   select *
      from lefttab as l inner join
           righttab as r
      on l.continent=r.continent;
```

See Example 4 on page 1106, Example 13 on page 1126, and Example 14 on page 1129 for more examples.

Outer Joins

Outer joins are inner joins that have been augmented with rows that did not match with any row from the other table in the join. The three types of outer joins are left, right, and full.

A left outer join, specified with the keywords LEFT JOIN and ON, has all the rows from the Cartesian product of the two tables for which the sql-expression is true, plus rows from the first (LEFTTAB) table that do not match any row in the second (RIGHTTAB) table.

```
proc sql;
   title 'Left Outer Join';
   select *
      from lefttab as l left join
           righttab as r
      on l.continent=r.continent;
```

Left Outer Join

Continent	Export	Country	Continent	Export	Country
AFR	oil	Egypt			
EUR	rice	Italy	EUR	beets	Belgium
EUR	corn	France	EUR	beets	Belgium
EUR	rice	Italy	EUR	corn	Spain
EUR	corn	France	EUR	corn	Spain
NA	wheat	Canada	NA	sugar	USA

A right outer join, specified with the keywords RIGHT JOIN and ON, has all the rows from the Cartesian product of the two tables for which the sql-expression is true, plus rows from the second (RIGHTTAB) table that do not match any row in the first (LEFTTAB) table.

```
proc sql;
   title 'Right Outer Join';
   select *
      from lefttab as l right join
           righttab as r
      on l.continent=r.continent;
```

Right Outer Join

Continent	Export	Country	Continent	Export	Country
			ASIA	rice	Vietnam
EUR	rice	Italy	EUR	beets	Belgium
EUR	rice	Italy	EUR	corn	Spain
EUR	corn	France	EUR	beets	Belgium
EUR	corn	France	EUR	corn	Spain
NA	wheat	Canada	NA	sugar	USA

A full outer join, specified with the keywords FULL JOIN and ON, has all the rows from the Cartesian product of the two tables for which the sql-expression is true, plus rows from each table that do not match any row in the other table.

```
proc sql;
   title 'Full Outer Join';
   select *
      from lefttab as l full join
           righttab as r
      on l.continent=r.continent;
```

Full Outer Join

Continent	Export	Country	Continent	Export	Country
AFR	oil	Egypt			
			ASIA	rice	Vietnam
EUR	rice	Italy	EUR	beets	Belgium
EUR	rice	Italy	EUR	corn	Spain
EUR	corn	France	EUR	beets	Belgium
EUR	corn	France	EUR	corn	Spain
NA	wheat	Canada	NA	sugar	USA

See Example 7 on page 1112 for another example.

Joining More Than Two Tables

Inner joins are usually performed on two or three tables, but they can be performed on up to 32 tables in PROC SQL. A join on three tables is described here to explain how and why the relationships work among the tables.

In a three-way join, the sql-expression consists of two conditions: one relates the first table to the second table and the other relates the second table to the third table. It is possible to break this example into stages, performing a two-way join into a temporary table and then joining that table with the third one for the same result. However, PROC SQL can do it all in one step as shown in the next example.

The example shows the joining of three tables: COMM, PRICE, and AMOUNT. To calculate the total revenue from exports for each country, you need to multiply the amount exported (AMOUNT table) by the price of each unit (PRICE table), and you must know the commodity that each country exports (COMM table).

COMM Table

Continent	Export	Country
NA	wheat	Canada
EUR	corn	France
EUR	rice	Italy
AFR	oil	Egypt

PRICE Table

Export	Price
rice	3.56
corn	3.45
oil	18
wheat	2.98

AMOUNT Table

Country	Quantity
Canada	16000
France	2400
Italy	500
Egypt	10000

```
proc sql;
   title  'Total Export Revenue';
   select c.Country, p.Export, p.Price,
          a.Quantity,a.quantity*p.price
          as Total
      from comm c, price p, amount a
      where c.export=p.export
            and c.country=a.country;
```

Total Export Revenue

Country	Export	Price	Quantity	Total
Italy	rice	3.56	500	1780
France	corn	3.45	2400	8280
Egypt	oil	18	10000	180000
Canada	wheat	2.98	16000	47680

See Example 9 on page 1118 for another example.

Comparison of Joins and Subqueries

You can often use a subquery and a join to get the same result. However, it is often more efficient to use a join if the outer query and the subquery do not return duplicate rows. For example, the following queries produce the same result. The second query is more efficient:

```
proc sql;
   select IDNumber, Birth
      from proclib.payroll
      where IDNumber in (select idnum
                     from proclib.staff
                     where lname like 'B%');

proc sql;
   select  p.IDNumber, p.Birth
     from proclib.payroll p, proclib.staff s
     where p.idnumber=s.idnum
           and s.lname like 'B%';
```

Note: PROCLIB.PAYROLL is shown in Example 2 on page 1103. △

LIKE condition

Tests for a matching pattern.

sql-expression <NOT> **LIKE** sql-expression

□ sql-expression is described in "sql-expression" on page 1081.

Details

The LIKE condition selects rows by comparing character strings with a pattern-matching specification. It resolves to true and displays the matched string(s) if the left operand matches the pattern specified by the right operand.

Patterns for Searching

Patterns are composed of three classes of characters:

underscore (_)
   matches any single character.

percent sign (%)
   matches any sequence of zero or more characters.

any other character
   matches that character.

These patterns can appear before, after, or on both sides of characters that you want to match. The LIKE condition is case-sensitive.

The following list uses these values: **Smith**, **Smooth**, **Smothers**, **Smart**, and **Smuggle**.

'Sm%'
matches **Smith**, **Smooth**, **Smothers**, **Smart**, **Smuggle**.

'%th'
matches **Smith**, **Smooth**.

'S__gg%'
matches **Smuggle**.

'S_o'
matches a three-letter word, so it has no matches here.

'S_o%'
matches **Smooth**, **Smothers**.

'S%th'
matches **Smith**, **Smooth**.

'Z'
matches the single, uppercase character **Z** only, so it has no matches here.

Searching for Mixed-Case Strings

To search for mixed-case strings, use the UPCASE function to make all the names uppercase before entering the LIKE condition:

```
upcase(name) like 'SM%';
```

Note: When you are using the % character, be aware of the effect of trailing blanks. You may have to use the TRIM function to remove trailing blanks in order to match values. △

query-expression

Retrieves data from tables.

See also: "table-expression" on page 1094, "Query Expressions (Subqueries)" on page 1083, and "In-Line Views" on page 1050

table-expression <*set-operator* table-expression>...

- table-expression is described in "table-expression" on page 1094.
- *set-operator* is one of the following:

  INTERSECT <CORRESPONDING> <ALL>

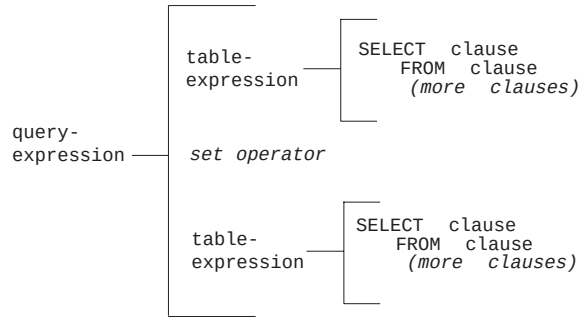
  OUTER UNION <CORRESPONDING>

  UNION <CORRESPONDING> <ALL>

  EXCEPT <CORRESPONDING> <ALL>

Query Expressions and Table Expressions

A query-expression is one or more table-expressions. Multiple table expressions are linked by set operators. The following figure illustrates the relationship between table-expressions and query-expressions.



Set Operators

PROC SQL provides traditional set operators from relational algebra:

OUTER UNION
concatenates the query results.

UNION
produces all unique rows from both queries.

EXCEPT
produces rows that are part of the first query only.

INTERSECT
produces rows that are common to both query results.

A query-expression with set operators is evaluated as follows.

- Each table-expression is evaluated to produce an (internal) intermediate result table.
- Each intermediate result table then becomes an operand linked with a set operator to form an expression, for example, A UNION B.
- If the query-expression involves more than two table-expressions, the result from the first two becomes an operand for the next set operator and operand, for example, (A UNION B) EXCEPT C, ((A UNION B) EXCEPT C) INTERSECT D, and so on.
- Evaluating a query-expression produces a single output table.

Set operators follow this order of precedence unless they are overridden by parentheses in the expression(s): INTERSECT is evaluated first. OUTER UNION, UNION, and EXCEPT have the same level of precedence.

PROC SQL performs set operations even if the tables or views that are referred to in the table-expressions do not have the same number of columns. The reason for this is that the ANSI Standard for SQL requires that tables or views involved in a set operation have the same number of columns and that the columns have matching data types. If a set operation is performed on a table or view that has fewer columns than the one(s) with which it is being linked, PROC SQL extends the table or view with fewer columns by creating columns with missing values of the appropriate data type. This temporary alteration enables the set operation to be performed correctly.

CORRESPONDING (CORR) Keyword

The CORRESPONDING keyword is used only when a set operator is specified. CORR causes PROC SQL to match the columns in table-expressions *by name* and not by ordinal position. Columns that do not match by name are excluded from the result table, except for the OUTER UNION operator. See "OUTER UNION" on page 1077.

For example, when performing a set operation on two table-expressions, PROC SQL matches the first specified column-name (listed in the SELECT clause) from one table-expression with the first specified column-name from the other. If CORR is omitted, PROC SQL matches the columns by ordinal position.

ALL Keyword

The set operators automatically eliminate duplicate rows from their output tables. The optional ALL keyword preserves the duplicate rows, reduces the execution by one step, and thereby improves the query-expression's performance. You use it when you want to display all the rows resulting from the table-expressions, rather than just the rows that are output because duplicates have been deleted. The ALL keyword is used only when a set operator is also specified.

OUTER UNION

Performing an OUTER UNION is very similar to performing the SAS DATA step with a SET statement. The OUTER UNION concatenates the intermediate results from the table-expressions. Thus, the result table for the query-expression contains all the rows produced by the first table-expression followed by all the rows produced by the second table-expression. Columns with the same name are in separate columns in the result table.

For example, the following query expression concatenates the ME1 and ME2 tables but does not overlay like-named columns. Output 34.3 on page 1078 shows the result.

```
proc sql;
   title 'ME1 and ME2: OUTER UNION';
   select *
      from me1
   outer union
   select *
      from me2;
```

ME1

IDnum	Jobcode	Salary	Bonus
1400	ME1	29769	587
1403	ME1	28072	342
1120	ME1	28619	986
1120	ME1	28619	986

ME2

IDnum	Jobcode	Salary
1653	ME2	35108
1782	ME2	35345
1244	ME2	36925

Output 34.3 OUTER UNION of ME1 and ME2 Tables

```
ME1 and ME2: OUTER UNION

      IDnum     Jobcode     Salary     Bonus  IDnum     Jobcode     Salary
      ---------------------------------------------------------------------
      1400      ME1          29769       587                              .
      1403      ME1          28072       342                              .
      1120      ME1          28619       986                              .
      1120      ME1          28619       986                              .
                                 .         .  1653      ME2          35108
                                 .         .  1782      ME2          35345
                                 .         .  1244      ME2          36925
```

To overlay columns with the same name, use the CORRESPONDING keyword.

