PROC SQL: Basic and Advance Using SAS

Outline

- Introduction
- Basic Queries
- Combining Tables
- Creating and Modifying Tables and Views
- Additional SQL Features

Introduction

Structured Query Language (SQL)

is a standardized language that is widely used to retrieve and update data in tables and in views based on those tables

The SQL procedure uses SQL to

- query SAS data sets
- generate reports from SAS data sets
- combine SAS data sets in many ways
- create and delete SAS data files, views, and indexes
- update existing SAS data sets.

Introduction (Cont'd)

Features of SQL

- The **PROC SQL** statement does not need to be repeated with each query.
- Each statement is processed individually.
- No **PROC PRINT** step is needed to view query results.
- No **PROC SORT** step is needed to order query results.
- No **RUN** statement is needed.
- Use a **QUIT** statement to terminate **PROC SQL**.

Basic Queries

- Overview of the SQL Procedure
- Specifying Columns
- Specifying Rows
- Presenting Data
- Summarizing Data
- Performing Subqueries

Overview of the SQL Procedure

```
PROC SQL  <option <option>…>;
      ALTER expression;
      CREATE expression;
      DELETE expression;
      DESCRIBE expression;
      DROP expression;
      INSERT expression;
      RESET expression;
      SELECT expression;
      UPDATE expression;
```

ALTER	adds, drops, and modifies columns in a table.
CREATE	builds new tables.
DELETE	eliminates unwanted rows from a table or view.
DESCRIBE	displays table attributes.
DROP	eliminates entire tables, views, or indexes.
INSERT	adds rows of data to tables.
RESET	adds to or changes PROC SQL options without re-invoking the procedure.
SELECT	specifies columns to be printed.
UPDATE	modifies data values in existing rows of a table or view.

Overview of the SQL Procedure (Cont'd)

- ✓ **The SELECT statement**

A **SELECT** statement is used to query one or more SAS data sets

```
proc sql;
    select EmpID, JobCode, Salary
        from airline.payrollmaster
        where JobCode contains 'NA'
        order by Salary desc;
```

Note: Use a comma to separate items in a list, such as column or table names.
Place a single semicolon at the end of the last clause

The SELECT Statement (Cont'd)

SELECT Statement Syntax

```
SELECT column <,column>…
    FROM table\view <, table\view>…
    <WHERE expression>
    <GROUP BY column <,column>…>
     <HAVING expression>
     <ORDER BY column <,column>…>;
```

SELECT	specifies the columns to be selected.
FROM	specifies the table to be queried.
WHERE	subsets the data based on a condition.
GROUP BY	classifies the data into groups.
HAVING	subsets groups of data based on a group condition.
ORDER BY	sorts rows by the values of specific columns.

✓ The order of the above clauses within the SQL SELECT statement **does** matter.

The SELECT Statement (Cont'd)

The **SELECT** statement

- selects data that meets certain conditions
- groups data
- specifies an order for the data
- formats the data
- queries 1 to 32 tables.

Note: **Tables names** can be 1 to 32 characters in length and are not case-sensitive.
Variable names can be 1 to 32 characters in length and are stored in mixed case but are normalized for lookups and comparisons. However, the first usage of the variable determines the capitalization pattern.
Librefs, filerefs, formats, and informats are limited to 8 characters.

The SELECT Statement (Cont'd)

The **VALIDATE** keyword

- is used only on a **SELECT** statement
- tests the syntax of a query without executing the query
- checks column name validity
- prints error messages for invalid queries.

```
1   proc sql;
2         validate
3         select EmployeeNumber, JobCode, Salary
4               from airline.payrollmaster
5               where JobCode contains 'NA'
6               order by Salary desc;
Note: PROC SQL statement has valid syntax.
```

The SELECT Statement (Cont'd)

The **NOEXEC** option can also be used for syntax checking.

```
1   proc sql noexec;
2         select EmployeeNumber, JobCode, Salary
3               from airline.payrollmaster
4               where JobCode contains 'NA'
5               order by Salary desc;
```

Note: Statement not executed due to NOEXEC option.

- ✓ The **NOEXEC** option checks for invalid syntax in all the statements previously mentioned, but the **VALIDATE** option applies only to the SELECT statement.

Specifying Columns

- ✓ **Retrieve Data from a Table**

Specify column names to be printed in the **SELECT** statement.

```
proc sql;
    select EmpID, JobCode, Salary
        from airline.payrollmaster;
```

An asterisk in the **SELECT** statement prints all columns in their originally stored order.

```
proc sql;
    select *
        from airline.payrollmaster;
```

Specify Columns (Cont'd)

✓ The FEEDBACK Option

Use the **FEEDBACK** option to write the expanded **SELECT** statement to the SAS log.

```
1   proc sql feedback;
2         select *
3               from airline.payrollmaster;
Note: Statement transforms to:
          select PAYROLLMASTER.Gender, PAYROLLMASTER.JobCode,
                 PAYROLLMASTER.Salary,
                 PAYROLLMASTER.DateOfBirth,
                 PAYROLLMASTER.DateOfHire
               from AIRLINE.PAYROLLMASTER;
```

- ✓ This option expands any use of an asterisk into the list of qualified columns it represents. NOFEEDBACK is the default.

Specify Columns (Cont'd)

✓ Expressions

- Calculate new columns from existing columns and name the new columns using the **AS** keyword.

```
proc sql;
     select EmpID, JobCode, Salary,
          Salary*.10 as Bonus
          from airline.payrollmaster;
```

Note: The new column is called an *alias*. The **AS** keyword is required. Omission causes the column heading to be blank.

Specify Columns (Cont'd)

Expressions (Cont'd)

- Use SAS DATA step functions for calculating columns.

```
proc sql;
     select EmpID, JobCode,
               int((today()-DateOfBirth) / 365.25) as Age
          from airline.payrollmaster ;
```

✓ All SAS DATA step functions are supported except **LAG**, **DIF**.

Specifying Rows

- By default, all rows in a table are returned in a query.
- Use the **DISTINCT** keyword to eliminate duplicate rows in query results.

```
proc sql;
     select distinct FlightNumber, Destination
          from airline.payrollmaster ;
```

✓ The **DISTINCT** keyword applies to all columns in the **SELECT** list. One row is displayed for each existing combination of values.

Specifying Rows (Cont'd)

✓ Subsetting with the WHERE clause

- Use a **WHERE** clause to specify a condition that the data must satisfy before being selected.

```
proc sql;
     select EmpID, JobCode, Salary
          from airline.payrollmaster
          where Salary > 112000 ;
```

- You can use all common comparison operators in a **WHERE** clause.

Mnemonic	Symbol	Definition
LT	<	Less than
GT	>	Greater than
EQ	=	Equal to
LE	<=	Less than or equal to
GE	>=	Greater than or equal to
NE	^=	Not equal to (ASCII)

Specifying Rows (Cont'd)

Subsetting with the WHERE clause (Cont'd)

- Use the **IN** operator to compare a value to a list of values. If the value matches at least one in the list, the expression is true; otherwise, the expression is false.

```
where JobCategory in ('PT', 'NA', 'FA')
where DayOfWeek in (2, 4, 6)
```

- Specify multiple expressions in a WHERE clause by using logical operators.

Mnemonic	Symbol	Definition
OR	|	or, either
AND	&	and, both
NOT	¬	not, negation ANSII

Specifying Rows (Cont'd)

Subsetting with the WHERE clause (Cont'd)

- Use either **CONTAINS** or **?** to select rows that include the substring specified.

```
where word ? 'LAM'
(BLAME, LAMENT, and BEDLAM are selected.)
```

- Use either **IS NULL** or **IS MISSING** to select rows with missing values.

```
where FlightNumber is missing
```

  - Alternative statetments are

```
where FlightNumber = ' '
where FlightNumber = .
```

  - With the = operator, you must know if FlightNumber is character or numeric. On the other hand, if you use MISSING=, you do not need advance knowledge of column type.

Specifying Rows (Cont'd)

Subsetting with the WHERE clause (Cont'd)

- Use **BETWEEN-AND** to select rows containing ranges of values, *inclusively*.

```
where Date between '01mar2000'd and '07mar2000'd
where Salary between 70000 and 80000;
```

- Use **LIKE** to select rows by comparing *character* values to specified patterns.
  - A % sign replaces any number of characters.

```
where LastName like ' H% '
```

  - A single underscore ('_') replaces individual characters.

```
where JobCode like '__1'
```

2 underscores followed by a 1

captures any two characters and 1, e.g., 'FA1', 'TA1', 'NA1'.

Specifying Rows (Cont'd)

Subsetting with the WHERE clause (Cont'd)

- Use **sounds-like (=*)** to select rows containing a spelling variation of the specified word(s).

```
where LastName =* 'SMITH'
```

selects values SMITT, SMYTHE, and SMOTHE, in addition to SMITH.

Specifying Rows (Cont'd)

✓ Subsetting with Calculated Values

- Because a **WHERE** clause is evaluated first, columns used in the **WHERE** clause must exist in the table or be derived from existing columns.

```
proc sql;
     select FlightNumber, Date, Destination,
               Boarded + Transferred + Nonrevenue as Total
          from airline.marchflights
          where Total < 100 ;
```

ERROR: The following columns were not found in the contributing tables: Total.

Specifying Rows (Cont'd)

Subsetting with Calculated Values (Cont'd)

- One solution is to repeat the calculation in the **WHERE** clause.

```
proc sql;
    select FlightNumber, Date, Destination,
           Boarded + Transferred + Nonrevenue as Total
        from airline.marchflights
        where Boarded + Transferred + Nonrevenue < 100 ;
```

- A more efficient method is to use the **CALCULATED** keyword to refer to already calculated columns in the **SELECT** clause.

```
proc sql;
    select FlightNumber, Date, Destination,
           Boarded + Transferred + Nonrevenue as Total
        from airline.marchflights
        where calculated Total < 100 ;
```

Specifying Rows (Cont'd)

Subsetting with Calculated Values (Cont'd)

- Use the **CALCULATED** keyword in other parts of a query, e.g., in a **SELECT** clause.

```
proc sql;
     select FlightNumber, Date, Destination,
               Boarded + Transferred + Nonrevenue as Total,
          calculated Total/2 as half
          from airline.marchflights ;
```

Presenting Data

✓ Ordering Data

- Use the **ORDER BY** clause to sort query results in
  - ascending order (the default)
  - descending order by following the column name with the **DESC** keyword

```
proc sql;
    select EmpID, JobCode, Salary
        from airline.payrollmaster
        where JobCode contains 'NA'
        order by Salary desc;
```

- ✓ **PROC SQL** uses information provided by a table's internal sort indicator (if applicable) to avoid performing unnecessary sorts.
- ✓ You can specify the collating sequence by using the **SORTSEQ**=option in the **PROC SQL** statement. Use this option if you want a collating sequence other than your system's or installation's default

Presenting Data (Cont'd)

Ordering Data (Cont'd)

- In an **ORDER BY** clause, you order query results by specifying
  - any column or expression (display or nondisplay)
  - a column name or a number that represents the position of an item in the SELECT list
  - multiple columns.

```
proc sql;
    select FlightNumber, Date, Origin, Destination,
           Boarded + Transferred + Nonrevenue
        from airline.marchflights
        where Destination = 'LHR'
        order by Date, 5 desc ;
```

Presenting Data (Cont'd)

✓ Enhancing Query Output

- You can use SAS formats and labels to customize **PROC SQL** output. After the column name in the **SELECT** list, you specify the
  - LABEL= option to alter the column heading
  - FORMAT= option to alter the appearance of the values in that column.

```
proc sql;
    select EmpID label= 'Employee Identifier',
           JobCode label= 'Job Code',
           Salary label= 'Annual Salary' format= dollar12.2
      from airline.payrollmaster
      where JobCode contains 'NA'
      order by Salary desc ;
```

Presenting Data (Cont'd)

Enhancing Query Output (Cont'd)

- You can
  - define a column containing a character constant by placing a text string in the SELECT list
  - use SAS titles and footnotes to enhance the query's appearance.

```
proc sql;
title 'Current Bonus Information' ;
title2 'Navigators – All Levels' ;
    select EmpID label='Employee Identifier',
           'bonus is: ',
            Salary * .05 format=dollar12.2
        from airline.payrollmaster
        where JobCode contains 'NA'
        order by Salary desc ;
```

Note: **TITLE** and **FOOTNOTE** statements must precede the **SELECT** statement

Summarizing Data

✓ Summary Functions

```
proc sql;
   select Date, FlightNumber, Boarded,
          Transferred, Nonrevenue,
     sum(Boarded,Transferred,Nonrevenue)
     as Total
     from airline.marchflights;
```

If you specify more than one column name in a summary function, the function acts like a DATA step function. The calculation is performed for each row.

```
proc sql;
   select avg(Salary) as MeanSalary
     from airline.payrollmaster;
```

If you specify only one column name in a summary function, the statistic is calculated down the column.