```
proc sql;
   title 'ME1 and ME2: OUTER UNION CORRESPONDING';
   select *
      from me1
   outer union corr
   select *
      from me2;
```

```
ME1 and ME2: OUTER UNION CORRESPONDING

             IDnum     Jobcode     Salary     Bonus
             ---------------------------------------
             1400      ME1          29769       587
             1403      ME1          28072       342
             1120      ME1          28619       986
             1120      ME1          28619       986
             1653      ME2          35108         .
             1782      ME2          35345         .
             1244      ME2          36925         .
```

In the resulting concatenated table, notice the following:

- OUTER UNION CORRESPONDING retains all nonmatching columns.
- For columns with the same name, if a value is missing from the result of the first table-expression, the value in that column from the second table-expression is inserted.
- The ALL keyword is not used with OUTER UNION because this operator's default action is to include all rows in a result table. Thus, both rows from the table ME1 where IDnum is **1120** appear in the output.

UNION

The UNION operator produces a table that contains all the unique rows that result from both table-expressions. That is, the output table contains rows produced by the first table-expression, the second table-expression, or both.

Columns are appended by position in the tables, regardless of the column names. However, the data type of the corresponding columns must match or the union will not occur. PROC SQL issues a warning message and stops executing.

The names of the columns in the output table are the names of the columns from the first table-expression unless a column (such as an expression) has no name in the first table-expression. In such a case, the name of that column in the output table is the name of the respective column in the second table-expression.

In the following example, PROC SQL combines the two tables:

```
proc sql;
   title 'ME1 and ME2: UNION';
   select *
      from me1
   union
   select *
      from me2;
```

```
ME1 and ME2: UNION

             IDnum     Jobcode     Salary     Bonus
             ---------------------------------------
             1120      ME1          28619       986
             1244      ME2          36925         .
             1400      ME1          29769       587
             1403      ME1          28072       342
             1653      ME2          35108         .
             1782      ME2          35345         .
```

In the following example, ALL includes the duplicate row from ME1. In addition, ALL changes the sorting by specifying that PROC SQL make one pass only. Thus, the values from ME2 are simply appended to the values from ME1.

```
proc sql;
   title 'ME1 and ME2: UNION ALL';
   select *
      from me1
   union all
   select *
      from me2;
```

```
ME1 and ME2: UNION ALL

             IDnum     Jobcode     Salary     Bonus
             ---------------------------------------
             1400      ME1          29769       587
             1403      ME1          28072       342
             1120      ME1          28619       986
             1120      ME1          28619       986
             1653      ME2          35108         .
             1782      ME2          35345         .
             1244      ME2          36925         .
```

See Example 5 on page 1108 for another example.

EXCEPT

The EXCEPT operator produces (from the first table-expression) an output table that has unique rows that are not in the second table-expression. If the intermediate result from the first table-expression has at least one occurrence of a row that is not in the intermediate result of the second table-expression, that row (from the first table-expression) is included in the result table.

In the following example, the IN_USA table contains flights to cities within and outside the USA. The OUT_USA table contains flights only to cities outside the USA. This example returns only the rows from IN_USA that are not also in OUT_USA:

```
proc sql;
    title 'Flights from IN_USA';
    select * from in_usa
    except
    select * from out_usa;
```

IN_USA

Flight	Dest
145	ORD
156	WAS
188	LAX
193	FRA
207	LON

OUT_USA

Flight	Dest
193	FRA
207	LON
311	SJA

Flights from IN_USA

Flight	Dest
145	ORD
156	WAS
188	LAX

INTERSECT

The INTERSECT operator produces an output table that has rows that are common to both tables. For example, using the IN_USA and OUT_USA tables shown above, the following example returns rows that are in both tables:

```
proc sql;
    title 'Flights from IN_USA and OUT_USA';
    select * from in_usa
    intersect
    select * from out_usa;
```

```
Flights from IN_USA and OUT_USA

                      Flight    Dest
                      ------------------
                      193       FRA
                      207       LON
```

sql-expression

Produces a value from a sequence of operands and operators.

operand operator operand

- *operand* is one of the following:
  - *constant* is a number or a quoted character string (or other special notation) that indicates a fixed value. Constants are also called *literals*. Constants are described in *SAS Language Reference: Dictionary*.
  - column-name is described in "column-name" on page 1061.
  - *SAS-function* is almost any SAS function. Functions are described in *SAS Language Reference: Dictionary*.
  - The ANSI SQL function COALESCE is supported.
  - summary-function is described in "summary-function" on page 1088.
  - query-expression is described in "query-expression" on page 1075.
  - USER is a literal that references the userid of the person who submitted the program. The userid that is returned is operating environment-dependent, but PROC SQL uses the same value that the &SYSJOBID macro variable has on the operating environment.
- *operator* is described in "Operators and the Order of Evaluation" on page 1082.

Note: SAS functions, including summary functions, can stand alone as SQL expressions. For example

```
select min(x) from table;

select scan(y,4) from table;
```

△

SAS Functions

PROC SQL supports the same SAS functions as the DATA step, except for the functions LAG, DIF, and SOUND. For example, the SCAN function is used in the following query:

```
select style, scan(street,1) format=$15.
   from houses;
```

See *SAS Language Reference: Dictionary* for complete documentation on SAS functions. Summary functions are also SAS functions. See "summary-function" on page 1088 for more information.

COALESCE Function

PROC SQL also supports the ANSI SQL function COALESCE. COALESCE accepts multiple column names of the same data type. The COALESCE function returns the first argument whose value is not a SAS missing value. In some SQL DBMSs, the COALESCE function is called the IFNULL function. See "PROC SQL and the ANSI Standard" on page 1098 for more information.

For an example that uses COALESCE, see Example 7 on page 1112.

USER Literal

USER can be specified in a view definition, for example, to create a view that restricts access to those in the user's department:

```
create view myemp as
   select * from dept12.employees
      where manager=user;
```

This view produces a different set of employee information for each manager who references it.

Operators and the Order of Evaluation

The order in which operations are evaluated is the same as in the DATA step with this one exception: NOT is grouped with the logical operators AND and OR in PROC SQL; in the DATA step, NOT is grouped with the unary plus and minus signs.

Unlike missing values in some versions of SQL, missing values in the SAS System always appear first in the collating sequence. Therefore, in Boolean and comparison operations, the following expressions resolve to true in a predicate:

```
 3>null
-3>null
 0>null
```

You can use parentheses to group values or to nest mathematical expressions. Parentheses make expressions easier to read and can also be used to change the order of evaluation of the operators. Evaluating expressions with parentheses begins at the deepest level of parentheses and moves outward. For example, SAS evaluates A+B*C as A+(B*C), although you can add parentheses to make it evaluate as (A+B)*C for a different result.

Higher priority operations are performed first: that is, group 0 operators are evaluated before group 5 operators. Table 34.2 on page 1082 shows the operators and their order of evaluation, including their priority groups.

Table 34.2 Operators and Order of Evaluation

Group	Operator	Description
0	( )	forces the expression enclosed to be evaluated first
1	case-expression	selects result values that satisfy specified conditions

Group	Operator	Description
2	**	raises to a power
	unary +, unary -	indicates a positive or negative number
3	*	multiplies
	/	divides
4	+	adds
	–	subtracts
5	||	concatenates
6	<NOT> BETWEEN condition	See "BETWEEN condition" on page 1057.
	<NOT> CONTAINS condition	see "CONTAINS condition" on page 1062.
	<NOT> EXISTS condition	See "EXISTS condition" on page 1066.
	<NOT> IN condition	See "IN condition" on page 1067.
	IS <NOT> condition	See "IS condition" on page 1067.
	<NOT> LIKE condition	See "LIKE condition" on page 1074.
7	=, eq	equals
	¬=, ^=, < >, ne	does not equal
	>, gt	is greater than
	<, lt	is less than
	>=, ge	is greater than or equal to
	<=, le	is less than or equal to
	=*	sounds like (use with character operands only). See Example 11 on page 1122.
8	&, AND	indicates logical AND
9	|, OR	indicates logical OR
10	¬, ^, NOT	indicates logical NOT

Symbols for operators may vary, depending on the operating environment. See *SAS Language Reference: Dictionary* for more information on operators and expressions.

Query Expressions (Subqueries)

Query-expressions are called *subqueries* when used in WHERE or HAVING clauses. A subquery is a query-expression that is nested as part of another query-expression. A subquery selects one or more rows from a table based on values in another table.

Depending on the clause that contains it, a subquery can return a single value or multiple values. If more than one subquery is used in a query-expression, the innermost query is evaluated first, then the next innermost query, and so on, moving outward.

PROC SQL allows a subquery (contained in parentheses) at any point in an expression where a simple column value or constant can be used. In this case, a subquery must return a *single value*, that is, one row with only one column. When a subquery returns one value, you can name the value with a column alias and refer to it by that name elsewhere in the query. This is useful for replacing values with other values returned using a subquery.

The following is an example of a subquery that returns one value. This PROC SQL step subsets the PROCLIB.PAYROLL table based on information in the PROCLIB.STAFF table. (PROCLIB.PAYROLL is shown in Example 2 on page 1103, and PROCLIB.STAFF is shown in Example 4 on page 1106.) PROCLIB.PAYROLL contains employee identification numbers (IdNumber) and their salaries (Salary) but does not contain their names. If you want to return only the row from PROCLIB.PAYROLL for one employee, you can use a subquery that queries the PROCLIB.STAFF table, which contains the employees' identification numbers and their names (Lname and Fname).

```
options ls=64 nodate nonumber;
proc sql;
    title 'Information for Earl Bowden';
    select *
       from proclib.payroll
       where idnumber=
             (select idnum
             from proclib.staff
             where upcase(lname)='BOWDEN');
```

```
Information for Earl Bowden

        Id
        Number  Sex  Jobcode    Salary    Birth    Hired
        -------------------------------------------------
        1403    M    ME1         28072  28JAN69  21DEC91
```

Subqueries can return *multiple values*. The following example uses the tables PROCLIB.DELAY and PROCLIB.MARCH. These tables contain information about the same flights and have the Flight column in common. The following subquery returns all the values for Flight in PROCLIB.DELAY for international flights. The values from the subquery complete the WHERE clause in the outer query. Thus, when the outer query is executed, only the international flights from PROCLIB.MARCH are in the output.

```
options ls=64 nodate nonumber;
proc sql outobs=5;
   title 'International Flights from';
   title2 'PROCLIB.MARCH';
   select Flight, Date, Dest, Boarded
      from proclib.march
      where flight in
            (select flight
             from proclib.delay
             where destype='International');
```

```
International Flights from
                        PROCLIB.MARCH

                Flight     Date  Dest   Boarded
                -------------------------------
                219      01MAR94  LON        198
                622      01MAR94  FRA        207
                132      01MAR94  YYZ        115
                271      01MAR94  PAR        138
                219      02MAR94  LON        147
```

Sometimes it is helpful to compare a value with a set of values returned by a subquery. The keywords ANY or ALL can be specified before a subquery when the subquery is the right-hand operand of a comparison. If ALL is specified, the comparison is true only if it is true for all values returned by the subquery. If a subquery returns no rows, the result of an ALL comparison is true for each row of the outer query.

If ANY is specified, the comparison is true if it is true for any one of the values returned by the subquery. If a subquery returns no rows, the result of an ANY comparison is false for each row of the outer query.

The following example selects all those in PROCLIB.PAYROLL who earn more than the highest paid **ME3**:

```
options ls=64 nodate nonumber ;
proc sql;
title ''Employees who Earn More than'';
title2 ''All ME's'';
   select *
     from proclib.payroll
     where salary > all (select salary
                         from proclib.payroll
                         where jobcode='ME3');
```

```
                Employees who Earn More than
                          All ME's

       Id
       Number  Sex  Jobcode    Salary    Birth    Hired
       ------------------------------------------------
       1333    M    PT2         88606  30MAR61  10FEB81
       1739    M    PT1         66517  25DEC64  27JAN91
       1428    F    PT1         68767  04APR60  16NOV91
       1404    M    PT2         91376  24FEB53  01JAN80
       1935    F    NA2         51081  28MAR54  16OCT81
       1905    M    PT1         65111  16APR72  29MAY92
       1407    M    PT1         68096  23MAR69  18MAR90
       1410    M    PT2         84685  03MAY67  07NOV86
       1439    F    PT1         70736  06MAR64  10SEP90
       1545    M    PT1         66130  12AUG59  29MAY90
       1106    M    PT2         89632  06NOV57  16AUG84
       1442    F    PT2         84536  05SEP66  12APR88
       1417    M    NA2         52270  27JUN64  07MAR89
       1478    M    PT2         84203  09AUG59  24OCT90
       1556    M    PT1         71349  22JUN64  11DEC91
```

Employees who Earn More than All ME's

Id Number	Sex	Jobcode	Salary	Birth	Hired
1352	M	NA2	53798	02DEC60	16OCT86
1890	M	PT2	91908	20JUL51	25NOV79
1107	M	PT2	89977	09JUN54	10FEB79
1830	F	PT2	84471	27MAY57	29JAN83
1928	M	PT2	89858	16SEP54	13JUL90
1076	M	PT1	66558	14OCT55	03OCT91

Note: See the first item in "Subqueries and Efficiency" on page 1087 for a note about efficiency when using ALL. △

Correlated Subqueries

In a correlated subquery, the WHERE expression in a subquery refers to values in a table in the outer query. The correlated subquery is evaluated for each row in the outer query. With correlated subqueries, PROC SQL executes the subquery and the outer query together.

The following example uses the PROCLIB.DELAY and PROCLIB.MARCH tables. A DATA step"PROCLIB.DELAY" on page 1506 creates PROCLIB.DELAY. PROCLIB.MARCH is shown in Example 13 on page 1126. PROCLIB.DELAY has the Flight, Date, Orig, and Dest columns in common with PROCLIB.MARCH:

```
proc sql outobs=5;
   title 'International Flights';
   select *
      from proclib.march
      where 'International' in
           (select destype
            from proclib.delay
            where march.Flight=delay.Flight);
```

The subquery resolves by substituting every value for MARCH.Flight into the subquery's WHERE clause, one row at a time. For example, when MARCH.Flight= **219**, the subquery resolves as follows:

1. PROC SQL retrieves all the rows from DELAY where Flight= **219** and passes their DESTYPE values to the WHERE clause.
2. PROC SQL uses the DESTYPE values to complete the WHERE clause:

   ```
   where 'International' in
      ('International','International', ...)
   ```

3. The WHERE clause checks to see if **International** is in the list. Because it is, all rows from MARCH that have a value of **219** for Flight become part of the output.

Output 34.4 on page 1086 contains the rows from MARCH for international flights only.

Output 34.4 International Flights for March

International Flights

Flight	Date	Depart	Orig	Dest	Miles	Boarded	Capacity
219	01MAR94	9:31	LGA	LON	3442	198	250
622	01MAR94	12:19	LGA	FRA	3857	207	250
132	01MAR94	15:35	LGA	YYZ	366	115	178
271	01MAR94	13:17	LGA	PAR	3635	138	250
219	02MAR94	9:31	LGA	LON	3442	147	250

Subqueries and Efficiency

- Use the MAX function in a subquery instead of the ALL keyword before the subquery. For example, the following queries produce the same result, but the second query is more efficient:

```
proc sql;
   select * from proclib.payroll
   where salary> all(select salary
                     from proclib.payroll
                     where jobcode='ME3');

proc sql;
   select * from proclib.payroll
   where salary> (select max(salary)
                  from proclib.payroll
                  where jobcode='ME3');
```

- With subqueries, use IN instead of EXISTS when possible. For example, the following queries produce the same result, but the second query is more efficient:

```
proc sql;
   select *
      from proclib.payroll p
      where exists (select *
                    from staff s
                    where p.idnum=s.idnum
                       and state='CT');

proc sql;
   select *
      from proclib.payroll
      where idnum in (select idnum
                         from staff
                         where state='CT');
```

summary-function

Performs statistical summary calculations.

Restriction: A summary function cannot appear in an ON clause or a WHERE clause.

See also: GROUP BY on page 1052, HAVING Clause on page 1053, SELECT Clause on page 1045, and "table-expression" on page 1094

Featured in: Example 8 on page 1116, Example 12 on page 1124, and Example 15 on page 1131

summary-function (<DISTINCT|ALL> sql-expression)

- sql-expression is described in "sql-expression" on page 1081.

Summarizing Data

Summary functions produce a statistical summary of the entire table or view listed in the FROM clause or for each group specified in a GROUP BY clause. If GROUP BY is omitted, all the rows in the table or view are considered to be a single group. These functions reduce all the values in each row or column in a table to one *summarizing* or *aggregate* value. For this reason, these functions are often called *aggregate functions*. For example, the sum (one value) of a column results from the addition of all the values in the column.

Function Names and the Corresponding Statistics

Some functions have more than one name to accommodate both SAS and SQL conventions:

AVG, MEAN
: means or average of values

COUNT, FREQ, N
: number of nonmissing values

CSS
: corrected sum of squares

CV
: coefficient of variation (percent)

MAX
: largest value

MIN
: smallest value

NMISS
: number of missing values

PRT
: probability of a greater absolute value of Student's *t*

RANGE
: range of values

STD
standard deviation

STDERR
standard error of the mean

SUM
sum of values

SUMWGT
sum of the WEIGHT variable values*

T
Student's *t* value for testing the hypothesis that the population mean is zero

USS
uncorrected sum of squares

VAR
variance

For a description and the formulas used for these statistics, see Appendix 1, "SAS Elementary Statistics Procedures," on page 1457

Counting Rows

The COUNT function counts rows. COUNT(*) returns the total number of rows in a group or in a table. If you use a column name as an argument to COUNT, the result is the total number of rows in a group or in a table that have a nonmissing value for that column. If you want to count the unique values in a column, specify COUNT(DISTINCT *column*).

If the SELECT clause of a table-expression contains one or more summary functions and that table-expression resolves to no rows, then the summary function results are missing values. The following are exceptions that return zeros:

COUNT(*)

COUNT(<DISTINCT> sql-expression)

NMISS(<DISTINCT> sql-expression)

See Example 8 on page 1116 and Example 15 on page 1131 for examples.

Calculating Statistics Based on the Number of Arguments

The number of arguments specified in a summary function affects how the calculation is performed. If you specify a single argument, the values in the column are calculated. If you specify multiple arguments, the arguments or columns listed are calculated for each row. For example, consider calculations on the following table.

```
proc sql;
   title 'Summary Table';
   select * from summary;
```

* Currently, there is no way to designate a WEIGHT variable for a table in PROC SQL. Thus, each row (or observation) has a weight of 1.

```
                    Summary Table

                    X         Y         Z
                ---------------------------
                    1         3         4
                    2         4         5
                    8         9         4
                    4         5         4
```

If you use one argument in the function, the calculation is performed on that column only. If you use more than one argument, the calculation is performed on each row of the specified columns. In the following PROC SQL step, the MIN and MAX functions return the minimum and maximum of the columns they are used with. The SUM function returns the sum of each row of the columns specified as arguments:

```
proc sql;
    select min(x) as Colmin_x,
           min(y) as Colmin_y,
           max(z) as Colmax_z,
           sum(x,y,z) as Rowsum
      from summary;
```

```
                    Summary Table

          Colmin_x  Colmin_y  Colmax_z    Rowsum
          ------------------------------------
                 1         3         5         8
                 1         3         5        11
                 1         3         5        21
                 1         3         5        13
```

Remerging Data

When you use a summary function in a SELECT clause or a HAVING clause, you may see the following message in the SAS log:

```
NOTE: The query requires remerging summary
      statistics back with the original
      data.
```

The process of *remerging* involves two passes through the data. On the first pass, PROC SQL

- calculates and returns the value of summary functions. It then uses the result to calculate the arithmetic expressions in which the summary function participates.
- groups data according to the GROUP BY clause.

On the second pass, PROC SQL retrieves any additional columns and rows that it needs to show in the output.

The following examples use the PROCLIB.PAYROLL table (shown in Example 2 on page 1103) to show when remerging of data is and is not necessary.

The first query requires remerging. The first pass through the data groups the data by Jobcode and resolves the AVG function for each group. However, PROC SQL must make a second pass in order to retrieve the values of IdNumber and Salary.

```
proc sql outobs=10;
   title 'Salary Information';
   title2 '(First 10 Rows Only)';
   select  IdNumber, Jobcode, Salary,
           avg(salary) as AvgSalary
      from proclib.payroll
      group by jobcode;
```

```
                    Salary Information
                   (First 10 Rows Only)

              Id
              Number  Jobcode    Salary  AvgSalary
              ------------------------------------
              1845    BCK         25996   25794.22
              1673    BCK         25477   25794.22
              1834    BCK         26896   25794.22
              1389    BCK         25028   25794.22
              1100    BCK         25004   25794.22
              1677    BCK         26007   25794.22
              1663    BCK         26452   25794.22
              1383    BCK         25823   25794.22
              1704    BCK         25465   25794.22
              1132    FA1         22413   23039.36
```

You can change the previous query to return only the average salary for each jobcode. The following query does not require remerging because the first pass of the data does the summarizing and the grouping. A second pass is not necessary.

```
proc sql outobs=10;
   title 'Average Salary for Each Jobcode';
   select Jobcode, avg(salary) as AvgSalary
   from proclib.payroll
   group by jobcode;
```

```
              Average Salary for Each Jobcode

                     Jobcode  AvgSalary
                     ------------------
                     BCK       25794.22
                     FA1       23039.36
                     FA2       27986.88
                     FA3       32933.86
                     ME1       28500.25
                     ME2       35576.86
                     ME3       42410.71
                     NA1        42032.2
                     NA2          52383
                     PT1          67908
```

When you use the HAVING clause, PROC SQL may have to remerge data to resolve the HAVING expression.

First, consider a query that uses HAVING but that does not require remerging. The query groups the data by values of Jobcode, and the result contains one row for each value of Jobcode and summary information for people in each Jobcode. On the first

pass, the summary functions provide values for the **Number**, **Average Age**, and **Average Salary** columns. The first pass provides everything that PROC SQL needs to resolve the HAVING clause, so no remerging is necessary.

```
proc sql outobs=10;
title 'Summary Information for Each Jobcode';
title2 '(First 10 Rows Only)';
   select Jobcode,
          count(jobcode) as number
              label='Number',
          avg(int((today()-birth)/365.25))
              as avgage format=2.
              label='Average Age',
          avg(salary) as avgsal format=dollar8.
              label='Average Salary'
      from proclib.payroll
      group by jobcode
      having avgage ge 30;
```

```
                    Summary Information for Each Jobcode                    1
                            (First 10 Rows Only)

                                          Average   Average
                    Jobcode    Number         Age    Salary
                    ------------------------------------
                    BCK             9          33   $25,794
                    FA1            11          30   $23,039
                    FA2            16          34   $27,987
                    FA3             7          36   $32,934
                    ME1             8          31   $28,500
                    ME2            14          37   $35,577
                    ME3             7          39   $42,411
                    NA2             3          39   $52,383
                    PT1             8          35   $67,908
                    PT2            10          40   $87,925
```

In the following query, PROC SQL remerges the data because the HAVING clause uses the SALARY column in the comparison and SALARY is not in the GROUP BY clause.

```
proc sql outobs=10;
title 'Employees who Earn More than the';
title2 'Average for Their Jobcode';
title3 '(First 10 Rows Only)';
   select Jobcode, Salary,
          avg(salary) as AvgSalary
      from proclib.payroll
      group by jobcode
      having salary > AvgSalary;
```

```
        Employees who Earn More than the
           Average for Their Jobcode
              (First 10 Rows Only)

          Jobcode    Salary  AvgSalary
          --------------------------
          BCK         25996   25794.22
          BCK         26896   25794.22
          BCK         26007   25794.22
          BCK         26452   25794.22
          BCK         25823   25794.22
          FA1         23738   23039.36
          FA1         23916   23039.36
          FA1         23644   23039.36
          FA1         23979   23039.36
          FA1         23177   23039.36
```

Keep in mind that PROC SQL remerges data when

- the values returned by a summary function are used in a calculation. For example, the following query returns the values of X and the percent of the total for each row. On the first pass, PROC SQL computes the sum of X, and on the second pass PROC SQL computes the percentage of the total for each value of X:

```
proc sql;
   title 'Percentage of the Total';
   select X, (100*x/sum(X)) as Pct_Total
      from summary;
```

```
                Percentage of the Total

                        x  Pct_Total
                ------------------
                       32   14.81481
                       86   39.81481
                       49   22.68519
                       49   22.68519
```

- the values returned by a summary function are compared to values of a column that is not specified in the GROUP BY clause. For example, the following query uses the PROCLIB.PAYROLL table. PROC SQL remerges data because the column Salary is not specified in the GROUP BY clause:

```
proc sql;
   select  jobcode,  salary,
           avg(salary) as avsal
      from proclib.payroll
      group by jobcode
      having salary > avsal;
```

- a column from the input table is specified in the SELECT clause and is not specified in the GROUP BY clause. This rule does not refer to columns used as arguments to summary functions in the SELECT clause.

  For example, in the following query, the presence of IdNumber in the SELECT clause causes PROC SQL to remerge the data because IdNumber is not involved in

grouping or summarizing during the first pass. In order for PROC SQL to retrieve the values for IdNumber, it must make a second pass through the data.