Summarizing Data (Cont'd)

Summary Functions (Cont'd)

AVG, MEAN	mean or average value
COUNT, FREQ, N	number of nonmissing values
MAX	largest value
MIN	smallest value
NMISS	number of missing values
STD	standard deviation
SUM	sum of values
VAR	variance.

Summarizing Data (Cont'd)

✓ Grouping Data

- You can use the **GROUP BY** clause to
  - classify the data into groups based on the values of one or more columns
  - calculate statistics for each unique value of the grouping columns.

```
proc sql;
     select JobCode, avg(Salary) as
     average format=dollar11.2
  from airline.payrollmaster
  group by JobCode;
```

Summarizing Data (Cont'd)

✓ Analyzing Groups of Data

- The COUNT(*) summary function counts the number of rows.

```
proc sql;
     select count(*) as count
        from airline.payrollmaster;
```

```
proc sql;
     select substr(JobCode,1,2) label = 'Job Category',
     count(*) as count
  from airline.payrollmaster
  group by 1;
```

Summarizing Data (Cont'd)

Analyzing Groups of Data (Cont'd)

```
proc sql;
    select EmpID, Salary, (Salary/sum(Salary)) as percent
    format = percent8.2
  from airline.payrollmaster
  where JobCode contains 'NA';
```

Note: PROC SQL automatically re-merges the summary statistic with the table to calculate the percentage. This requires two passes through the data: one to compute the column sum and another to compute each row's percentage of the total. A note appears in the SAS log when re-merging occurs.

Summarizing Data (Cont'd)

✓ Selecting Groups of Data with the HAVING Clause

The **WHERE** clause selects data based on values for individual rows. To select entire groups of data, use the **HAVING** clause.

```
proc sql;
    select JobCode, avg(Salary) as average format=dollar11.2
    from airline.payrollmaster
    group by JobCode
    having avg(Salary) > 56000;
```

Alternatively, you can code the **HAVING** clause as follows:

- ✓ having average > 56000;
- ✓ having calculated average > 56000;

Subqueries

- are inner queries that return values to be used by an outer query to complete a subsetting expression in a **WHERE** or **HAVING** clause
- return single or multiple values to be used by the outer query
- can return only a single column.

Subqueries (Cont'd)

Noncorrelated subqueries

- A noncorrelated subquery is a subquery that is independent of the outer query and it can executed on its own without relying on main outer query.

Correlated subqueries

- cannot be evaluated independently, but depend on the values returned by the outer query for their results.
- are evaluated for each row in the outer query.

Subqueries (Cont'd)

✓ **Noncorrelated Subqueries**

- Example: Display job codes where the group's average salary exceeds the company's average salary.

```
proc sql;
     select JobCode, avg(Salary) as MeanSalary
        from airline.payrollmaster
        group by JobCode
        having avg(Salary) >
            (select avg(Salary) from airline.payrollmaster) ;
```

Evaluated first, then pass result to outer query

Alternatively, you can code the **HAVING** clause as follows:

✓ having average > 56000;

✓ having calculated average > 56000;

Subqueries (Cont'd)

Noncorrelated Subqueries (Cont'd)

- Example: Send birthday cards to employees with February birthdays. Names and addresses are in airline.staffmaster, and birth dates in airline.payrollmaster.

```
proc sql;
     select LastName, FirstName, City, State
          from airline.staffmaster
          where EmpID in
               (select EmpID
                from airline.payrollmaster
                where month(DateOfBirth)=2) ;
```

Subqueries (Cont'd)

Noncorrelated Subqueries: How do they work?

✓ Step 1: Evaluate the inner query and build a virtual table that satisfies the **WHERE** criteria.

```
proc sql;
     select LastName, FirstName,
               City,   State
          from airline.staffmaster
          where EmpID in
              (select EmpID
               from airline.payrollmaster
               where month(DateOfBirth)=2) ;
```

Virtual table contains
'1420', ' 1390', ' 1403', ' 1404', ' 1834', ' 1103'.

Airline.payrollmaster
Partial Listing

EmpID	DateOfBirth
. . .	. . .
1038	11/13/1967
1420	02/23/1963
1561	12/03/1961
1434	07/14/1960
1414	03/28/1970
1112	12/03/1962
1390	02/23/1963
1332	09/20/1968
. . .	. . .

Subqueries (Cont'd)

Noncorrelated Subqueries: How do they work?

✓ Step 2: Pass values in the virtual table to the outer query.

```
proc sql;
     select LastName, FirstName, City, State
          from airline.staffmaster
          where EmpID in
           ('1420', '1390', '1403',
            '1404', '1834', '1103') ;
```

Pass '1420', ' 1390', '1403', '1404', '1834', '1103' to the outer query.

Airline.payrollmaster
Partial Listing

EmpID	DateOfBirth
. . .	. . .
1038	11/13/1967
1420	02/23/1963
1561	12/03/1961
1434	07/14/1960
1414	03/28/1970
1112	12/03/1962
1390	02/23/1963
1332	09/20/1968
. . .	. . .

Subqueries (Cont'd)

✓ **Correlated Subqueries**

Example: Display the names and states of all navigator managers.

```
proc sql;
     select LastName, FirstName, State
          from airline.staffmaster
          where 'NA' =
               (select JobCategory
                    from airline.supervisors
                    where staffmaster.EmpID=supervisors.EmpID) ;
```

You must qualify each column with a table name.

Subqueries (Cont'd)

Correlated Subqueries (Cont'd)

- Step 1: The outer query takes the first row in AIRLINE.STAFFMASTER and finds the EmpID, LastName, FirstName, and State.

```
proc sql;
    select LastName, FirstName, State
        from airline.staffmaster
        where 'NA' =
            (select JobCategory
                from airline.supervisors
                where staffmaster.EmpID
                    =supervisors.EmpID) ;
```

Airline.staffmaster - Partial Listing

EmpID	LastName	FirstName
1919	ADAMS	GERALD
1401	AVERY	JERRY
1269	CASTON	FRANKLIN
1935	FERNANDEZ	KATRINA
1124	FIELDS	DIANA
1677	KRAMER	JACKSON
1442	NEWKIRK	SANDRA
1417	NEWKIRK	WILLIAM
1352	RIVERS	SIMON

Airline.supervisors - Partial Listing

EmpID	State	JobCategory
1677	CT	BC
1834	NY	BC
1431	CT	FA
1433	NJ	FA
1385	CT	ME
1420	NJ	ME
1882	NY	ME
1935	CT	NA
1417	NJ	NA
1352	NY	NA
1106	CT	PT
1442	NJ	PT
1405	NJ	SC
1564	NY	SC
1401	NJ	TA
1126	NY	TA

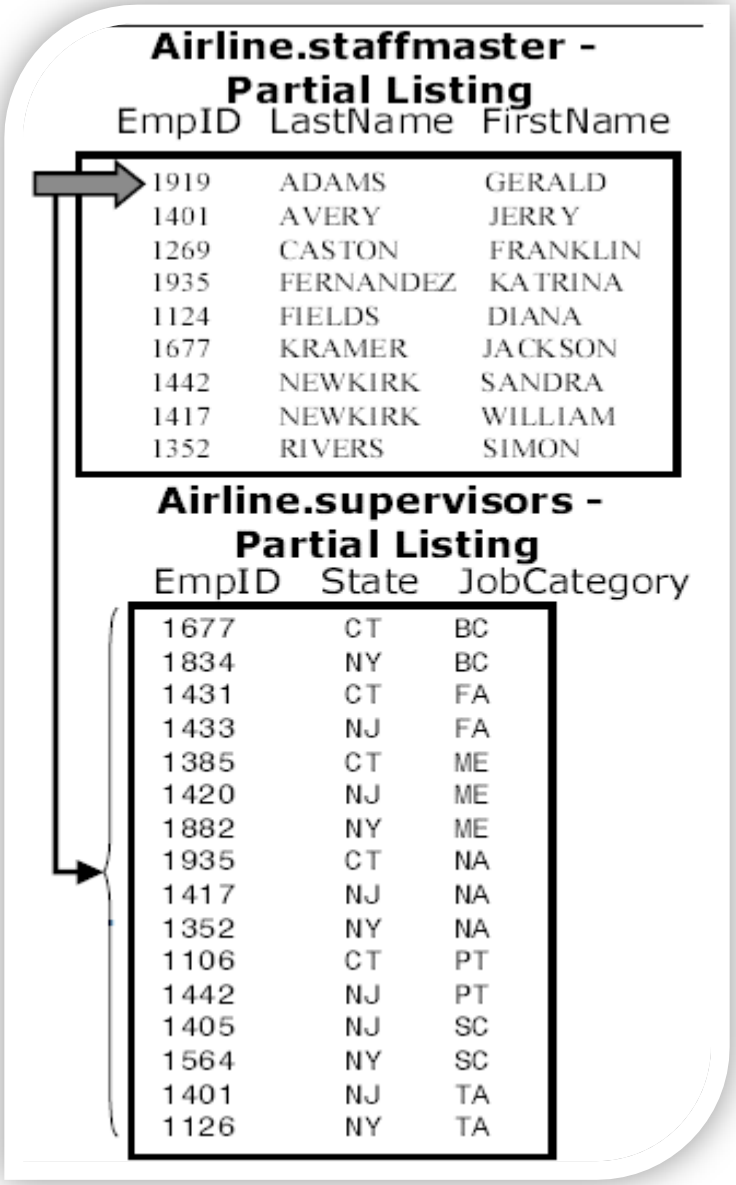
Subqueries (Cont'd)

Correlated Subqueries (Cont'd)

- Step 2: Match STAFFMASTER.EMPID with SUPERVISORS.EMPID to find the qualifying row in AIRLINE.SUPERVISORS.

```
proc sql;
    select LastName, FirstName, State
        from airline.staffmaster
        where 'NA' =
            (select JobCategory
                from airline.supervisors
                where staffmaster.EmpID
                    =supervisors.EmpID);
```