```
proc sql;
   select IdNumber, jobcode,
          avg(salary) as avsal
      from proclib.payroll
      group by jobcode;
```

table-expression

Defines part or all of a query-expression.

See also: "query-expression" on page 1075

SELECT <**DISTINCT**> *object-item*<,*object-item*>…
<**INTO** :*macro-variable-specification*
<, :*macro-variable-specification*>…>
FROM *from-list*
<**WHERE** sql-expression>
<**GROUP BY** *group-by-item* <,*group-by-item*>…>
<**HAVING** sql-expression>

See "SELECT Statement" on page 1044 for complete information on the SELECT statement.

Details

A table-expression is a SELECT statement. It is the fundamental building block of most SQL procedure statements. You can combine the results of multiple table-expressions with set operators, which creates a query-expression. Use one ORDER BY clause for an entire query-expression. Place a semicolon only at the end of the entire query-expression. A query-expression is often only one SELECT statement or table-expression.

Concepts

Using SAS Data Set Options with PROC SQL

PROC SQL can apply most of the SAS data set options, such as KEEP= and DROP=, to tables or SAS/ACCESS views. In the SQL procedure, SAS data set options that are separated by spaces are enclosed in parentheses, and they follow immediately after the table or SAS/ACCESS view name. You can also use SAS data set options on tables or SAS/ACCESS views listed in the FROM clause of a query. In the following PROC SQL

step, RENAME= renames LNAME to LASTNAME for the STAFF1 table. OBS= restricts the number of rows written to STAFF1 to 15:

```
proc sql;
   create table
          staff1(rename=(lname=lastname)) as
      select *
         from staff(obs=15);
```

You cannot use SAS data set options with DICTIONARY tables because DICTIONARY tables are read-only objects.

The only SAS data set options that you can use with PROC SQL views are those that assign and provide SAS passwords: READ=, WRITE=, ALTER=, and PW=.

See *SAS Language Reference: Dictionary* for a description of SAS data set options.

Connecting to a DBMS Using the SQL Procedure Pass-Through Facility

The SQL Procedure Pass-Through Facility enables you to send DBMS-specific SQL statements directly to a DBMS for execution. The Pass-Through Facility uses a SAS/ACCESS interface engine to connect to the DBMS. Therefore, you must have SAS/ACCESS software installed for your DBMS.

You submit SQL statements that are DBMS-specific. For example, you pass Transact-SQL statements to a SYBASE database. The Pass-Through Facility's basic syntax is the same for all the DBMSs. Only the statements that are used to connect to the DBMS and the SQL statements are DBMS-specific.

With the Pass-Through Facility, you can perform the following tasks:

- establish a connection with the DBMS using a CONNECT statement and terminate the connection with the DISCONNECT statement.
- send nonquery DBMS-specific SQL statements to the DBMS using the EXECUTE statement.
- retrieve data from the DBMS to be used in a PROC SQL query with the CONNECTION TO component in a SELECT statement's FROM clause.

You can use the Pass-Through Facility statements in a query, or you can store them in a PROC SQL view. When a view is stored, any options that are specified in the corresponding CONNECT statement are also stored. Thus, when the PROC SQL view is used in a SAS program, the SAS System can automatically establish the appropriate connection to the DBMS.

See "CONNECT Statement" on page 1033, "DISCONNECT Statement" on page 1040, "EXECUTE Statement" on page 1042, "CONNECTION TO" on page 1062, and your SAS/ACCESS documentation.

Return Codes

As you use PROC SQL statements that are available in the Pass-Through Facility, any errors are written to the SAS log. The return codes and messages that are generated by the Pass-Through Facility are available to you through the SQLXRC and SQLXMSG macro variables. Both macro variables are described in "Using Macro Variables Set by PROC SQL" on page 1096.

Connecting to a DBMS using the LIBNAME Statement

For many DBMSs, you can directly access DBMS data by assigning a libref to the DBMS using the SAS/ACCESS LIBNAME statement. Once you have associated a libref

with the DBMS, you can specify a DBMS table in a two-level SAS name and work with the table like any SAS data set. You can also embed the LIBNAME statement in a PROC SQL view (see "CREATE VIEW Statement" on page 1037).

PROC SQL will take advantage of the capabilities of a DBMS by passing it certain operations whenever possible. For example, before implementing a join, PROC SQL checks to see if the DBMS can do the join. If it can, PROC SQL passes the join to the DBMS. This increases performance by reducing data movement and translation. If the DBMS cannot do the join, PROC SQL processes the join. Using the SAS/ACCESS LIBNAME statement can often provide you with the performance benefits of the SQL Procedure Pass-Through Facility without having to write DBMS-specific code.

To use the SAS/ACCESS LIBNAME statement, you must have SAS/ACCESS installed for your DBMS. For more information on the SAS/ACCESS LIBNAME statement, refer to your SAS/ACCESS documentation.

Using Macro Variables Set by PROC SQL

PROC SQL sets up macro variables with certain values after it executes each statement. These macro variables can be tested inside a macro to determine whether to continue executing the PROC SQL step. SAS/AF software users can also test them in a program after an SQL SUBMIT block of code, using the SYMGET function.

After each PROC SQL statement has executed, the following macro variables are updated with these values:

SQLOBS
: contains the number of rows executed by an SQL procedure statement. For example, it contains the number of rows formatted and displayed in SAS output by a SELECT statement or the number of rows deleted by a DELETE statement.

SQLRC
: contains the following status values that indicate the success of the SQL procedure statement:

  0
  : PROC SQL statement completed successfully with no errors.

  4
  : PROC SQL statement encountered a situation for which it issued a warning. The statement continued to execute.

  8
  : PROC SQL statement encountered an error. The statement stopped execution at this point.

  12
  : PROC SQL statement encountered an internal error, indicating a bug in PROC SQL that should be reported to SAS Institute. These errors can occur only during compile time.

  16
  : PROC SQL statement encountered a user error. This error code is used, for example, when a subquery (that can only return a single value) evaluates to more than one row. These errors can only be detected during run time.

  24
  : PROC SQL statement encountered a system error. This error is used, for example, if the system cannot write to a PROC SQL table because the disk is full. These errors can occur only during run time.

28
PROC SQL statement encountered an internal error, indicating a bug in PROC SQL that should be reported to SAS Institute. These errors can occur only during run time.

SQLOOPS
contains the number of iterations that the inner loop of PROC SQL executes. The number of iterations increases proportionally with the complexity of the query. See also the description of the LOOPS option on page 1029.

SQLXRC
contains the DBMS-specific return code that is returned by the Pass-Through Facility.

SQLXMSG
contains descriptive information and the DBMS-specific return code for the error that is returned by the Pass-Through Facility.

This example retrieves the data but does not display them in SAS output because of the NOPRINT option in the PROC SQL statement. The %PUT macro statement displays the macro variables values.

```
proc sql noprint;
   select *
      from proclib.payroll;

%put sqlobs=**&sqlobs**
     sqloops=**&sqloops**
     sqlrc=**&sqlrc**;
```

The message in Output 34.5 on page 1097 appears in the SAS log and gives you the macros' values.

Output 34.5 PROC SQL Macro Variable Values

```
1   options ls=80;
2   proc sql noprint;
3      select *
4         from proclib.payroll;
5
6   %put sqlobs=**&sqlobs**
7        sqloops=**&sqloops**
8        sqlrc=**&sqlrc**;
sqlobs=**1**        sqloops=**11**        sqlrc=**0**
```

Updating PROC SQL and SAS/ACCESS Views

You can update PROC SQL and SAS/ACCESS views using the INSERT, DELETE, and UPDATE statements, under the following conditions.

- If the view accesses a DBMS table, you must have been granted the appropriate authorization by the external database management system (for example, DB2). You must have installed the SAS/ACCESS software for your DBMS. See the SAS/ACCESS interface guide for your DBMS for more information on SAS/ACCESS views.

- You can update only a single table through a view. The table cannot be joined to another table or linked to another table with a set-operator. The view cannot contain a subquery.
- You can update a column in a view using the column's alias, but you cannot update a derived column, that is, a column produced by an expression. In the following example, you can update the column SS, but not WeeklySalary.

```
create view EmployeeSalaries as
   select Employee, SSNumber as SS,
          Salary/52 as WeeklySalary
          from employees;
```

- You cannot update a view containing an ORDER BY.

PROC SQL and the ANSI Standard

PROC SQL follows most of the guidelines set by the American National Standards Institute (ANSI) in its implementation of SQL. However, it is not fully compliant with the current ANSI Standard for SQL.*

The SQL research project at SAS Institute has focused primarily on the expressive power of SQL as a query language. Consequently, some of the database features of SQL have not yet been implemented in the SAS System.

This section describes

- enhancements to SQL that SAS Institute has made through PROC SQL
- the ways in which PROC SQL differs from the current ANSI Standard for SQL.

SQL Procedure Enhancements

Most of the enhancements described here are required by the current ANSI Standard.

Reserved Words

PROC SQL reserves very few keywords and then only in certain contexts. The ANSI Standard reserves all SQL keywords in all contexts. For example, according to the Standard you cannot name a column GROUP because of the keywords GROUP BY.

The following words are reserved in PROC SQL:

- The keyword CASE is always reserved; its use in the CASE expression (an SQL2 feature) precludes its use as a column name.

  If you have a column named CASE in a table and you want to specify it in a PROC SQL step, you can use the SAS data set option RENAME= to rename that column for the duration of the query. You can also surround CASE in double quotes ("CASE") and set the PROC SQL option DQUOTE=ANSI.
- The keywords AS, ON, FULL, JOIN, LEFT, FROM, WHEN, WHERE, ORDER, GROUP, RIGHT, INNER, OUTER, UNION, EXCEPT, HAVING, and INTERSECT cannot normally be used for table aliases. These keywords all introduce clauses that appear after a table name. Since the alias is optional, PROC SQL deals with this ambiguity by assuming that any one of these words introduces the corresponding clause and is not the alias. If you want to use one of these keywords as an alias, use the PROC SQL option DQUOTE=ANSI.

* International Organization for Standardization (ISO): *Database SQL*. Document ISO/IEC 9075:1992. Also available as American National Standards Institute (ANSI) Document ANSI X3.135-1992.

- The keyword USER is reserved for the current userid. If you have a column named USER in a table and you want to specify it in a PROC SQL step, you can use the SAS data set option RENAME= to rename that column for the duration of the query. You can also surround USER in double quotes ("USER") and set the PROC SQL option DQUOTE=ANSI.

Column Modifiers

PROC SQL supports the SAS System's INFORMAT=, FORMAT=, and LABEL= modifiers for expressions within the SELECT clause. These modifiers control the format in which output data are displayed and labeled.

Alternate Collating Sequences

PROC SQL allows you to specify an alternate collating (sorting) sequence to be used when you specify the ORDER BY clause. See the description of the SORTSEQ= option in "PROC SQL Statement" on page 1027 for more information.

ORDER BY Clause in a View Definition

PROC SQL permits you to specify an ORDER BY clause in a CREATE VIEW statement. When the view is queried, its data are always sorted according to the specified order unless a query against that view includes a different ORDER BY clause. See "CREATE VIEW Statement" on page 1037 for more information.

In-Line Views

The ability to code nested query-expressions in the FROM clause is a requirement of the ANSI Standard. PROC SQL supports such nested coding.

Outer Joins

The ability to include columns that both match and do not match in a join-expression is a requirement of the ANSI Standard. PROC SQL supports this ability.

Arithmetic Operators

PROC SQL supports the SAS System exponentiation (**) operator. PROC SQL uses the notation < > to mean not equal.

Orthogonal Expressions

PROC SQL permits the combination of comparison, Boolean, and algebraic expressions. For example, (X=3)*7 yields a value of 7 if X=3 is true because true is defined to be 1. If X=3 is false, it resolves to 0 and the entire expression yields a value of 0.

PROC SQL permits a subquery in any expression. This feature is required by the ANSI Standard. Therefore, you can have a subquery on the left side of a comparison operator in the WHERE expression.

PROC SQL permits you to order and group data by any kind of mathematical expression (except those including summary functions) using ORDER BY and GROUP BY clauses. You can also group by an expression that appears on the SELECT clause by using the integer that represents the expression's ordinal position in the SELECT clause. You are not required to select the expression by which you are grouping or

ordering. See ORDER BY Clause on page 1053 and GROUP BY Clause on page 1052 for more information.

Set Operators

The set operators UNION, INTERSECT, and EXCEPT are required by the ANSI Standard. PROC SQL provides these operators plus the OUTER UNION operator.

The ANSI Standard also requires that the tables being operated upon all have the same number of columns with matching data types. The SQL procedure works on tables that have the same number of columns, as well as on those that do not, by creating virtual columns so that a query can evaluate correctly. See "query-expression" on page 1075 for more information.

Statistical Functions

PROC SQL supports many more summary functions than required by the ANSI Standard for SQL.

PROC SQL supports the remerging of summary function results into the table's original data. For example, computing the percentage of total is achieved with 100*x/SUM(x) in PROC SQL. See "summary-function" on page 1088 for more information on the available summary functions and remerging data.

SAS System Functions

PROC SQL supports all the functions available to the SAS DATA step, except for LAG, DIF, and SOUND. Other SQL databases support their own set of functions.

SQL Procedure Omissions

PROC SQL differs from the ANSI Standard for SQL in the following ways.

COMMIT Statement

The COMMIT statement is not supported.

ROLLBACK Statement

The ROLLBACK statement is not supported. The UNDO_POLICY= option in the PROC SQL statement addresses rollback. See the description of the UNDO_POLICY= option in "PROC SQL Statement" on page 1027 for more information.

Identifiers and Naming Conventions

In the SAS System, table names, column names, and aliases are limited to 32 characters and can contain mixed case. For more information on SAS naming conventions, see *SAS Language Reference: Dictionary*. The ANSI Standard for SQL allows longer names.

Granting User Privileges

The GRANT statement, PRIVILEGES keyword, and authorization-identifier features of SQL are not supported. You may want to use operating environment-specific means of security instead.

Three-Valued Logic

ANSI-compatible SQL has three-valued logic, that is, special cases for handling comparisons involving NULL values. Any value compared with a NULL value evaluates to NULL.

PROC SQL follows the SAS System convention for handling missing values: when numeric NULL values are compared to non-NULL numbers, the NULL values are less than or smaller than all the non-NULL values; when character NULL values are compared to non-NULL characters, the character NULL values are treated as a string of blanks.

Embedded SQL

Currently there is no provision for embedding PROC SQL statements in other SAS programming environments, such as the DATA step or SAS/IML software.

Examples

Example 1: Creating a Table and Inserting Data into It

Procedure features:

CREATE TABLE statement

column-modifier

INSERT statement

VALUES clause

SELECT clause

FROM clause

Table: PROCLIB.PAYLIST

This example creates the table PROCLIB.PAYLIST and inserts data into it.

Program