NO MATCH



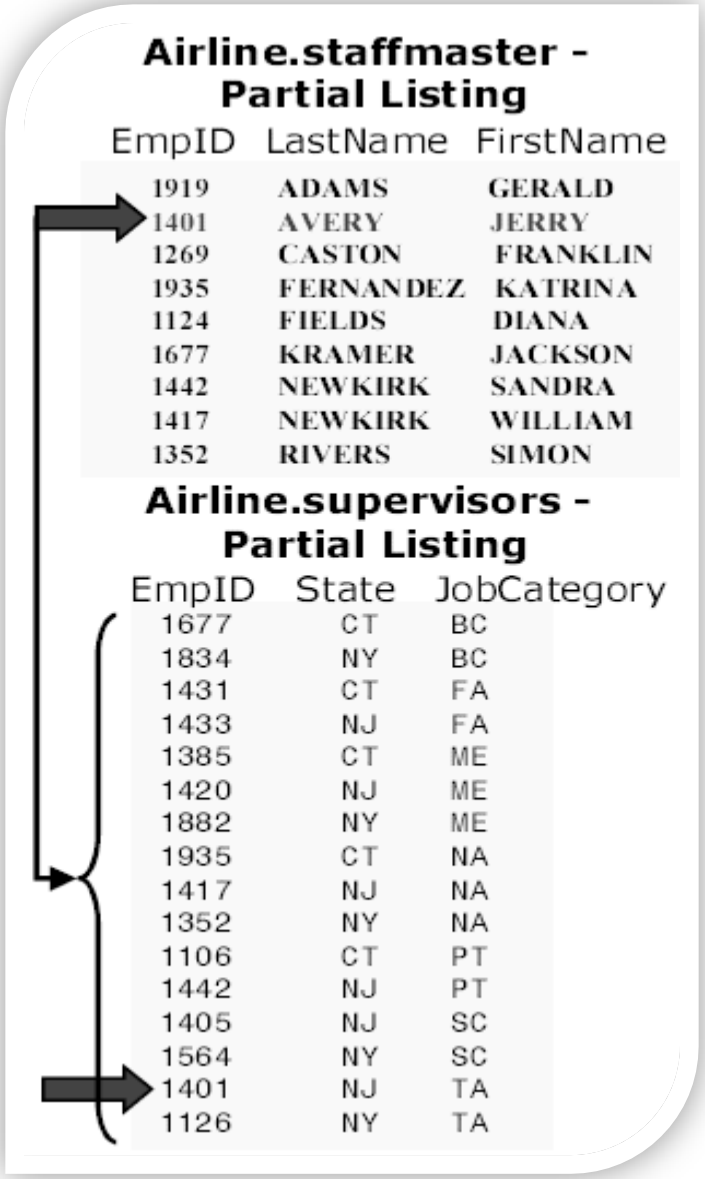
Subqueries (Cont'd)

Correlated Subqueries (Cont'd)

- Step 1 and 2 (repeated): Read the next row from AIRLINE.STAFFMASTER, and identify the qualifying row in AIRLINE.SUPERVISORS.

```
proc sql;
    select LastName, FirstName, State
        from airline.staffmaster
        where 'NA' =
            (select JobCategory
                from airline.supervisors
                where staffmaster.EmpID
                    =supervisors.EmpID) ;
```

MATCH



Subqueries (Cont'd)

Correlated Subqueries (Cont'd)

- Step 3: The inner query now passes the JobCategory of the selected row in AIRLINE.SUPERVISORS back to the outer query via the = operator, where the JobCategory is matched for selection in the outer query.

```
proc sql;
    select LastName, FirstName, State
        from airline.staffmaster
        where 'NA' = 'TA'
            (select JobCategory
                from airline.supervisors
                where staffmaster.EmpID
                    =supervisors.EmpID) ;
```

Resolves to FALSE

Airline.staffmaster - Partial Listing

EmpID	LastName	FirstName
1919	ADAMS	GERALD
1401	AVERY	JERRY
1269	CASTON	FRANKLIN
1935	FERNANDEZ	KATRINA
1124	FIELDS	DIANA
1677	KRAMER	JACKSON
1442	NEWKIRK	SANDRA
1417	NEWKIRK	WILLIAM
1352	RIVERS	SIMON

Airline.supervisors - Partial Listing

EmpID	State	JobCategory
1677	CT	BC
1834	NY	BC
1431	CT	FA
1433	NJ	FA
1385	CT	ME
1420	NJ	ME
1882	NY	ME
1935	CT	NA
1417	NJ	NA
1352	NY	NA
1106	CT	PT
1442	NJ	PT
1405	NJ	SC
1564	NY	SC
1401	NJ	TA
1126	NY	TA

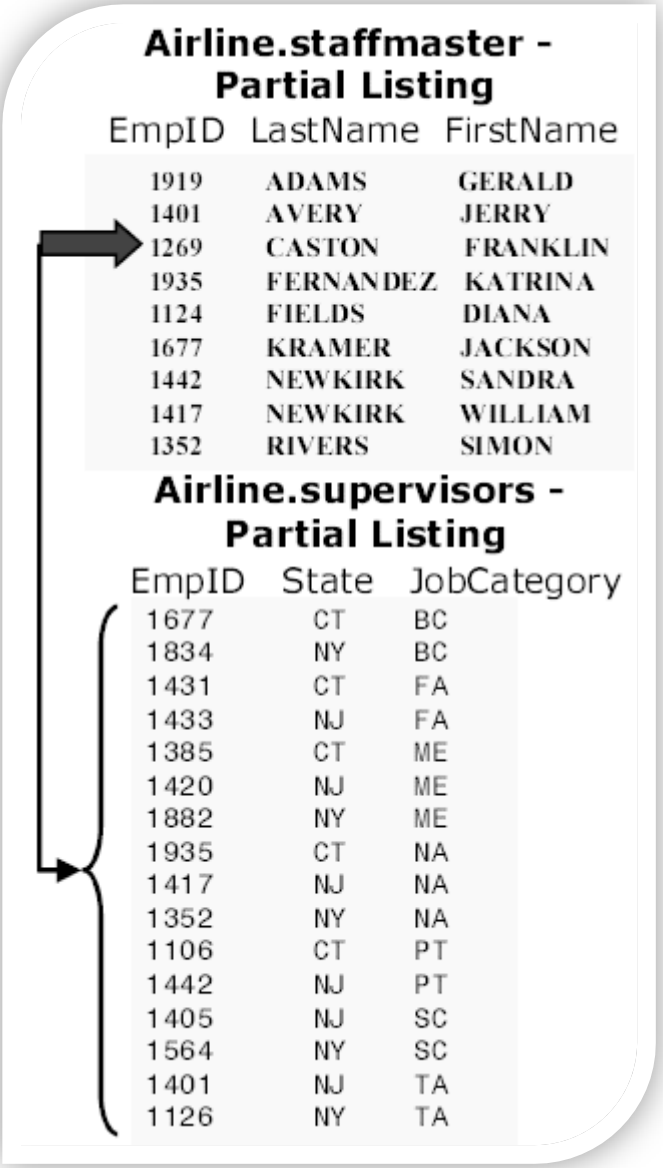
Subqueries (Cont'd)

Correlated Subqueries (Cont'd)

- Continue repeating steps 2 and 3 until all rows are read from AIRLINE.STAFFMASTER.

```
proc sql;
    select LastName, FirstName, State
        from airline.staffmaster
        where 'NA' =
            (select JobCategory
                from airline.supervisors
                where staffmaster.EmpID
                    =supervisors.EmpID) ;
```

NO MATCH

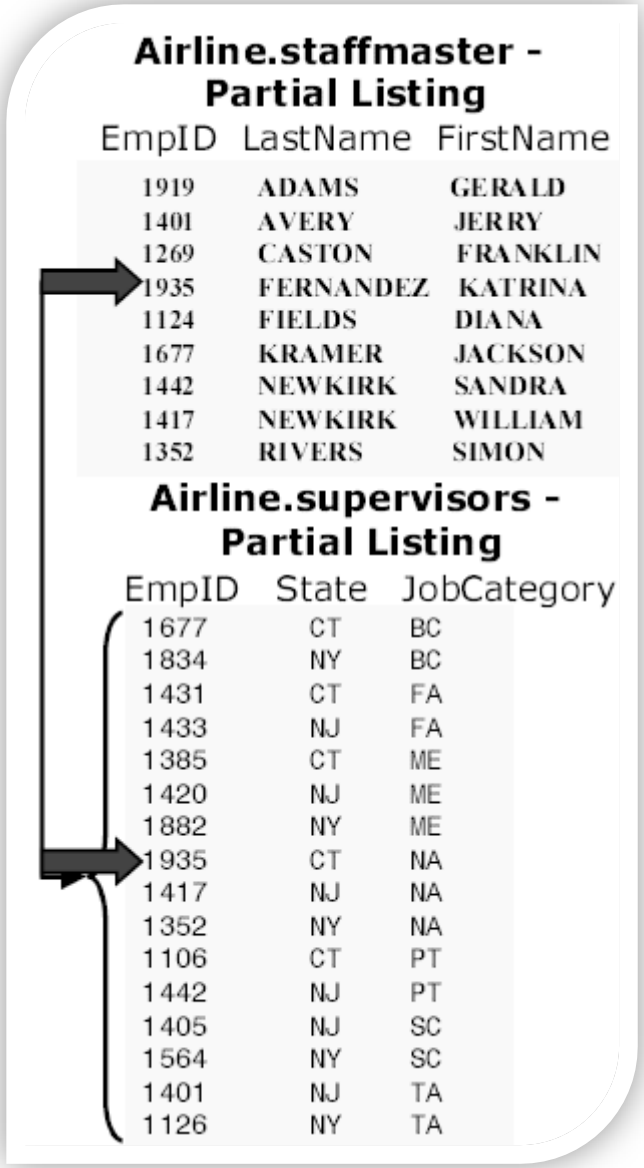


Subqueries (Cont'd)

Correlated Subqueries (Cont'd)

```
proc sql;
    select LastName, FirstName, State
        from airline.staffmaster
        where 'NA' =
            (select JobCategory
                from airline.supervisors
                where staffmaster.EmpID
                    =supervisors.EmpID) ;
```

MATCH



Subqueries (Cont'd)

Correlated Subqueries (Cont'd)

```
proc sql;
    select LastName, FirstName, State
        from airline.staffmaster
        where 'NA' = 'NA'
            (select JobCategory
                from airline.supervisors
                where staffmaster.EmpID
                    =supervisors.EmpID) ;
```

Resolves to TRUE

Pass JobCategory from AIRLINE.SUPERVISORS to outer query for comparison.

Airline.staffmaster - Partial Listing

EmpID	LastName	FirstName
1919	ADAMS	GERALD
1401	AVERY	JERRY
1269	CASTON	FRANKLIN
1935	FERNANDEZ	KATRINA
1124	FIELDS	DIANA
1677	KRAMER	JACKSON
1442	NEWKIRK	SANDRA
1417	NEWKIRK	WILLIAM
1352	RIVERS	SIMON

Airline.supervisors - Partial Listing

EmpID	State	JobCategory
1677	CT	BC
1834	NY	BC
1431	CT	FA
1433	NJ	FA
1385	CT	ME
1420	NJ	ME
1882	NY	ME
1935	CT	NA
1417	NJ	NA
1352	NY	NA
1106	CT	PT
1442	NJ	PT
1405	NJ	SC
1564	NY	SC
1401	NJ	TA
1126	NY	TA

Subqueries (Cont'd)

Correlated Subqueries (Cont'd)

```
proc sql;
    select LastName, FirstName, State
        from airline.staffmaster
        where 'NA' = 'NA'
            (select JobCategory
                from airline.supervisors
                where staffmaster.EmpID
                    =supervisors.EmpID) ;
```

Resolves to TRUE

Write LastName, FirstName, and State from AIRLINE.STAFFMASTER as the first row in a newly-created report.

Airline.staffmaster - Partial Listing

EmpID	LastName	FirstName
1919	ADAMS	GERALD
1401	AVERY	JERRY
1269	CASTON	FRANKLIN
1935	FERNANDEZ	KATRINA
1124	FIELDS	DIANA
1677	KRAMER	JACKSON
1442	NEWKIRK	SANDRA
1417	NEWKIRK	WILLIAM
1352	RIVERS	SIMON

Airline.supervisors - Partial Listing

EmpID	State	JobCategory
1677	CT	BC
1834	NY	BC
1431	CT	FA
1433	NJ	FA
1385	CT	ME
1420	NJ	ME
1882	NY	ME
1935	CT	NA
1417	NJ	NA
1352	NY	NA
1106	CT	PT
1442	NJ	PT
1405	NJ	SC
1564	NY	SC
1401	NJ	TA
1126	NY	TA

Subqueries (Cont'd)

Correlated Subqueries (Cont'd)

```
proc sql;
    select LastName, FirstName, State
        from airline.staffmaster
        where 'NA' = 'NA'
            (select JobCategory
                from airline.supervisors
                where staffmaster.EmpID
                    =supervisors.EmpID);
```

Resolves to TRUE

Write LastName, FirstName, and State from AIRLINE.STAFFMASTER as the first row in a newly-created report.

Airline.staffmaster - Partial Listing

EmpID	LastName	FirstName
1919	ADAMS	GERALD
1401	AVERY	JERRY
1269	CASTON	FRANKLIN
1935	FERNANDEZ	KATRINA
1124	FIELDS	DIANA
1677	KRAMER	JACKSON
1442	NEWKIRK	SANDRA
1417	NEWKIRK	WILLIAM
1352	RIVERS	SIMON

Airline.supervisors - Partial Listing

EmpID	State	JobCategory
1677	CT	BC
1834	NY	BC
1431	CT	FA
1433	NJ	FA
1385	CT	ME
1420	NJ	ME
1882	NY	ME
1935	CT	NA
1417	NJ	NA
1352	NY	NA
1106	CT	PT
1442	NJ	PT
1405	NJ	SC
1564	NY	SC
1401	NJ	TA
1126	NY	TA

Subqueries (Cont'd)

Correlated Subqueries (Cont'd)

Build first row of report:

LastName	FirstName	State
FERNANDEZ	KATRINA	CT

Build third (and final) row of report:

LastName	FirstName	State
FERNANDEZ	KATRINA	CT
NEWKIRK	WILLIAM	NJ
RIVERS	SIMON	NY

Subqueries (Cont'd)

Correlated Subqueries (Cont'd)

The EXISTS condition tests for the existence of a set of values returned by the subquery.

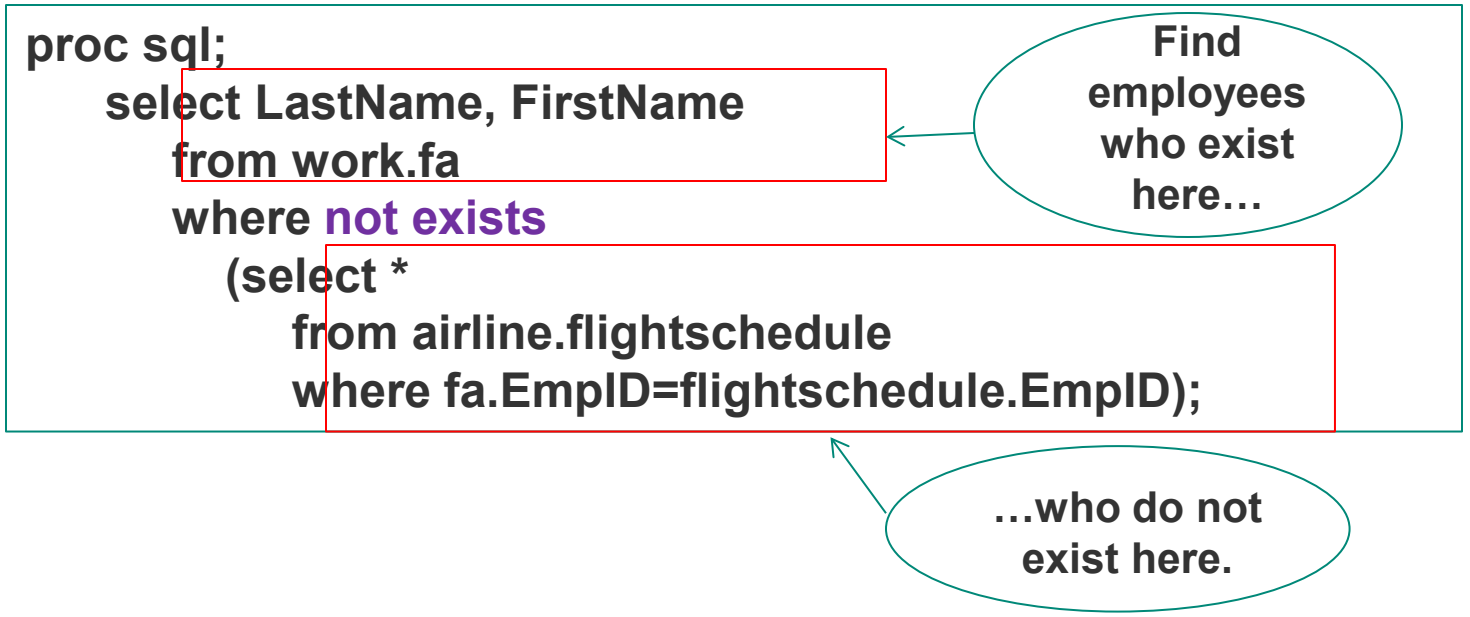
- The EXISTS condition is true if the subquery returns at least one row.
- The NOT EXISTS condition is true if the subquery returns no data.

Subqueries (Cont'd)

Correlated Subqueries (Cont'd)

- Example: The temporary table WORK.FA is a subset of AIRLINE.STAFFMASTER containing the names and IDs of all flight attendants. The AIRLINE.FLIGHTSCHEDULE table contains a row for each crew member assigned to a flight for each date.

Determine which flight attendants have not been scheduled.



Subqueries (Cont'd)

✓ Selecting Data

- If you specify the **ANY** keyword before a subquery, the comparison is true if it is true for any of the values that the subquery returns.

Keyword ANY	Signifies…
> ANY(20,30,40) returned from inner query	>20
< ANY(20,30,40) returned from inner query	<40
=ANY(20,30,40) returned from inner query	=20 or =30 or =40

Subqueries (Cont'd)

Selecting Data (Cont'd)

- Example: Are any low-level flight attendants (FA1 or FA2) older than any of the high-level flight attendants (FA3)?

```
proc sql;
title "FA1's or FA2's Older Than ANY FA3's";
   select EmpID, JobCode, DateOfBirth
      from airline.payrollermaster
      where JobCode in ('FA1', 'FA2')
         and DateOfBirth < any
            (select DateOfBirth
               from airline.payrollmaster
               where JobCode='FA3') ;
```

An alternative **WHERE** clause is

✓ **where JobCode in ('FA1' , 'FA2') and DateOfBirth < (select max(DateOfBirth) from..) ;**

Subqueries (Cont'd)

Selecting Data (Cont'd)

- The ALL keyword is true only if the comparison is true for all values returned.

Keyword ALL	Signifies…
> ALL(20,30,40) returned from inner query	>40
< ALL(20,30,40) returned from inner query	<20

Subqueries (Cont'd)

Selecting Data (Cont'd)

- Example: Are there FA1's or FA2's who are older than all of the FA3's?

```
proc sql;
title "FA1's or FA2's Older Than ALL FA3's";
   select EmplD, JobCode, DateOfBirth
      from airline.payrollermaster
      where JobCode in ('FA1', 'FA2')
         and DateOfBirth < all
            (select DateOfBirth
               from airline.payrollmaster
               where JobCode='FA3') ;
```

An alternative **WHERE** clause is

✓ **where JobCode in ('FA1' , 'FA2') and**
DateOfBirth < (select min(DateOfBirth) from..) ;

Combining Tables

- Overview
- Joins
- Complex Joins
- Set Operators

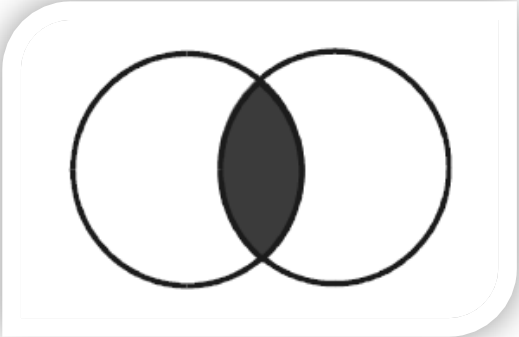
Overview

- ✓ *Joins* combine tables horizontally (side by side).

- ✓ *Set operations* combine tables vertically (one on top of the other).

Joins

Types of Joins



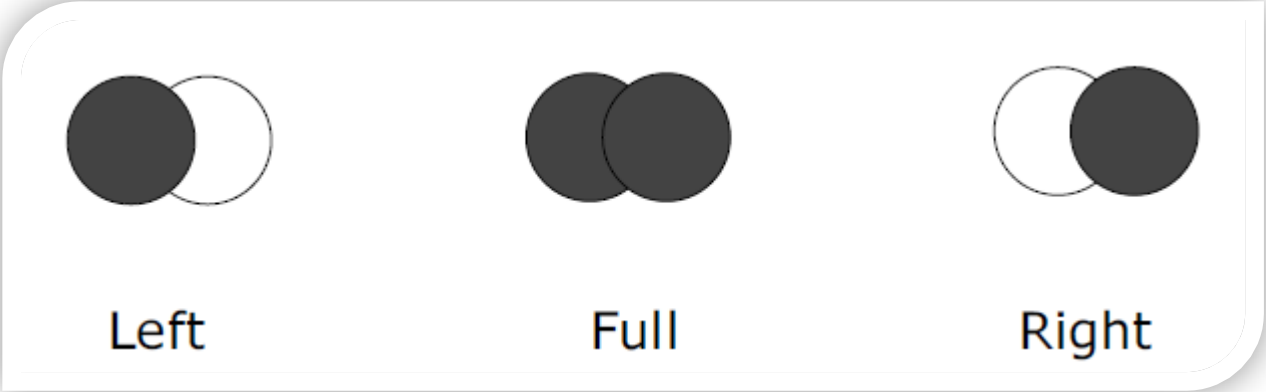
PROC SQL supports two types of joins:

✓ inner joins

- return only matching rows
- allow a maximum of 32 tables to be joined at the same time.

✓ outer joins

- return all matching rows, plus nonmatching rows from one or both tables
- can be performed on only two tables or views at a time.



Joins (Cont'd)

✓ Cartesian Product

A query that lists multiple tables in the **FROM** clause, without row restrictions, results in all possible combinations of rows from all tables.

```
proc sql;
    select *
        from one, two ;
```

Joins (Cont'd)

Cartesian Product (Cont'd)

Table ONE

X	A
1	a
4	d
2	b

Table TWO

X	B
2	x
3	y
5	v

X	A	X	B
1	a	2	x
1	a	3	y
1	a	5	v
4	d	2	x
4	d	3	y
4	d	5	v
2	b	2	x
2	b	3	y
2	b	5	v

✓ The number of rows in a Cartesian product is the product of the number of rows in the contributing tables.

Joins (Cont'd)

✓ Inner Joins

Inner join syntax resembles Cartesian product syntax, but it has a **WHERE** clause that restricts how the rows can be combined.

```
SELECT col1, col2, …
      FROM table1, table2, …
      WHERE join-condition(s)
            <AND other subsetting conditions>
       <other clauses> ;
```

The distinguishing characteristics of inner join syntax are

- ✓ a list of two or more table names in the **FROM** clause
- ✓ one or more join conditions in the **WHERE** clause.