```
libname proclib 'SAS-data-library';


options nodate pageno=1 linesize=80 pagesize=40;
```

The CREATE TABLE statement creates PROCLIB.PAYLIST with six empty columns. Each column definition indicates whether the column is character or numeric. The number in parentheses specifies the width of the column. INFORMAT= and FORMAT= assign date informats and formats to the Birth and Hired columns.

```
proc sql;
   create table proclib.paylist
       (IdNum char(4),
        Gender char(1),
        Jobcode char(3),
        Salary num,
        Birth num informat=date7.
                  format=date7.,
        Hired num informat=date7.
                  format=date7.);
```

The INSERT statement inserts data values into PROCLIB.PAYLIST according to the position in the VALUES clause. Therefore, in the first VALUES clause, **1639** is inserted into the first column, **F** into the second column, and so forth. Dates in SAS are stored as integers with 0 equal to January 1, 1960. Suffixing the date with a **d** is one way to use the internal value for dates.

```
insert into proclib.paylist
    values('1639','F','TA1',42260,'26JUN70'd,'28JAN91'd)
    values('1065','M','ME3',38090,'26JAN54'd,'07JAN92'd)
    values('1400','M','ME1',29769.'05NOV67'd,'16OCT90'd)
```

The value **null** represents a missing value for the character column Jobcode. The period represents a missing value for the numeric column Salary.

```
    values('1561','M',null,36514,'30NOV63'd,'07OCT87'd)
    values('1221','F','FA3',.,'22SEP63'd,'04OCT94'd);
```

The SELECT clause selects columns from PROCLIB.PAYLIST. The asterisk (*) selects all columns. The FROM clause specifies PROCLIB.PAYLIST as the table to select from.

```
title 'PROCLIB.PAYLIST Table';
select *
   from proclib.paylist;
```

Output Table

PROCLIB.PAYLIST

```
                        PROCLIB.PAYLIST Table                          1

             Id
             Num   Gender  Jobcode    Salary    Birth    Hired
             -------------------------------------------------
             1639  F       TA1         42260  26JUN70  28JAN91
             1065  M       ME3         38090  26JAN54  07JAN92
             1400  M       ME1         29769  05NOV67  16OCT90
             1561  M                   36514  30NOV63  07OCT87
             1221  F       FA3             .  22SEP63  04OCT94
```

Example 2: Creating a Table from a Query's Result

Procedure features:

CREATE TABLE statement

AS query-expression

SELECT clause

column alias
FORMAT= column-modifier
object-item

Other features:

data set option

OBS=

Tables:

PROCLIB.PAYROLL, PROCLIB.BONUS

This example builds a column with an arithmetic expression and creates the PROCLIB.BONUS table from the query's result.

Input Table

PROCLIB.PAYROLL (Partial Listing)

PROCLIB.PAYROLL
First 10 Rows Only

Id Number	Sex	Jobcode	Salary	Birth	Hired
1919	M	TA2	34376	12SEP60	04JUN87
1653	F	ME2	35108	15OCT64	09AUG90
1400	M	ME1	29769	05NOV67	16OCT90
1350	F	FA3	32886	31AUG65	29JUL90
1401	M	TA3	38822	13DEC50	17NOV85
1499	M	ME3	43025	26APR54	07JUN80
1101	M	SCP	18723	06JUN62	01OCT90
1333	M	PT2	88606	30MAR61	10FEB81
1402	M	TA2	32615	17JAN63	02DEC90
1479	F	TA3	38785	22DEC68	05OCT89

Program

```
libname proclib 'SAS-data-library';

options nodate pageno=1 linesize=80 pagesize=40;
```

The CREATE TABLE statement creates the table PROCLIB.BONUS from the result of the subsequent query.

```
proc sql;
   create table proclib.bonus as
```

The SELECT clause specifies that three columns will be in the new table: IdNumber, Salary, and Bonus. FORMAT= assigns the DOLLAR8. format to Salary. The Bonus column is built with the SQL expression **`salary*.025`**.

```
    select IdNumber, Salary format=dollar8.,
           salary*.025 as Bonus format=dollar8.
      from proclib.payroll;
```

The SELECT clause selects columns from PROCLIB.BONUS. The asterisk (*) selects all columns. The FROM clause specifies PROCLIB.BONUS as the table to select from. The OBS= data set option limits the printing of the output to 10 rows.

```
    title 'BONUS Information';
    select *
      from proclib.bonus(obs=10);
```

Output

PROCLIB.BONUS

```
                          BONUS Information                              1

                      Id
                      Number    Salary     Bonus
                      -------------------------
                      1919     $34,376      $859
                      1653     $35,108      $878
                      1400     $29,769      $744
                      1350     $32,886      $822
                      1401     $38,822      $971
                      1499     $43,025    $1,076
                      1101     $18,723      $468
                      1333     $88,606    $2,215
                      1402     $32,615      $815
                      1479     $38,785      $970
```

Example 3: Updating Data in a PROC SQL Table

Procedure features:

ALTER TABLE statement

DROP clause

MODIFY clause
UPDATE statement
SET clause
CASE expression

Table: EMPLOYEES

This example updates data values in the EMPLOYEES table and drops a column.

Input

```
data Employees;
   input IdNum $4. +2 LName $11. FName $11. JobCode $3.
         +1 Salary 5. +1 Phone $12.;
   datalines;
1876  CHIN       JACK       TA1 42400 212/588-5634
1114  GREENWALD  JANICE     ME3 38000 212/588-1092
1556  PENNINGTON MICHAEL    ME1 29860 718/383-5681
1354  PARKER     MARY       FA3 65800 914/455-2337
1130  WOOD       DEBORAH    PT2 36514 212/587-0013
;
```

Program

```
options nodate pageno=1 linesize=80 pagesize=40;
```

The SELECT clause displays the table before the updates. The asterisk (*) selects all columns for display.The FROM clause specifies EMPLOYEES as the table to select from.

```
proc sql;
   title 'Employees Table';
   select * from Employees;
```

The UPDATE statement updates the values in EMPLOYEES. The SET clause specifies that the data in the Salary column be multiplied by 1.04 when the job code ends with a **1** and 1.025 for all other job codes. (The two underscores represent any character.) The CASE expression returns a value for each row that completes the SET clause.

```
update employees
      set salary=salary*
      case when jobcode like '__1' then 1.04
           else 1.025
      end;
```

The ALTER TABLE statement specifies EMPLOYEES as the table to alter. The MODIFY clause permanently modifies the format of the Salary column. The DROP clause permanently drops the Phone column.

```
alter table employees
   modify salary num format=dollar8.
   drop phone;
```

The SELECT clause displays the EMPLOYEES table after the updates. The asterisk (*) selects all columns.

```
title 'Updated Employees Table';
select * from employees;
```

Output

```
                          Employees Table                                1

Id                                    Job
Num   LName         FName         Code    Salary  Phone
-------------------------------------------------------------
1876  CHIN          JACK          TA1      42400  212/588-5634
1114  GREENWALD     JANICE        ME3      38000  212/588-1092
1556  PENNINGTON    MICHAEL       ME1      29860  718/383-5681
1354  PARKER        MARY          FA3      65800  914/455-2337
1130  WOOD          DEBORAH       PT2      36514  212/587-0013
```

```
                     Updated Employees Table                           2

Id                                    Job
Num   LName         FName         Code    Salary
---------------------------------------------
1876  CHIN          JACK          TA1    $44,096
1114  GREENWALD     JANICE        ME3    $38,950
1556  PENNINGTON    MICHAEL       ME1    $31,054
1354  PARKER        MARY          FA3    $67,445
1130  WOOD          DEBORAH       PT2    $37,427
```

Example 4: Joining Two Tables

Procedure features:

FROM clause
    table alias
inner join
joined-table component
PROC SQL statement option
    NUMBER
WHERE clause
    IN condition

Tables: PROCLIB.STAFF, PROCLIB.PAYROLL

This example joins two tables in order to to get more information about data that are common to both tables.

Input Tables

PROCLIB.STAFF (Partial Listing)

PROCLIB.STAFF
First 10 Rows Only

Id Num	Lname	Fname	City	State	Hphone
1919	ADAMS	GERALD	STAMFORD	CT	203/781-1255
1653	ALIBRANDI	MARIA	BRIDGEPORT	CT	203/675-7715
1400	ALHERTANI	ABDULLAH	NEW YORK	NY	212/586-0808
1350	ALVAREZ	MERCEDES	NEW YORK	NY	718/383-1549
1401	ALVAREZ	CARLOS	PATERSON	NJ	201/732-8787
1499	BAREFOOT	JOSEPH	PRINCETON	NJ	201/812-5665
1101	BAUCOM	WALTER	NEW YORK	NY	212/586-8060
1333	BANADYGA	JUSTIN	STAMFORD	CT	203/781-1777
1402	BLALOCK	RALPH	NEW YORK	NY	718/384-2849
1479	BALLETTI	MARIE	NEW YORK	NY	718/384-8816

PROCLIB.PAYROLL (Partial Listing)

PROCLIB.PAYROLL
First 10 Rows Only

Id Number	Sex	Jobcode	Salary	Birth	Hired
1919	M	TA2	34376	12SEP60	04JUN87
1653	F	ME2	35108	15OCT64	09AUG90
1400	M	ME1	29769	05NOV67	16OCT90
1350	F	FA3	32886	31AUG65	29JUL90
1401	M	TA3	38822	13DEC50	17NOV85
1499	M	ME3	43025	26APR54	07JUN80
1101	M	SCP	18723	06JUN62	01OCT90
1333	M	PT2	88606	30MAR61	10FEB81
1402	M	TA2	32615	17JAN63	02DEC90
1479	F	TA3	38785	22DEC68	05OCT89

Program

```
libname proclib 'SAS-data-library';
```

```
options nodate pageno=1 linesize=120 pagesize=40;
```

NUMBER adds a column that contains the row number.