Joins (Cont'd)

Inner Joins (Cont'd)

Conceptually, PROC SQL

- first builds a Cartesian product
- then applies the specified restriction(s) and removes rows.

X	A	X	B
1	a	2	x
1	a	3	y
1	a	5	v
4	d	2	x
4	d	3	y
4	d	5	v
2	b	2	x
2	b	3	y
2	b	5	v

```
select *
        from one, two
        where one.x=two.x ;
```

Joins (Cont'd)

Inner Joins (Cont'd)

Table ONE

X	A
1	a
4	d
2	b

Table TWO

X	A
2	x
3	y
5	v

X	A	X	B
2	a	2	x

```
select *
        from one, two
        where one.x=two.x ;
```

An inner join is sometimes called a conventional join, natural join, or equijoin.

- ✓ Tables do not have to be sorted before they are joined.
- ✓ Column X exists in both tables and occurs twice in the query result.

Joins (Cont'd)

Inner Joins (Cont'd)

Display the X column only once.

Table ONE

X	A
1	a
4	d
2	b

Table TWO

X	A
2	x
3	y
5	v

X	A	X	B
2	a	2	x

```
select one.x, a, b
        from one, two
        where one.x=two.x ;
```

```
select *
        FROM one INNER JOIN two
        ON one.x=two.x ;
```

Joins (Cont'd)

Inner Joins (Cont'd)

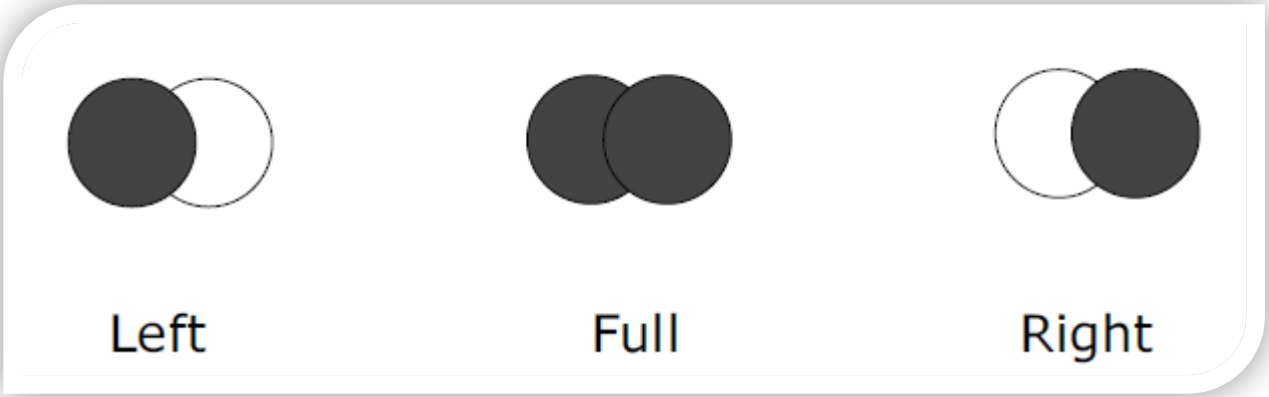
- Example: Display the names, job codes, and ages of all New York employees.
- Employee names are found in the AIRLINE.STAFFMASTER table.
- Employee job codes and birth dates are found in the AIRLINE.OATRIKKNASTER table.

```
proc sql;
title "New York Employees";
   select substr(FirstName, 1, 1) || '.' || LastName as Name,
          JobCode,
          int((today()-DateOfBirth)/365.25) as Age
      from airline.payrollmaster, airline.staffmaster
      where payrollmaster.EmpID=staffmaster.EmpID
            and State = 'NY'
      order by JobCode;
```

Joins (Cont'd)

✓ Outer Joins

You can retrieve nonmatching rows, as well as matching rows, by using an outer join. Outer join are limited to two tables at a time.



Note: An outer join is an augmentation of an inner join. It returns all the rows by an inner join, plus others.

Joins (Cont'd)

Outer Joins (Cont'd)

```
SELECT col1, col2, …
      FROM table1
            LEFT|RIGHT|FULL JOIN table2
            ON join-condition(s)
      <other clauses> ;
```

The distinguishing characteristics of outer join syntax are

- ✓ exactly two table names flanking one of the three **JOIN** operators in the **FROM** clause
- ✓ a special ON clause specifying the join condition(s).
- ✓ a **WHERE** clause is permitted in order to specify general subsetting conditions.

Joins (Cont'd)

Outer Joins (Cont'd)

- A *left join* retrieves matching rows from both tables, plus nonmatching rows from the left table (the first table in the **FROM** clause).

Table ONE

X	A
1	a
2	b
4	d

Table TWO

X	B
2	x
3	y
5	v

X	A	X	B
1	a	.	
2	b	2	x
4	d	.	

```
select *
      from one left join two
      on one.x=two.x ;
```

Joins (Cont'd)

Outer Joins (Cont'd)

- A *right join* retrieves matching rows from both tables, plus nonmatching rows from the right table (the second table in the **FROM** clause).

Table ONE

X	A
1	a
2	b
4	d

Table TWO

X	B
2	x
3	y
5	v

X	A	X	B
2	b	2	x
.		3	y
.		5	v

```
select *
      from one right join two
      on one.x=two.x ;
```

Joins (Cont'd)

Outer Joins (Cont'd)

- A *full join* retrieves matching rows and nonmatching rows from both tables.

Table ONE

X	A
1	a
2	b
4	d

Table TWO

X	B
2	x
3	y
5	v

X	A	X	B
1	a	.	
2	b	2	x
.		3	y
4	d	.	
.		5	v

```
select *
    from one full join two
    on one.x=two.x ;
```

Joins (Cont'd)

✓ Using a Table Alias

- An *alias* is a table nickname. You can assign an alias to a table by following the table name in the **FROM** clause with the **AS** keyword and a nickname for the table. Then use the alias in other clauses of the QUERY statement.
- A table alias is primarily used to reduce the amount of typing required to write a query. It is usually optional. There are, however, two situations that require a table alias:

- a self-join (a table is joined to itself), for example,

```
from airline.staffmaster as s1, airline.staffmaster as as2
```

- when referencing same-named columns from same-named tables in different libraries, for example,

```
from airline.flightdelays as ad,
    work.flightdalays as wd
        where ad.delay > wd.delay
```

Joins (Cont'd)

Using a Table Alias (Cont'd)

```
proc sql;
    select l.date,
           l.flightnumber label='Flight Number',
           l.destination label='Left',
           l.destination label='Right',
           delay
        from airline.marchflights as l
            left join
            airline.flightdalays as r
        on l.date=r.date and
           l.flightnumber=r.flightnumber
        order by delay ;
```

Note: The **AS** keyword is optional in a table alias. The alias can directly follow the table name in the **FROM** clause.

Joins (Cont'd)

✓ SQL Join versus DATA Step Merge

A DATA step with MERGE and BY statements combines rows differently from an outer join.

Table ONE

X	A
1	a
2	b
4	d

Table TWO

X	B
2	x
3	y
5	v

```
data merged;
    merge one two;
    by x;
run;
```

Table MERGED

X	A	B
1	a	
2	b	x
3		y
4	d	
5		v

Joins (Cont'd)

SQL Join versus DATA Step Merge (Cont'd)

Table ONE

X	A
1	a
2	b
4	d

Table TWO

X	B
2	x
3	y
5	v

```
proc sql;
     select one.x, a, b
        from one full join two
        on one.x=two.x;
```

X	A	B
1	a	
2	b	x
		y
4	d	
		v

Note: In the SQL procedure, the two X columns are not overlaid by default.

Joins (Cont'd)

SQL Join versus DATA Step Merge (Cont'd)

You can use the **COALESCE** function to overlay two columns. The COALESCE function

- returns the first value that is a SAS nonmissing value.
- Requires all arguments to have the same data type.

Table ONE

X	A
1	a
2	b
4	d

Table TWO

X	B
2	x
3	y
5	v

```
proc sql;
    select coalesce(one.x, two.x)
                label='x', a, b
        from one full join two
        on one.x=two.x;
```

X	A	B
1	a	
2	b	x
3		y
4	d	
5		v

Note: If you omit the LABEL=option or an alias in a coalesced column, it appears without a column heading.

Joins (Cont'd)

SQL Join versus DATA Step Merge (Cont'd)

Joins do not require

- sorted or indexed tables
- same-named columns in join expressions
- equality in join expressions.

Note: Tables can be joined on inequalities, for example,

```
select columns
  from table1 as a, table2 as b
  where a.itemnumber=b.itemnumber
      and a.cost > b.price ;
```

Complex Joins

✓ In-line Views

An in-line view is

- a temporary table that exists only during query execution
- created when a **FROM** clause contains a query expression in place of a table name.

```
proc sql;
     select *, Late/(Late+Early) as prob
               format=5.2 label='Probability of Delay'
          from (select Destination,
                       avg(Delay) as average
                    format=3.0 label='Average Delay',
                       max(Delay) as max
                    format=3.0 label='Maximum Delay',
                       sum(Delay<=0) as early
                    format=3.0 label='Number of Early Arrivals'
                    from airline.flightdelays
                    group by 1)
          order by 2;
```

Complex Joins (Cont'd)

In-line Views (Cont'd)

Boolean expressions can be used in the **SELECT** clause.

```
proc sql;
     select Delay,
                (Delay > 0) as Late
          from airline.flightdelays;
```

Delay	Late
0	0
8	1
-5	0
18	1

Note: A Boolean expression resolves either to 1 (true) or 0 (false).

Complex Joins (Cont'd)

Handling a Complex Query

What are the names of the supervisors for the crew on the flight to Copenhagen on March 4, 2000?

- Step 1: Identify the crew for the flight.

```
proc sql;
     select EmpID
          from airline.flightschedule
          where Date='04mar2000'd
                    and Destination='CPH'
```

Complex Joins (Cont'd)

Handling a Complex Query (Cont'd)

- Step 2: Find the states and job categories of the crew returned from the first query.

```
proc sql;
     select substr(JobCode,1,2) as JobCategory,
            State
        from airline.staffmaster as s,
             airline.payrollmaster as p
        where s.empid=p.empid and s.empid in
            (select EmpID
                 from airline.flightschedule
                 where Date='04mar2000'd
                    and Destination='CPH');
```

Complex Joins (Cont'd)

Handling a Complex Query (Cont'd)

- Step 3: Find the employee numbers of the crew supervisors based on the states and job categories generated by the second query.

```
proc sql;
    select EmpID
        from airline.supervisors as m,
            (select substr(JobCode,1,2) as JobCategory,
                    State
                from airline.staffmaster as s,
                    airline.payrollmaster as p
                where s.empid=p.empid and s.empid in
                    (select EmpID
                        from airline.flightschedule
                        where Date='04mar2000'd and
                            Destination='CPH')) as c
            where m.jobcategory=c.jobcategory
                and m.state=c.state;
```

Complex Joins (Cont'd)

Handling a Complex Query (Cont'd)

- Step 4: Find the names of the supervisors.

```
proc sql;
    select FirstName, LastName
       from airline.staffmaster where empid in
          (select EmpID
              from airline.supervisors as m,
                 (select substr(JobCode,1,2) as JobCategory,
                         State
                    from airline.staffmaster as s,
                           airline.payrollmaster as p
                    where s.empid=p.empid and s.empid in
                       (select EmpID
                            from airline.flightschedule
                            where Date='04mar2000'd and
                                  Destination='CPH')) as c
       where m.jobcategory=c.jobcategory
          and m.state=c.state);
```

Complex Joins (Cont'd)

Handling a Complex Query (Cont'd)

You can also solve this problem by using a multiway join.

```
proc sql;
    select distinct e.firstname, e.lastname
        from airline.flightschedule as a,
             airline.staffmaster as b,
             airline.payrollmaster as c,
             airline.supervisors as d,
             airline.staffmaster as e
       where a.date='04mar2000'd and
             a.destination='CPH' and
             a.empid=b.empid and
             a.empid=c.empid and
             d.jobcategory=substr(c.jobcode,1,2) and
             d.state=b.state and
             d.empid=e.empid;
```

Complex Joins (Cont'd)

Handling a Complex Query (Cont'd)

Comparison with traditional SAS programs

```
/* Find the crew for the flight. */

proc sort data=airline.flightschedule (drop=flightnumber)
                out=crew (keep=empid);
     where destination='CPH' and date='04mar2000'd;
     by empid;
run;
```

Complex Joins (Cont'd)

Handling a Complex Query (Cont'd)

```
/* Find the State and job code for the crew. */

proc sort data=airline.payrollmaster (keep=empid jobcode)
                out=payroll;
    by empid;
run;

proc sort data=airline.staffmaster (keep=empid state firstname lastname)
                out=staff;
    by empid;
run;

data st_cat (keep=state category);
    merge crew (in=c) staff payroll;
        by empid;
        if c;
        jobcategory=substr(jobcode,1,2);
run;
```

Complex Joins (Cont'd)

Handling a Complex Query (Cont'd)

```
/* Find the supervisor IDs. */

proc sort data=st_cat;
    by jobcategory state;
run;

proc sort data=airline.supervisors
                out=superv;
    by jobcategory state;
run;

data super (keep=empid);
    merge st_cat (in=s) superv;
        by jobcategory state;
        if s;
run;
```

Complex Joins (Cont'd)

Handling a Complex Query (Cont'd)

```
/* Find the names of the supervisors. */

proc sort data=super;
    by empid;
run;

data names (keep=empid);
    merge super (in=super)
          staff (keep=empid firstname lastname);
        by empid;
        if super;
run;

proc print data=names noobs uniform;
run;
```

Complex Joins (Cont'd)

Choosing Between SQL Joins and DATA Step Merges

- DATA step merges are usually more efficient than SQL joins in combining small tables.
- SQL joins are usually more efficient than DATA step merges in combining large, unsorted tables.
- SQL joins are usually more efficient than DATA step merges in combining a large, indexed table with a small table.
- For ad hoc queries, select the method that you can code in the shortest time.
- For production jobs, experiment with different coding techniques and evaluate performance statistics.

Set Operators

Types of Set Operators

Set operators combine rows from two tables vertically.

There are four set operators:

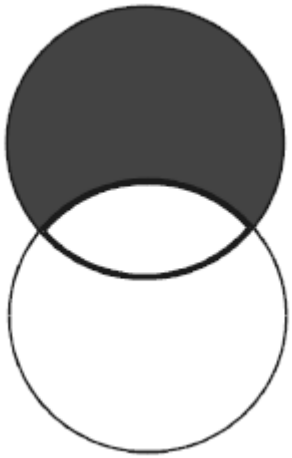
- **EXCEPT**
- **INTERSECT**
- **UNION**
- **OUTER UNION**

- Columns are matched by position and must be the same data type.
- Column names in the result set are determined by the first table.

- All columns from both tables are selected.

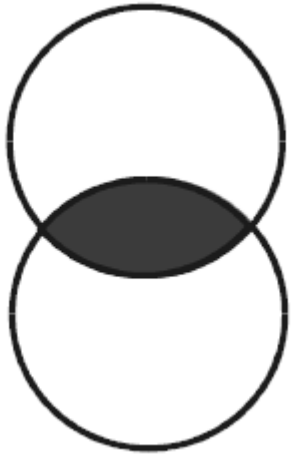
Set Operators (Cont'd)

Types of Set Operators (Cont'd)



EXCEPT

- Unique rows from the first table that are **not** found in the second table are selected.



INTERSECT

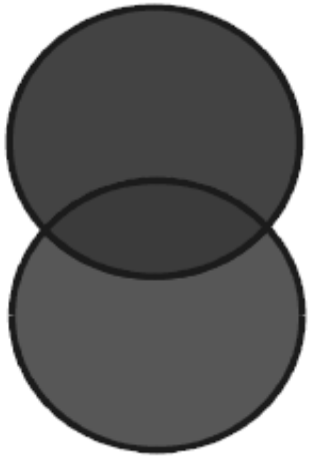
- **Common** unique rows from both tables are selected.

Set Operators (Cont'd)

Types of Set Operators (Cont'd)

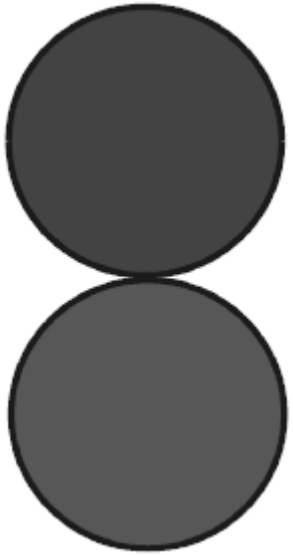
UNION

- **All** unique rows from both tables are selected with columns overlaid.



OUTER UNION

- **All** rows from both tables, **unique as well as non-unique**, are selected.
- Columns are **not** overlaid.



Set Operators (Cont'd)

Modifiers

You can use two keywords to modify the behavior of set operators:

- **ALL**

- does not remove duplicate rows, and so avoids an extra pass through the data. Use the ALL keyword for better performance when it is possible.
- is not allowed in connection with an **OUTER UNION** operator.

- **CORRESPONDING**

- overlays columns by name, instead of by position
- removes any columns in EXCEPT, INTERSECT, and UNION operations
- causes common columns to be overlaid when used in OUTER UNION operations
- can be abbreviated as CORR.

Set Operators (Cont'd)

The EXCEPT Operator

Display the unique rows in table ONE that are not found in table TWO.

Table ONE

X	A
1	a
1	a
1	b
2	c
3	v
4	e
6	g

Table TWO

X	B
1	x
2	y
3	z
3	v
5	w

```
proc sql;
    select *
        from one
    except
    select *
        from two;
```

X	A
1	a
1	b
2	c
4	e
6	g

Note: Duplicate rows are omitted.

Set Operators (Cont'd)

The EXCEPT Operator (Cont'd)

Display the rows (duplicates included) that are found in table ONE but not in TWO.

Table ONE

X	A
1	a
1	a
1	b
2	c
3	v
4	e
6	g

Table TWO

X	B
1	x
2	y
3	z
3	v
5	w

```
proc sql;
    select *
        from one
    except all
    select *
        from two;
```

X	A
1	a
1	a
1	b
2	c
4	e
6	g

Set Operators (Cont'd)

The EXCEPT Operator (Cont'd)

Display the unique rows that exist in table ONE and not in table TWO, based on same-named columns.

Table ONE

X	A
1	a
1	a
1	b
2	c
3	v
4	e
6	g

Table TWO

X	B
1	x
2	y
3	z
3	v
5	w

```
proc sql;
    select *
        from one
    except corr
    select *
        from two;
```

X
4
6

Set Operators (Cont'd)

The EXCEPT Operator (Cont'd)

Example: How many employees have no changes in salary or job code?

```
proc sql;
    select count (*) label='No. of Persons'
        from (select EmpID
                  from airline.staffmaster
              except all
              select EmpID
                  from airline.staffchanges);
```

Set Operators (Cont'd)

The INTERSECT Operator

Display the unique rows common to table ONE and table TWO.

Table ONE

X	A
1	a
1	a
1	b
2	c
3	v
4	e
6	g

Table TWO

X	B
1	x
2	y
3	z
3	v
5	w

```
proc sql;
    select *
        from one
    intersect
    select *
        from two;
```

X	A
3	v

Set Operators (Cont'd)

The INTERSECT Operator (Cont'd)

Display the unique rows common to table ONE and table TWO, based on same-named columns.

Table ONE

X	A
1	a
1	a
1	b
2	c
3	v
4	e
6	g

Table TWO

X	B
1	x
2	y
3	z
3	v
5	w

```
proc sql;
    select *
        from one
    intersect corr
    select *
        from two;
```

X
1
2
3

Set Operators (Cont'd)

The INTERSECT Operator (Cont'd)

Example: What are the names of the old employees who have changed salary or job code?

```
proc sql;
    select FirstName, LastName
        from airline.staffmaster
    intersect all
    select FirstName, LastName
        from airline.staffchanges;
```

Set Operators (Cont'd)

The UNION Operator

Table ONE

X	A
1	a
1	a
1	b
2	c
3	v
4	e
6	g

Table TWO

X	B
1	x
2	y
3	z
3	v
5	w

```
proc sql;
    select *
        from one
    union
    select *
        from two;
```

X	A
1	a
1	b
1	x
2	c
2	y
3	v
3	z
4	e
5	w
6	g

Set Operators (Cont'd)

The UNION Operator (Cont'd)

Table ONE

X	A
1	a
1	a
1	b
2	c
3	v
4	e
6	g

Table TWO

X	B
1	x
2	y
3	z
3	v
5	w

```
proc sql;
    select *
        from one
    union corr
    select *
        from two;
```

X
1
2
3
4
5
6

Set Operators (Cont'd)

The UNION Operator (Cont'd)

Example: Add the miles traveled, bonus points earned, and bonus points used by frequent flyers.

```
proc sql;
   title 'Points and Miles Traveled by Frequent Flyers';
   select 'Total Points Earned :',
          sum(PointsEarned) format=comma12.
      from airline.frequentflyers
           union
   select 'Total Points Used :',
          sum(PointsUsed) format=comma12.
      from airline.frequentflyers
           union
    select 'Total Miles Traveled :',
          sum(MilesTraveled) format=comma12.
      from airline.frequentflyers;
```

Set Operators (Cont'd)

The OUTER UNION Operator

Display all data values from table ONE and table TWO.

Table ONE

X	A
1	a
1	a
1	b
2	c
3	v
4	e
6	g

Table TWO

X	B
1	x
2	y
3	z
3	v
5	w

```
proc sql;
    select *
        from one
    outer union
    select *
        from two;
```

X	A	X	B
1	a	.	
1	a	.	
1	b	.	
2	c	.	
3	v	.	
4	e	.	
6	g	.	
.		1	x
.		2	y
.		3	z
.		3	v
.		5	w

Set Operators (Cont'd)

The OUTER UNION Operator (Cont'd)

Display all data values from table ONE and table TWO, but overlay common columns.