```
proc sql number;
```

The SELECT clause selects the columns to output.

```
     title 'Information for Certain Employees Only';
     select Lname, Fname, City, State,
            IdNumber, Salary, Jobcode
```

The FROM clause lists the tables to select from.

```
         from proclib.staff, proclib.payroll
```

The WHERE clause specifies that the tables are joined on the ID number from each table. WHERE also further subsets the query with the IN condition, which returns rows for only four employees.

```
         where idnumber=idnum and idnum in
                ('1919','1400', '1350', '1333');
```

Output

```
                Information for Certain Employees Only                  1

                                                                     Id
   Row  Lname            Fname            City             State  Number
          Salary  Jobcode
-------------------------------------------------------------------------
     1  ADAMS            GERALD           STAMFORD         CT     1919
           34376  TA2

     2  ALHERTANI        ABDULLAH         NEW YORK         NY     1400
           29769  ME1

     3  ALVAREZ          MERCEDES         NEW YORK         NY     1350
           32886  FA3

     4  BANADYGA         JUSTIN           STAMFORD         CT     1333
           88606  PT2
```

Example 5: Combining Two Tables

Procedure features:

DELETE statement
IS condition
RESET statement option
    DOUBLE
UNION set operator

Tables: PROCLIB.NEWPAY, PROCLIB.PAYLIST, PROCLIB.PAYLIST2

This example creates a new table, PROCLIB.NEWPAY, by concatenating two other tables: PROCLIB.PAYLIST and PROCLIB.PAYLIST2.

Input Tables

PROCLIB.PAYLIST

PROCLIB.PAYLIST Table

Id Num	Gender	Jobcode	Salary	Birth	Hired
1639	F	TA1	42260	26JUN70	28JAN91
1065	M	ME3	38090	26JAN54	07JAN92
1400	M	ME1	29769	05NOV67	16OCT90
1561	M		36514	30NOV63	07OCT87
1221	F	FA3	.	22SEP63	04OCT94

PROCLIB.PAYLIST2

PROCLIB.PAYLIST2 Table

Id Num	Gender	Jobcode	Salary	Birth	Hired
1919	M	TA2	34376	12SEP66	04JUN87
1653	F	ME2	31896	15OCT64	09AUG92
1350	F	FA3	36886	31AUG55	29JUL91
1401	M	TA3	38822	13DEC55	17NOV93
1499	M	ME1	23025	26APR74	07JUN92

Program

```
libname proclib 'SAS-data-library';

options nodate pageno=1 linesize=80 pagesize=60;
```

The SELECT clauses select all the columns from the tables listed in the FROM clauses. The UNION set operator concatenates the query results that are produced by the two SELECT clauses. UNION orders the result by IdNum.

```
proc sql;
   create table proclib.newpay as
      select * from proclib.paylist
      union
      select * from proclib.paylist2;
```

The DELETE statement deletes rows from PROCLIB.NEWPAY that satisfy the WHERE expression. The IS condition specifies rows that contain missing values in the Jobcode or Salary column.

```
   delete
      from proclib.newpay
      where jobcode is missing or salary is missing;
```

RESET changes the procedure environment without stopping and restarting PROC SQL. The DOUBLE option double-spaces the output. (The DOUBLE option has no effect on ODS output.) The SELECT clause selects all columns from the newly created table, PROCLIB.NEWPAY.

```
   reset double;
   title 'Personnel Data';
   select *
      from proclib.newpay;
```

Output

```
                          Personnel Data                                1

             Id
             Num   Gender  Jobcode    Salary    Birth    Hired
             -------------------------------------------------
             1065  M       ME3         38090  26JAN54  07JAN92

             1350  F       FA3         36886  31AUG55  29JUL91

             1400  M       ME1         29769  05NOV67  16OCT90

             1401  M       TA3         38822  13DEC55  17NOV93

             1499  M       ME1         23025  26APR74  07JUN92

             1639  F       TA1         42260  26JUN70  28JAN91

             1653  F       ME2         31896  15OCT64  09AUG92

             1919  M       TA2         34376  12SEP66  04JUN87
```

Example 6: Reporting from DICTIONARY Tables

Procedure features:
DESCRIBE TABLE statement
DICTIONARY.*table-name* component
Table: DICTIONARY.MEMBERS

This example uses DICTIONARY tables to show a list of the SAS files in a SAS data library. If you do not know the names of the columns in the DICTIONARY table that you are querying, use a DESCRIBE TABLE statement with the table.

Program

```
libname proclib 'SAS-data-library';
```

SOURCE writes the programming statements to the SAS log.

```
options nodate pageno=1 source linesize=80 pagesize=60;
```

DESCRIBE TABLE writes the column names from DICTIONARY.MEMBERS to the SAS log.

```
proc sql;
   describe table dictionary.members;
```

The SELECT clause selects the MEMNAME and MEMTYPE columns. The FROM clause specifies DICTIONARY.MEMBERS as the table to select from. The WHERE clause subsets the output to include only those rows that have a libref of *PROCLIB* in the LIBNAME column.

```
   title 'SAS Files in the PROCLIB Library';
   select memname, memtype
      from dictionary.members
      where libname='PROCLIB';
```

Log

```
2  options nodate pageno=1 source linesize=80 pagesize=60;
3  proc sql;
4     describe table dictionary.members;
NOTE: SQL table DICTIONARY.MEMBERS was created like:

create table DICTIONARY.MEMBERS
  (
   libname char(8) label='Library Name',
   memname char(32) label='Member Name',
   memtype char(8) label='Member Type',
   engine char(8) label='Engine Name',
   index char(32) label='Indexes',
   path char(1024) label='Path Name'
  );

5     title 'SAS Files in the PROCLIB Library';
6     select memname, memtype
7        from dictionary.members
8        where libname='PROCLIB';
```

Output

```
                    SAS Files in the PROCLIB Library                    1

                                          Member
                Member Name               Type
                ----------------------------------------
                BONUS                     DATA
                BONUS95                   DATA
                DELAY                     DATA
                HOUSES                    DATA
                INTERNAT                  DATA
                JOBS                      VIEW
                MARCH                     DATA
                NEWPAY                    DATA
                PAYDATA                   VIEW
                PAYINFO                   VIEW
                PAYLIST                   DATA
                PAYLIST2                  DATA
                PAYROLL                   DATA
                PAYROLL2                  DATA
                SCHEDULE                  DATA
                SCHEDULE2                 DATA
                STAFF                     DATA
                STAFF2                    DATA
                SUPERV                    DATA
                SUPERV2                   DATA
```

Example 7: Performing an Outer Join

Procedure features:

joined-table component
left outer join
SELECT clause

COALESCE function
WHERE clause
CONTAINS condition
Tables: PROCLIB.PAYROLL, PROCLIB.PAYROLL2

This example illustrates a left outer join of the PROCLIB.PAYROLL and PROCLIB.PAYROLL2 tables.

Input Tables

PROCLIB.PAYROLL (Partial Listing)

PROCLIB.PAYROLL
First 10 Rows Only

Id Number	Sex	Jobcode	Salary	Birth	Hired
1009	M	TA1	28880	02MAR59	26MAR92
1017	M	TA3	40858	28DEC57	16OCT81
1036	F	TA3	39392	19MAY65	23OCT84
1037	F	TA1	28558	10APR64	13SEP92
1038	F	TA1	26533	09NOV69	23NOV91
1050	M	ME2	35167	14JUL63	24AUG86
1065	M	ME2	35090	26JAN44	07JAN87
1076	M	PT1	66558	14OCT55	03OCT91
1094	M	FA1	22268	02APR70	17APR91
1100	M	BCK	25004	01DEC60	07MAY88

PROCLIB.PAYROLL2

PROCLIB.PAYROLL2

Id Num	Sex	Jobcode	Salary	Birth	Hired
1036	F	TA3	42465	19MAY65	23OCT84
1065	M	ME3	38090	26JAN44	07JAN87
1076	M	PT1	69742	14OCT55	03OCT91
1106	M	PT3	94039	06NOV57	16AUG84
1129	F	ME3	36758	08DEC61	17AUG91
1221	F	FA3	29896	22SEP67	04OCT91
1350	F	FA3	36098	31AUG65	29JUL90
1369	M	TA3	36598	28DEC61	13MAR87
1447	F	FA1	22123	07AUG72	29OCT92
1561	M	TA3	36514	30NOV63	07OCT87
1639	F	TA3	42260	26JUN57	28JAN84
1998	M	SCP	23100	10SEP70	02NOV92

Program

```
libname proclib 'SAS-data-library';


options nodate pageno=1 linesize=80 pagesize=60;
```

OUTOBS= limits the output to 10 rows. The SELECT clause lists the columns to select. Some column names are prefixed with a table alias because they are in both tables. LABEL= and FORMAT= are column modifiers.

```
proc sql outobs=10;
   title 'Most Current Jobcode and Salary Information';
   select p.IdNumber, p.Jobcode, p.Salary,
          p2.jobcode label='New Jobcode',
          p2.salary label='New Salary' format=dollar8.
```

The FROM clause lists the tables to join and assigns table aliases. The keywords LEFT JOIN specify the type of join. The order of the tables in the FROM clause is important. PROCLIB.PAYROLL is listed first and is considered the "left" table, PROCLIB.PAYROLL2 is the "right" table.

```
       from proclib.payroll as p left join proclib.payroll2 as p2
```

The ON clause specifies that the join be performed based on the values of the ID numbers from each table.

```
       on p.IdNumber=p2.idnum;
```

Output

As the output shows, all rows from the left table, PROCLIB.PAYROLL, are returned. PROC SQL assigns missing values for rows in the left table, PAYROLL, that have no matching values for IdNum in PAYROLL2.

```
                Most Current Jobcode and Salary Information                1

                Id                          New           New
                Number  Jobcode    Salary  Jobcode    Salary
                ------------------------------------------
                1009    TA1         28880                 .
                1017    TA3         40858                 .
                1036    TA3         39392  TA3       $42,465
                1037    TA1         28558                 .
                1038    TA1         26533                 .
                1050    ME2         35167                 .
                1065    ME2         35090  ME3       $38,090
                1076    PT1         66558  PT1       $69,742
                1094    FA1         22268                 .
                1100    BCK         25004                 .
```

The SELECT clause lists the columns to select. COALESCE overlays the like-named columns. For each row, COALESCE returns the first nonmissing value of either P2.JOBCODE or P.JOBCODE. Because P2.JOBCODE is the first argument, if there is a nonmissing value for P2.JOBCODE, COALESCE returns that value. Thus, the output contains the most recent jobcode information for every employee. LABEL= assigns a column label.

```
title 'Most Current Jobcode and Salary Information';
select p.idnumber, coalesce(p2.jobcode,p.jobcode)
     label='Current Jobcode',
```

For each row, COALESCE returns the first nonmissing value of either P2.SALARY or P.SALARY. Because P2.SALARY is the first argument, if there is a nonmissing value for P2.SALARY, COALESCE returns that value. Thus, the output contains the most recent salary information for every employee.

```
        coalesce(p2.salary,p.salary) label='Current Salary'
                format=dollar8.
```

The FROM clause lists the tables to join and assigns table aliases. The keywords LEFT JOIN specify the type of join. The ON clause specifies that the join is based on the ID numbers from each table.

```
    from proclib.payroll p left join proclib.payroll2 p2
    on p.IdNumber=p2.idnum;
```

Output

```
                Most Current Jobcode and Salary Information                1

                        Id      Current   Current
                        Number  Jobcode    Salary
                        -------------------------
                        1009    TA1       $28,880
                        1017    TA3       $40,858
                        1036    TA3       $42,465
                        1037    TA1       $28,558
                        1038    TA1       $26,533
                        1050    ME2       $35,167
                        1065    ME3       $38,090
                        1076    PT1       $69,742
                        1094    FA1       $22,268
                        1100    BCK       $25,004
```

The WHERE clause subsets the left join to include only those rows containing the value **TA**.

```
title 'Most Current Information for Ticket Agents';
select p.IdNumber,
```

```
    coalesce(p2.jobcode,p.jobcode) label='Current Jobcode',
    coalesce(p2.salary,p.salary) label='Current Salary'
from proclib.payroll p left join proclib.payroll2 p2
on p.IdNumber=p2.idnum
where p2.jobcode contains 'TA';
```

Output

```
                Most Current Information for Ticket Agents                1

                        Id      Current   Current
                        Number  Jobcode    Salary
                        -------------------------
                        1036    TA3         42465
                        1369    TA3         36598
                        1561    TA3         36514
                        1639    TA3         42260
```

Example 8: Creating a View from a Query's Result

Procedure features:

CREATE VIEW statement
GROUP BY clause
SELECT clause
    COUNT function
HAVING clause

Other features:

AVG summary function
data set option
    PW=

Tables: PROCLIB.PAYROLL, PROCLIB.JOBS

This example creates the PROC SQL view PROCLIB.JOBS from the result of a query-expression.

Input Table

PROCLIB.PAYROLL (Partial Listing)

PROCLIB.PAYROLL
First 10 Rows Only

Id Number	Sex	Jobcode	Salary	Birth	Hired
1009	M	TA1	28880	02MAR59	26MAR92
1017	M	TA3	40858	28DEC57	16OCT81
1036	F	TA3	39392	19MAY65	23OCT84
1037	F	TA1	28558	10APR64	13SEP92
1038	F	TA1	26533	09NOV69	23NOV91
1050	M	ME2	35167	14JUL63	24AUG86
1065	M	ME2	35090	26JAN44	07JAN87
1076	M	PT1	66558	14OCT55	03OCT91
1094	M	FA1	22268	02APR70	17APR91
1100	M	BCK	25004	01DEC60	07MAY88

Program

```
libname proclib 'SAS-data-library';


options nodate pageno=1 linesize=80 pagesize=60;
```

CREATE VIEW creates the PROC SQL view PROCLIB.JOBS. The PW= data set option assigns password protection to the data generated by this view.

```
proc sql;
   create view proclib.jobs(pw=red) as
```

The SELECT clause specifies four columns for the view: Jobcode and three columns, Number, AVGAGE, and AVGSAL, whose values are the products of functions. COUNT returns the number of nonmissing values for each jobcode because the data are grouped by Jobcode. LABEL= assigns a label to the column.

```
      select Jobcode,
             count(jobcode) as number label='Number',
```

The AVG summary function calculates the average age and average salary for each jobcode.

```
             avg(int((today()-birth)/365.25)) as avgage
                format=2. label='Average Age',
             avg(salary) as avgsal
                format=dollar8. label='Average Salary'
```

The FROM clause specifies PAYROLL as the table to select from. PROC SQL assumes the libref of PAYROLL to be PROCLIB because PROCLIB is used in the CREATE VIEW statement.

```
from payroll
```

The GROUP BY clause groups the data by the values of Jobcode. Thus, any summary statistics are calculated for each grouping of rows by value of Jobcode. The HAVING clause subsets the grouped data and returns rows for job codes that contain an average age of greater than or equal to 30.

```
group by jobcode
having avgage ge 30;
```

The SELECT statement selects all columns from PROCLIB.JOBS. PW=RED is necessary because the view is password-protected.

```
title 'Current Summary Information for Each Job Category';
title2 'Average Age Greater Than Or Equal to 30';
select * from proclib.jobs(pw=red);
```

Output

```
Current Summary Information for Each Job Category                1
     Average Age Greater Than Or Equal to 30

                          Average   Average
Jobcode    Number             Age    Salary
-----------------------------------------
BCK             9              33   $25,794
FA2            16              34   $27,987
FA3             7              35   $32,934
ME1             8              30   $28,500
ME2            14              36   $35,577
ME3             7              39   $42,411
NA2             3              38   $52,383
PT1             8              34   $67,908
PT2            10              39   $87,925
PT3             2              50   $10,505
SCP             7              34   $18,309
TA1             9              32   $27,721
TA2            20              33   $33,575
TA3            12              37   $39,680
```

Example 9: Joining Three Tables

Procedure features:

FROM clause

joined-table component

WHERE clause

Tables: PROCLIB.STAFF2, PROCLIB.SCHEDULE2, PROCLIB.SUPERV2

This example joins three tables and produces a report that contains columns from each table.

Input Tables

PROCLIB.STAFF2

PROCLIB.STAFF2

Id Num	Lname	Fname	City	State	Hphone
1106	MARSHBURN	JASPER	STAMFORD	CT	203/781-1457
1430	DABROWSKI	SANDRA	BRIDGEPORT	CT	203/675-1647
1118	DENNIS	ROGER	NEW YORK	NY	718/383-1122
1126	KIMANI	ANNE	NEW YORK	NY	212/586-1229
1402	BLALOCK	RALPH	NEW YORK	NY	718/384-2849
1882	TUCKER	ALAN	NEW YORK	NY	718/384-0216
1479	BALLETTI	MARIE	NEW YORK	NY	718/384-8816
1420	ROUSE	JEREMY	PATERSON	NJ	201/732-9834
1403	BOWDEN	EARL	BRIDGEPORT	CT	203/675-3434
1616	FUENTAS	CARLA	NEW YORK	NY	718/384-3329

PROCLIB.SCHEDULE2

PROCLIB.SCHEDULE2

Flight	Date	Dest	Id Num
132	01MAR94	BOS	1118
132	01MAR94	BOS	1402
219	02MAR94	PAR	1616
219	02MAR94	PAR	1478
622	03MAR94	LON	1430
622	03MAR94	LON	1882
271	04MAR94	NYC	1430
271	04MAR94	NYC	1118
579	05MAR94	RDU	1126
579	05MAR94	RDU	1106

PROCLIB.SUPERV2

PROCLIB.SUPERV2

Supervisor Id	State	Job Category
1417	NJ	NA
1352	NY	NA
1106	CT	PT
1442	NJ	PT
1118	NY	PT
1405	NJ	SC
1564	NY	SC
1639	CT	TA
1126	NY	TA
1882	NY	ME

Program

```
libname proclib 'SAS-data-library';


options nodate pageno=1 linesize=80 pagesize=60;
```

The SELECT clause specifies the columns to select. IdNum is prefixed with a table alias because it appears in two tables.

```
   proc sql;
title 'All Flights for Each Supervisor';
   select s.IdNum, Lname, City 'Hometown', Jobcat,
          Flight, Date
```

The FROM clause lists the three tables for the join and assigns an alias to each table.

```
       from proclib.schedule2 s, proclib.staff2 t, proclib.superv2 v
```

The WHERE clause specifies the columns that join the tables. The STAFF2 and SCHEDULE2 tables have an IdNum column, which has related values in both tables. The STAFF2 and SUPERV2 tables have the IdNum and SUPID columns, which have related values in both tables.

```
       where s.idnum=t.idnum and t.idnum=v.supid;
```

Output

```
                    All Flights for Each Supervisor                        1

      Id                                        Job
      Num   Lname            Hometown          Category  Flight     Date
      ------------------------------------------------------------------
      1106  MARSHBURN        STAMFORD          PT        579      05MAR94
      1118  DENNIS           NEW YORK          PT        132      01MAR94
      1118  DENNIS           NEW YORK          PT        271      04MAR94
      1126  KIMANI           NEW YORK          TA        579      05MAR94
      1882  TUCKER           NEW YORK          ME        622      03MAR94
```

Example 10: Querying an In-Line View

Procedure features:
FROM clause
in-line view
Tables: PROCLIB.STAFF, PROCLIB.SCHEDULE, PROCLIB.SUPERV

This example uses the query explained in Example 9 on page 1118 as an in-line view. The example also shows how to rename columns with an in-line view.

Program

```
libname proclib 'SAS-data-library';


options nodate pageno=1 linesize=80 pagesize=60;
```

The SELECT clause selects all columns returned by the query in the FROM clause.

```
proc sql outobs=10;
   title 'All Flights for Each Supervisor';
   select *
```

The query that joins the three tables is used in the FROM clause instead of the name of a table or view. In the in-line query, the SELECT clause lists the columns to select. IdNum is prefixed with a table alias because it appears in two tables. The FROM clause lists the three tables for the join and assigns an alias to each table. The WHERE clause specifies the columns that join the tables. The STAFF2 and SCHEDULE2 tables have an IdNum column, which has related values in both tables. The STAFF2 and SUPERV2 tables have the IdNum and SUPID columns, which have related values in both tables.

```
      from (select lname, s.idnum, city, jobcat,
                   flight, date
```

```
from proclib.schedule2 s, proclib.staff2 t,
     proclib.superv2 v
where s.idnum=t.idnum and t.idnum=v.supid)
```

The alias THREE refers to the entire query. The names in parentheses become the names for the columns in the output. The label **Job Category** appears in the output instead of the name Jobtype because PROC SQL prints a column's label if the column has a label.

```
as three (Surname, Emp_ID, Hometown,
          Jobtype, FlightNumber, FlightDate);
```

Output

All Flights for Each Supervisor 1

Surname	Emp_ID	Hometown	Job Category	FlightNumber	FlightDate
MARSHBURN	1106	STAMFORD	PT	579	05MAR94
DENNIS	1118	NEW YORK	PT	132	01MAR94
DENNIS	1118	NEW YORK	PT	271	04MAR94
KIMANI	1126	NEW YORK	TA	579	05MAR94
TUCKER	1882	NEW YORK	ME	622	03MAR94

Example 11: Retrieving Values with the SOUNDS-LIKE Operator

Procedure features:
ORDER BY clause
SOUNDS-LIKE operator

Table: PROCLIB.STAFF

This example returns rows based on the functionality of the SOUNDS-LIKE operator in a WHERE clause.

Input Table

PROCLIB.STAFF

```
                              PROCLIB.STAFF
                            First 10 Rows Only

 Id
 Num   Lname            Fname            City             State  Hphone
 ------------------------------------------------------------------------------
 1919  ADAMS            GERALD           STAMFORD         CT     203/781-1255
 1653  ALIBRANDI        MARIA            BRIDGEPORT       CT     203/675-7715
 1400  ALHERTANI        ABDULLAH         NEW YORK         NY     212/586-0808
 1350  ALVAREZ          MERCEDES         NEW YORK         NY     718/383-1549
 1401  ALVAREZ          CARLOS           PATERSON         NJ     201/732-8787
 1499  BAREFOOT         JOSEPH           PRINCETON        NJ     201/812-5665
 1101  BAUCOM           WALTER           NEW YORK         NY     212/586-8060
 1333  BANADYGA         JUSTIN           STAMFORD         CT     203/781-1777
 1402  BLALOCK          RALPH            NEW YORK         NY     718/384-2849
 1479  BALLETTI         MARIE            NEW YORK         NY     718/384-8816
```

Program

```
libname proclib 'SAS-data-library';


options nodate pageno=1 linesize=80 pagesize=60;
```

The SELECT clause selects all columns from the table in the FROM clause, PROCLIB.STAFF.

```
proc sql;
   title "Employees Whose Last Name Sounds Like 'Johnson'";
   select *
      from proclib.staff
```

The WHERE clause uses the SOUNDS-LIKE operator to subset the table by those employees whose last name sounds like **Johnson**. The ORDER BY clause orders the output by the second column.

```
      where lname=*"Johnson"
      order by 2;
```

Output

```
              Employees Whose Last Name Sounds Like 'Johnson'                1

 Id
 Num   Lname            Fname            City             State  Hphone
 ------------------------------------------------------------------------------
 1411  JOHNSEN          JACK             PATERSON         NJ     201/732-3678
 1113  JOHNSON          LESLIE           NEW YORK         NY     718/383-3003
 1369  JONSON           ANTHONY          NEW YORK         NY     212/587-5385
```

SOUNDS-LIKE is useful, but there may be instances where it does not return every row that seems to satisfy the condition. PROCLIB.STAFF has an employee with the last name **SANDERS** and an employee with the last name **SANYERS**. The algorithm does not find **SANYERS**, but it does find **SANDERS** and **SANDERSON**.