Table ONE

X	A
1	a
1	a
1	b
2	c
3	v
4	e
6	g

Table TWO

X	B
1	x
2	y
3	z
3	v
5	w

```
proc sql;
    select *
        from one
    outer union corr
    select *
        from two;
```

X	A	B
1	a	
1	a	
1	b	
2	c	
3	v	
4	e	
6	g	
1		x
2		y
3		z
3		v
5		w

Set Operators (Cont'd)

The OUTER UNION Operator (Cont'd)

Example: Display the employee numbers, job codes, and salaries of all machines.

```
proc sql;
   select *
      from airline.mechanicslevel1
         outer union corr
   select *
      from airline.mechanicslevel2
         outer union corr
   select *
      from airline.mechanicslevel3;
```

Set Operators (Cont'd)

Comparing Methods of Combining Tables Vertically

The following programs produce the same report:

```
data three;
     set one two;
run;
```

```
proc sql;
     select * from one
          outer union corr
     select * from two;
quit;
```

```
proc append base=one data =two;
run;
proc print data=one noobs;
run;
```

Set Operators (Cont'd)

Comparing Methods of Combining Tables Vertically (Cont'd)

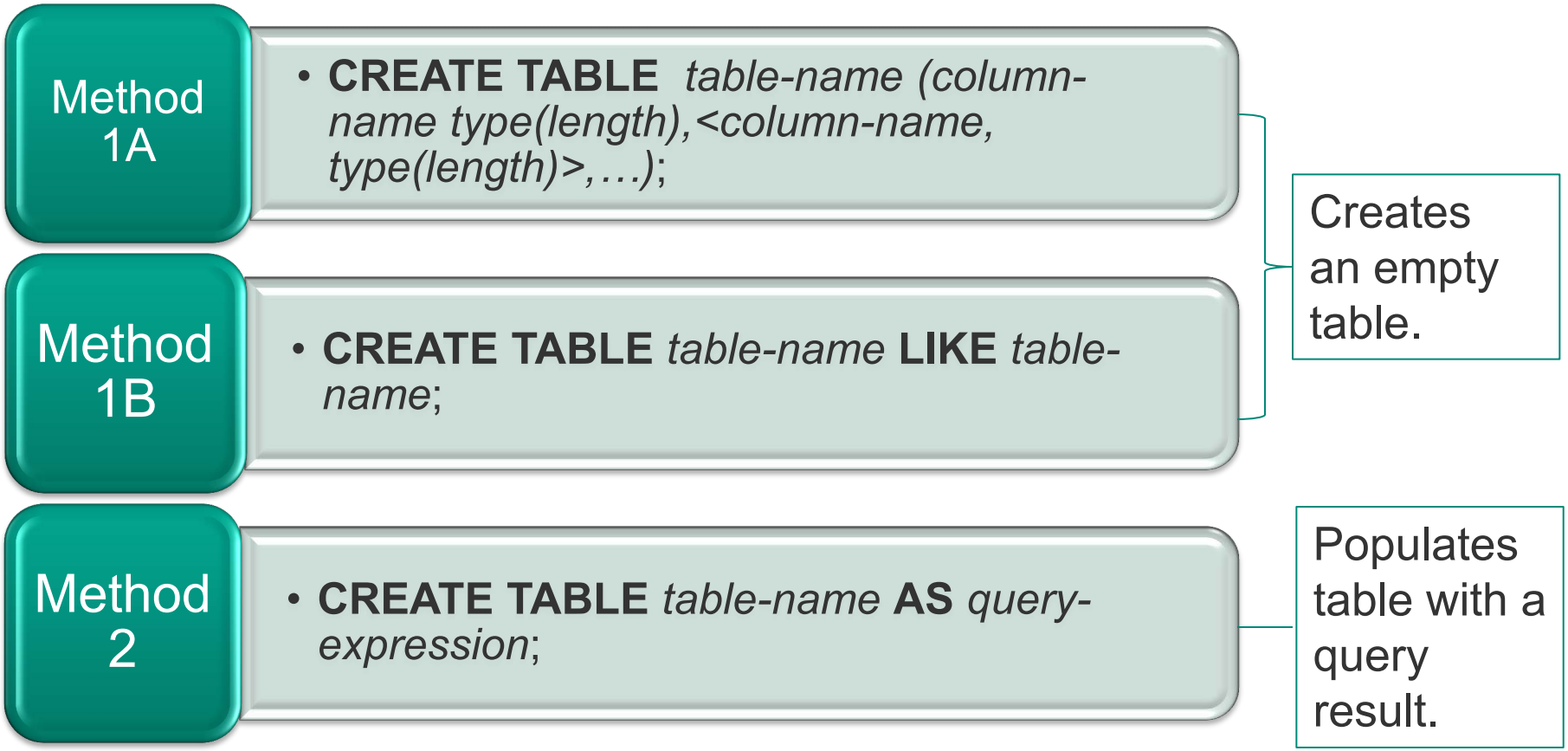
- PROC APPEND is the fastest method of performing a simple concatenation of two tables. The BASE = table is not completely read; only the DATA = table is completely read.
- When logical conditions are involved, you can choose either the DATA step of PROC SQL.
- SQL set operators generally require more computer resources, but are more convenient and flexible, than the DATA step equivalents.
- With the DATA step, you can process an unlimited number of tables at one time.
- With SQL set operators, you can work on only two tables at a time.
- If multiple DATA steps are required to perform the task, consider using PROC SQL.

Creating and Modifying Tables and Views

- Creating Tables
- Creating Views
- Creating Indexes
- Maintaining Tables

Creating Tables

Use the **CREATE TABLE** statement in three ways.



Creating Tables (Cont'd)

Method 1A: Define the columns and fill in the data rows later.

```
proc sql;
    create table x
        (Name char(20),
         BirthDate date,
         Salary num format=comma10.2) ;
```

#	Variable	Type	Len	Format
1	Name	Char	20	
2	BirthDate	Num	8	DATE.
3	Salary	Num	8	COMMA10.2

Note: The table created above does not contain any rows. Use this method when the table you want to create is unlike any other existing table.

Creating Tables (Cont'd)

- Example: Create a table to store discounts for certain destinations and time periods in March. Define columns for destination, discount, and beginning and ending dates of the discount.

```
proc sql;
    create table discount
        (Destination char(3),
          BeginDate date label='BEGINS',
          EndDate date label='ENDS',
          Discount num) ;
```

Creating Tables (Cont'd)

PROC SQL accepts

- types of CHARACTER or VARCHAR, but interprets both as SAS **CHARACTER**. Default length is 8 bytes.
- types of INTEGER, SMALLINT, DECIMAL, NUMERIC, FLOAT, REAL, and DOUBLE PRECISION, interpreting all as SAS **NUMERIC** with a length of 8 bytes.
- a type of DATE, interpreted as a SAS **NUMERIC**, with a length of 8 bytes and a DATE, Informat and format.

Note: Although SAS reads all of the above mentioned data types, only CHARACTER and NUMERIC are used in SAS tables.

Creating Tables (Cont'd)

Method 1B: Copy a table. Use column definitions from another table and fill in the rows of data later.

```
proc sql;
    create table airline.delaycat
        (drop=delayCategory DestinationType)
         like airline.flightdelays ;
```

The columns in this table are copied to the new table.

Note: Use Methods 1A and 1B to create tables containing columns that do not already exist in other tables, that is, define your own columns.

Creating Tables (Cont'd)

Method 2: Store a query result in a table that defines both columns and rows.

```
proc sql;
    create table airline.fa as
        select LastName, FirstName, Salary
          from airline.payrollmaster,
                airline.staffmaster
          where payrollmasater.EmpID
                =staffmaster.EmpID
                and JobCode contains 'FA' ;
        select *
           from airline.fa ;
```

- ✓ This method is particularly helpful when you create subsets or supersets of tables.
- ✓ Use of the CREATE TABLE statement shuts off the automatic report generation. Also, this is the only method of the three that both creates and populates a table at the same time.
- ✓ Use this method when the table you want to create is similar or identical to another existing table.

Creating Tables (Cont'd)

Loading Data into a Table

Once the table is created, you can enter rows of data using the INSERT statement with one of three methods.

Method 1: The SET Clause
INSERT INTO *table-name*
SET *column-name=value, column-name=value, …;*

Method B: The VALUES Clause
INSERT INTO *table-name <(column list)>*
VALUES *(value, value, value, …;)*

Method C: The Query-expression
INSERT INTO *table-name<(column list)>*
SELECT *columns* **FROM** *table -name;*

Creating Tables (Cont'd)

Method A: The SET Clause

```
proc sql;
    insert into discount
        set Destination = 'LHR',
            BeginDate = '01MAR2000'd,
            EndDate = '05MAR2000'd, Discount=.33
        set Destination = 'CPH',
            BeginDate = '03MAR2000'd,
            EndDate = '10MAR2000'd, Discount=.15;
```

You may nest a SELECT statement within a SET statement, as follows:

```
proc sql;
    insert into discount
        set Destination = 'LHR', BeginDate=select max(Date)
            from airline.flightdelays;
```

Creating Tables (Cont'd)

Method B: The VALUES Clause

```
proc sql;
    insert into discount
        values ('LHR', '01MAR2000'd,
                '05MAR2000'd, .33)
        values ('CPH', '03MAR2000'd,
                '10MAR2000'd, .15) ;
```

Creating Tables (Cont'd)

Method C: A Query-expression

```
proc sql;
    insert into discount (Distination, Discount)
        select Destination, Rate*.25
            from work.fares
            where Type = 'special' ;
```

Creating Tables (Cont'd)

Example: Create the discount table, insert four rows of data, and display the table.

```
proc sql;
    create table discount
          (Destination char(3),
           BeginDate date label='BEGINS',
           EndDate date label='ENDS',
           discount num) ;
    insert into discount
    values ('LHR', '01MAR2000'd,  '05MAR2000'd, .33)
    values ('CPH', '03MAR2000'd, '10MAR2000'd, .15)
    values ('CDG', '03MAR2000'd, '10MAR2000'd, .15)
    values ('LHR', '10MAR2000'd, '12MAR2000'd, .05) ;

    select *
         from discount ;
```

Creating Tables (Cont'd)

Integrity Constraints

- *Integrity constraints* are rules that table modifications must follow to guarantee validity of data.
- You can preserve the consistency and correctness of data by specifying integrity constraints for a SAS data file.
- SAS uses the integrity constraints to validate data when you insert or update the values of a variable for which you have defined integrity constraints.

Creating Tables (Cont'd)

Integrity Constraints (Cont'd)

Integrity constraints

- are part of Version 8 of base SAS software
- follow ANSI standards
- cannot be defined for views
- cannot be defined for historical versions of generation data sets
- can be specified when a table is created or later when a table contains data.

Creating Tables (Cont'd)

Five Integrity Constraints

General:

- NOT NULL
- CHECK
- UNIQUE

Referential:

- PRIMARY KEY
- FOREIGN KEY

Creating Tables (Cont'd)

Five Integrity Constraints (Cont'd)

NOT NULL	means that data is required and ensures that corresponding columns have non-missing values in each row.
CHECK	specifies what values may be entered in a column. If a user attempts to enter data that violates this constraint, SAS rejects the value.
UNIQUE	ensures that every value in a column is unique. The same column can be defined as NULL, but only a single null value is allowed per UNIQUE column.
PRIMARY KEY	identifies the column as the table's primary key. Only unique values are permitted and the primary key cannot contain missing values.
FOREIGN KEY	links one or more rows in a table to specific row in another table by matching a foreign key in one table with the primary key in another table. This parent/child relationship limits modifications made to both primary and foreign keys. The only acceptable values for a foreign key are values of the primary key or missing values.

Creating Tables (Cont'd)

Using PROC SQL to Create Integrity Constraints

Example: Re-create the DISCOUNT table with an integrity constraint to limit ticket discounting.

```
proc sql;
    create table discount
       (Destination char(3),
        BeginDate date label='BEGINS',
        EndDate date label='ENDS',
        discount num.
        CONSTRAINT ok_discount check
            (discount le .5)) ;
```

Creating Tables (Cont'd)

Using PROC SQL to Create Integrity Constraints (Cont'd)

Example: Insert two rows using default UNDO_POLICY option (required).

```
proc sql;
    insert into discount
        values ('LHR', '01MAR2000'd,
                '05MAR2000'd, .15)
        values ('CPH', '03MAR2000'd,
                '10MAR2000'd, .55) ;
```

Stockholders may not tolerate excessive airline generosity!