```
title "Employees Whose Last Name Sounds Like 'Sanders'";
select *
   from proclib.staff
   where lname=*"Sanders"
   order by 2;
```

```
              Employees Whose Last Name Sounds Like 'Sanders'                2

Id
Num   Lname            Fname            City             State  Hphone
-------------------------------------------------------------------------------
1561  SANDERS          RAYMOND          NEW YORK         NY     212/588-6615
1414  SANDERSON        NATHAN           BRIDGEPORT       CT     203/675-1715
1434  SANDERSON        EDITH            STAMFORD         CT     203/781-1333
```

Example 12: Joining Two Tables and Calculating a New Value

Procedure features:

GROUP BY clause
HAVING clause
SELECT clause
- ABS function
- FORMAT= column-modifier
- LABEL= column-modifier
- MIN summary function
- ** operator, exponentiation
- SQRT function

Tables: STORES, HOUSES

This example joins two tables in order to compare and analyze values that are unique to each table yet have a relationship with a column that is common to both tables.

```
options ls=80 ps=60 nodate pageno=1 ;
data stores;
  input Store $ x y;
  datalines;
store1 6 1
store2 5 2
store3 3 5
store4 7 5
```

```
;
data houses;
   input House $ x y;
   datalines;
house1 1 1
house2 3 3
house3 2 3
house4 7 7
;
```

Input Tables

STORES and HOUSES

The tables contain X and Y coordinates that represent the location of the stores and houses.

```
                              STORES Table                                  1
                          Coordinates of Stores

                        Store            x          y
                        ----------------------------
                        store1           6          1
                        store2           5          2
                        store3           3          5
                        store4           7          5
```

```
                              HOUSES Table                                  2
                          Coordinates of Houses

                        House            x          y
                        ----------------------------
                        house1           1          1
                        house2           3          3
                        house3           2          3
                        house4           7          7
```

Program

```
options nodate pageno=1 linesize=80 pagesize=60;
```

The SELECT clause specifies three columns: HOUSE, STORE, and DIST. The arithmetic expression uses the square root function (SQRT) to create the values of DIST, which contain the distance from HOUSE to STORE for each row. The double asterisk (**) represents exponentiation. LABEL= assigns a label to STORE and to DIST.

```
proc sql;
   title 'Each House and the Closest Store';
   select house, store label='Closest Store',
          sqrt((abs(s.x-h.x)**2)+(abs(h.y-s.y)**2)) as dist
```

```
        label='Distance' format=4.2
from stores s, houses h
```

The minimum distance from each house to all the stores is calculated because the data are grouped by house. The HAVING clause specifies that each row be evaluated to determine if its value of DIST is the same as the minimum distance for that house to any store.

```
group by house
having dist=min(dist);
```

Output

```
                    Each House and the Closest Store                    1

                              Closest
                House         Store      Distance
                ---------------------------------
                house1        store2         4.12
                house2        store3         2.00
                house3        store3         2.24
                house4        store4         2.00
```

Example 13: Producing All the Possible Combinations of the Values in a Column

Procedure features:
CASE expression
joined-table component
SELECT clause
DISTINCT keyword
Tables: PROCLIB.MARCH, FLIGHTS

This example joins a table with itself to get all the possible combinations of the values in a column.

Input Table

PROCLIB.MARCH (Partial Listing)

PROCLIB.MARCH
First 10 Rows Only

Flight	Date	Depart	Orig	Dest	Miles	Boarded	Capacity
114	01MAR94	7:10	LGA	LAX	2475	172	210
202	01MAR94	10:43	LGA	ORD	740	151	210
219	01MAR94	9:31	LGA	LON	3442	198	250
622	01MAR94	12:19	LGA	FRA	3857	207	250
132	01MAR94	15:35	LGA	YYZ	366	115	178
271	01MAR94	13:17	LGA	PAR	3635	138	250
302	01MAR94	20:22	LGA	WAS	229	105	180
114	02MAR94	7:10	LGA	LAX	2475	119	210
202	02MAR94	10:43	LGA	ORD	740	120	210
219	02MAR94	9:31	LGA	LON	3442	147	250

Program

```
libname proclib 'SAS-data-library';


options nodate pageno=1 linesize=80 pagesize=60;
```

The CREATE TABLE statement creates the table FLIGHTS from the output of the query. The SELECT clause selects the unique values of Dest. DISTINCT specifies that only one row for each value of city be returned by the query and stored in the table FLIGHTS. The FROM clause specifies PROCLIB.MARCH as the table to select from.

```
proc sql;
   create table flights as
      select distinct dest
         from proclib.march;

   title 'Cities Serviced by the Airline';


select * from flights;
```

Output

FLIGHTS Table

```
                        Cities Serviced by the Airline                          1

                                    Dest
                                    ----
                                    FRA
                                    LAX
                                    LON
                                    ORD
                                    PAR
                                    WAS
                                    YYZ
```

The SELECT clause specifies three columns for the output. The prefixes on DEST are table aliases to specify which table to take the values of Dest from. The CASE expression creates a column that contains the character string **to and from**.

```
title 'All Possible Connections';
select f1.Dest, case
                   when f1.dest ne ' ' then 'to and from'
                end,
       f2.Dest
```

The FROM clause joins FLIGHTS with itself and creates a table that contains every possible combination of rows. The table contains two rows for each possible route, for example, **PAR <-> WAS** and **WAS <-> PAR**.

```
       from flights as f1, flights as f2
```

The WHERE clause subsets the internal table by choosing only those rows where the name in F1.Dest sorts before the name in F2.Dest. Thus, there is only one row for each possible route.

```
       where f1.dest < f2.dest
```

ORDER BY sorts the result by the values of F1.Dest.

```
       order by f1.dest;
```

Output

```
                          All Possible Connections                          2

                          Dest               Dest
                          -----------------------
                          FRA   to and from  WAS
                          FRA   to and from  YYZ
                          FRA   to and from  LAX
                          FRA   to and from  ORD
                          FRA   to and from  LON
                          FRA   to and from  PAR
                          LAX   to and from  PAR
                          LAX   to and from  LON
                          LAX   to and from  WAS
                          LAX   to and from  ORD
                          LAX   to and from  YYZ
                          LON   to and from  WAS
                          LON   to and from  PAR
                          LON   to and from  YYZ
                          LON   to and from  ORD
                          ORD   to and from  WAS
                          ORD   to and from  PAR
                          ORD   to and from  YYZ
                          PAR   to and from  YYZ
                          PAR   to and from  WAS
                          WAS   to and from  YYZ
```

Example 14: Matching Case Rows and Control Rows

Procedure features:
joined-table component

Tables: MATCH_11 on page 1504, MATCH

This example uses a table that contains data for a case-control study. Each row contains information for a case or a control. To perform statistical analysis, you need a table with one row for each case-control pair. PROC SQL joins the table with itself in order to match the cases with their appropriate controls. After the rows are matched, differencing can be performed on the appropriate columns.

The input table MATCH_11 contains one row for each case and one row for each control. Pair contains a number that associates the case with its control. Low is 0 for the controls and 1 for the cases. The remaining columns contain information about the cases and controls.

Input Table

MATCH_11 Table
First 10 Rows Only

Pair	Low	Age	Lwt	Race	Smoke	Ptd	Ht	UI	race1	race2
1	0	14	135	1	0	0	0	0	0	0
1	1	14	101	3	1	1	0	0	0	1
2	0	15	98	2	0	0	0	0	1	0
2	1	15	115	3	0	0	0	1	0	1
3	0	16	95	3	0	0	0	0	0	1
3	1	16	130	3	0	0	0	0	0	1
4	0	17	103	3	0	0	0	0	0	1
4	1	17	130	3	1	1	0	1	0	1
5	0	17	122	1	1	0	0	0	0	0
5	1	17	110	1	1	0	0	0	0	0

Program

```
options nodate pageno=1 linesize=80 pagesize=60;
```

The SELECT clause specifies the columns for the table MATCH. SQL expressions in the SELECT clause calculate the differences for the appropriate columns and create new columns.

```
proc sql;
   create table match as
      select
         one.Low,
         one.Pair,
         (one.lwt - two.lwt) as Lwt_d,
         (one.smoke - two.smoke) as Smoke_d,
         (one.ptd - two.ptd) as Ptd_d,
         (one.ht - two.ht) as Ht_d,
         (one.ui - two.ui) as UI_d
```

The FROM clause lists the table MATCH_11 twice. Thus, the table is joined with itself. The WHERE clause returns only the rows for each pair that show the difference when the values for control are subtracted from the values for case.

```
      from match_11 one, match_11 two
      where (one.pair=two.pair and one.low>two.low);
```

The SELECT clause selects all the columns from MATCH. The OBS= data set option limits the printing of the output to five rows.

```
   title 'Differences for Cases and Controls';
   select *
      from match(obs=5);
```

Output

MATCH Table

Differences for Cases and Controls 1

Low	Pair	Lwt_d	Smoke_d	Ptd_d	Ht_d	UI_d
1	1	-34	1	1	0	0
1	2	17	0	0	0	1
1	3	35	0	0	0	0
1	4	27	1	1	0	1
1	5	-12	0	0	0	0

Example 15: Counting Missing Values with a SAS Macro

Procedure feature:
COUNT function
Table: SURVEY

This example uses a SAS macro to create columns. The SAS macro is not explained here. See the SAS Guide to Macro Processing for complete documentation on the SAS COUNTM macro.

Input Table

SURVEY contains data from a questionnaire about diet and exercise habits. SAS enables you to use a special notation for missing values. In the EDUC column, the **.x** notation indicates that the respondent gave an answer that is not valid, and **.n** indicates that the respondent did not answer the question. A period as a missing value indicates a data entry error.

```
data survey;
  input id $ diet $ exer $ hours xwk educ;
  datalines;
1001 yes yes 1 3 1
1002 no  yes 1 4 2
1003 no  no  . . .n
1004 yes yes 2 3 .x
1005 no  yes 2 3 .x
1006 yes yes 2 4 .x
1007 no  yes .5 3 .
1008 no  no  . .  .
;
```

Program

```
options nodate pageno=1 linesize=80 pagesize=60;
```

The COUNTM macro uses the COUNT function to perform various counts for a column. Each COUNT function uses a CASE expression to select the rows to be counted. The first COUNT function uses only the column as an argument to return the number of nonmissing rows.

```
%macro countm(col);
   count(&col) "Valid Responses for &col",
```

The IS MISSING keywords return the rows that have any type of missing value: **.n**, **.x**, or a period. The PUT function returns a character string to be counted.

```
   count(case
            when &col is missing  then put(&col, 2.)
            end) "Missing or NOT VALID Responses for &col",
```

The last three COUNT functions use CASE expressions to count the occurrences of the three notations for missing values.

```
   count(case
            when &col=.n  then put(&col, 2.)
            end) "Coded as NO ANSWER for &col",
   count(case
            when &col=.x  then put(&col, 2.)
            end) "Coded as NOT VALID answers for &col",
   count(case
            when &col=.  then put(&col, 1.)
            end) "Data Entry Errors for &col"
%mend;
```

The SELECT clause specifies the columns that are in the output. COUNT(*) returns the total number of rows in the table. The COUNTM macro uses the values of the EDUC column to create the columns defined in the macro.

```
proc sql;
   title 'Counts for Each Type of Missing Response';
   select count(*)  "Total No. of Rows",
          %countm(educ)
      from survey;
```

Output

Counts for Each Type of Missing Response 1

Total No. of Rows	Valid Responses for educ	Missing or NOT VALID Responses for educ	Coded as NO ANSWER for educ	Coded as NOT VALID answers for educ	Data Entry Errors for educ
8	2	6	1	3	2

The correct bibliographic citation for this manual is as follows: SAS Institute Inc., *SAS® Procedures Guide, Version 8,* Cary, NC: SAS Institute Inc., 1999. 1729 pp.

SAS® Procedures Guide, Version 8

ISBN 1–58025–482–9

SAS Institute Inc., SAS Campus Drive, Cary, North Carolina 27513.

1st printing, October 1999