Creating Tables (Cont'd)

Using PROC SQL to Create Integrity Constraints (Cont'd)

Partial Log

```
proc sql;
    insert into discount
        values ('LHR', '01MAR2000'd,
                '05MAR2000'd, .33)
        values ('CPH', '03MAR2000'd,
                '10MAR2000'd, .15) ;
ERROR: Add/Update failed for data set WORK.DISCOUNT
because data value(s) do not comply with integrity constraint
ok_discount.
NOTE: This insert failed while attempting to add data from
VALUES clause 2 to the dataset.
NOTE: Deleting the successful inserts before error noted
above to restore table to a consistent state.
```

0 rows inserted.

Creating Tables (Cont'd)

Rollbacks

If an **INSERT** or **UPDATE** statement experiences an error while it processes the statement, then the inserts or updates that were completed up to the point of the error by that statement can be undone by use of the UNDO_POLICY option.

- UNDO_POLICY=REQUIRED (the default)

  undoes all inserts or updates that have been done to the point of the error. Sometimes the UNDO operation cannot be done reliably.
- UNDO_POLICY=NONE

  prevents any updates or inserts from violating a constraint.
- UNDO_POLICY=OPTIONAL

  reverses any updates or inserts that it can reverse reliably.

Creating Tables (Cont'd)

Rollbacks (Cont'd)

UNDO_POLICY=REQUIRED

PROC SQL performs UNDO processing for INSERT and UPDATE statements.

If the UNDO operation cannot be done reliably, PROC SQL does not execute the statement, and issues an ERROR message.

UNDO cannot be attempted reliably in the following situations:

- A SAS data set opened with CNTLLEV=RECORD can allow other users to update newly inserted records. An error during the insert deletes the record that the other user inserted.
- A SAS/ACCESS view is not able to rollback the changes made by this statement without rolling back other changes at the same time.

Default: UNDO_POLICY=REQUIRED

Creating Tables (Cont'd)

Rollbacks (Cont'd)

UNDO_POLICY=NONE

PROC SQL skips records that cannot be inserted or updated, and writes to the SAS log a warning message similar to that written by PROC APPEND.

UNDO_POLICY=OPTIONAL

PROC SQL performs UNDO processing if it can be done reliably. If the UNDO cannot be done reliably, then no UNDO processing is attempted.

This option is a combination of the first two. If UNDO can be done reliably, then it is done. PROC SQL proceeds as if UNDO_POLICY=REQUIRED is in effect. Otherwise, it proceeds as if UNDO_POLICY=NONE was specified.

Creating Tables (Cont'd)

Using PROC SQL to Create Integrity Constraints (Cont'd)

Example: Insert two rows using UNDO_POLICY=NONE.

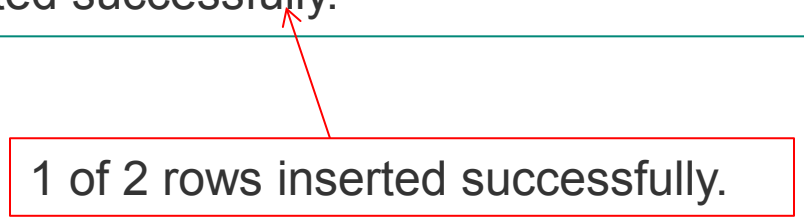
```
proc sql undo_policy=none;
    insert into discount
        values ('LHR', '01MAR2000'd,
                '05MAR2000'd, .15)
        values ('CPH', '03MAR2000'd,
                '10MAR2000'd, .55) ;
```

Creating Tables (Cont'd)

Using PROC SQL to Create Integrity Constraints (Cont'd)

Partial Log

```
WARNING: The SQL option UNDO_POLICY=REQUIRED is
not in effect. If an error is detected when processing this
INSERT statement, that error will not cause the entire
statement to fail.
ERROR: Add/Update failed for data set WORK.DISCOUNT
because data value(s) do not comply with integrity constraint
ok_discount.
NOTE: This insert failed while attempting to add data from
VALUES clause 2 to the dataset.
NOTE: 2 rows were inserted into WORK.DISCOUNT. Of
these 1 row was rejected as an ERROR, leaving 1 row that
was inserted successfully.
```



Creating Tables (Cont'd)

Documenting Table and View Definitions and Integrity Constraints

The DESCRIBE statement displays the definition of the view or CREATE TABLE statement of a table.

General form of the **DESCRIBE** statement:

```
PROC SQL;
    DESCRIBE TABLE table-name,<,table-name>…;
    DESCRIBE VIEW proc-sql-view <,proc-sql-view>…;
    DESCRIBE TABLE CONSTRAINTS table-name
        <,table-name> …;
```

Creating Tables (Cont'd)

Documenting Table Definitions and Integrity Constraints (Cont'd)

Example: Show the constraints for the DISCOUNT table.

```
proc sql;
     describe table discount;
```

- ✓ The **DESCRIBE TABLE** statement (without the **CONSTRAINTS** keyword) writes a **CREATE TABLE** statement to the SAS log for the specified table regardless of how the table was originally created (for example, with a **DATA** step).
- ✓ If the table contains an index, **CREATE INDEX** statements for those indexes are also written to the SAS log.

Creating Views

A **PROC SQL** view

- is a stored query. It contains no rows of data.
- can be used in SAS programs in place of an actual SAS data file.
- can be derived from one or more tables, PROC SQL views, DATA step views, or SAS/ACESS views.
- extracts underlying data when used, thus accessing the most current data.

Creating Views (Cont'd)

Creating a View

General form of the **CREATE VIEW** statement:

```
PROC SQL;
    CREATE VIEW view-name AS query-expression;
```

- ✓ Views are not separate copies of the data and are referred to as virtual tables because they do not exist as independent entities as do real tables. It may be helpful to think of a view as a movable frame or window through which you can see the data.
- ✓ Thus, when the view is referenced by a SAS procedure or in a DATA step, it is executed, and conceptually, an internal table is built. PROC SQL processes this internal table as if it were any other table.

Creating Views (Cont'd)

Creating a View (Cont'd)

Example: Create a view containing personal information for flight attendants. Have the view always return the employee's age as of the current date.

```
proc sql ;
    create view airline.fa as
        select LastName, FirstName, Gender,
               int((today()-DateOfBirth)/365.25) as Age,
               substr(JobCode,3,1) as Level, Salary
          from airline.payrollmaster,
               airline.staffmaster
          where JobCode contains 'FA' and
                staffmaster.EmpID=
                payrollmaster.EmpID ;
```

Creating Views (Cont'd)

Creating a View (Cont'd)

An alternative: Embed the LIBNAME statement within a **USING** clause.

```
PROC SQL;
    CREATE VIEW proc-sql-view AS query-expression
        <USING statement<,libname-clause>…>;
```

This allows you to store a SAS libref in the view and does not conflict with an identically named libref in the SAS session.

Creating Views (Cont'd)

Using a View

Example: Calculate the flight attendants' mean age, by level, using the AIRLINE.FA view.

```
proc tabulate data= airline.fa;
    class Level;
    var Age;
    table Level*Age*mean ;
run;
```

Your view

NOTE: In both of the above examples, it only appears that the PROC SQL view, AIRLINE.FA, is a table because the view name itself is used in the same way as a SAS table name. However, it is **not** a table but a stored query-expression only! Both tables and views are considered SAS data sets.

Creating Views (Cont'd)

Why Use Views?

You can

- access the most current data in changing tables, DATA step views, or SAS/ACCESS views
- pull together data from multiple database tables or even different databases.
- simplify complex query-expressions and prevent users from altering code
- avoid storing a SAS copy of a large table.

Creating Views (Cont'd)

Administering Views

Example: Write the view defnition for AIRLINE.FA to the SAS log.

```
proc sql ;
    describe view airline.fa;
```

NOTE: SQL view AIRLINE.FA is defined as:

Select LastName, FirstName, Gender, INT((TODAY()-DateOfBirth)/365.25) as Age, SUBSTR(JobCode,3,1) as Level, Salary from AIRLINE.PAYROLLMASTER, AIRLINE.STAFFMASTER where JobCode contains 'FA' and (staffmaster.EmpID=payrollmaster.EmpID);

Creating Views (Cont'd)

Administering Views (Cont'd)

Some General Guidelines

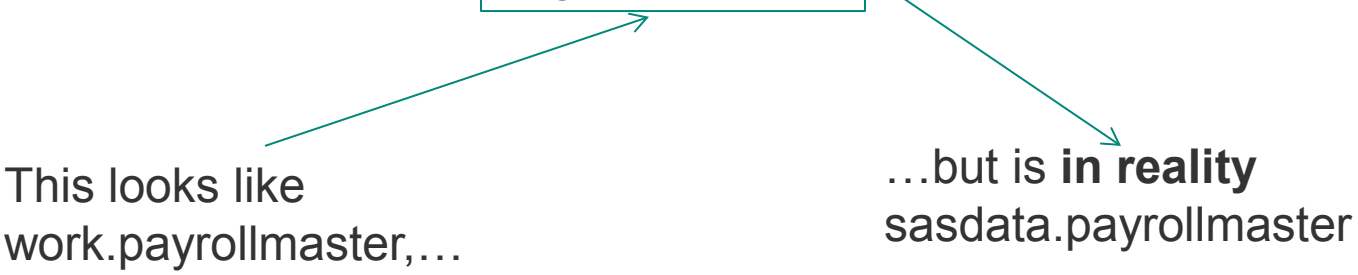
- Avoid the **ORDER BY** clause in a view definition. Otherwise, the data must be stored each time the view is referenced.
- If the same data is used many times in one program, create a table rather than a view.
- Avoid specifying two-level names in the **FROM** clause when you create a permanent view that resides in the same library as the contributing table(s).

Creating Views (Cont'd)

Administering Views (Cont'd)

Example:

```
proc sql ;
    create view sasdata.master as
        select *
            from payrollmaster;
```



Creating Views (Cont'd)

Administering Views (Cont'd)

Using the Embedded LIBNAME Statement

```
libname sasdata 'SAS-data library one';
libname airline   'SAS-data library two';
proc sql ;
    create view sasdata.journeymen as
        select *
            from airline.payrollmaster
            where jobcode like '  2'
            using libname airline 'SAS-data library three';
quit;
proc print data=sasdata.journeymen;
```

3) …overriding any earlier assignment for the duration of the view's execution.

2) …libref AIRLINE becomes active…

1) While the view SASDATA.JOURNEYMEN is executing…

4) After view executes, original libref assignment (3) is re-established and embedded assignment (2) is cleared.

Creating Indexes

An *index* is an auxiliary data structure that specifies the location of rows based on the values of one or more **key** columns.

You can use indexes for subsetting, grouping, and joining tables.

- Indexes provide fast access to small subsets of data…

```
proc sql ;
    select *
       from airline.payrollmaster
       where JobCode= 'NA1' ;
```

One of many values of the variable JobCode

NOTE: A small subset is <=15%.

Creating Indexes (Cont'd)

- … and also enhance join performance.

```
proc sql ;
    select *
        from airline.payrollmaster,
             airline.staffmaster
        where staffmaster.EmpID=
              payrollmaster.EmpID;
```

NOTE: When you subset data, you can select an index to optimize not only a **WHERE** clause with an equals comparison, but also a WHERE clause with the **TRIM** or **SUBSTR** function or the **CONTAINS** or **LIKE** operator.

Creating Indexes (Cont'd)

Index Terminology

Two types of indexes are

simple	based on values of only one column
composite	based on values of more than one column concatenated to form a single value, for example, Date and FlightNumber.

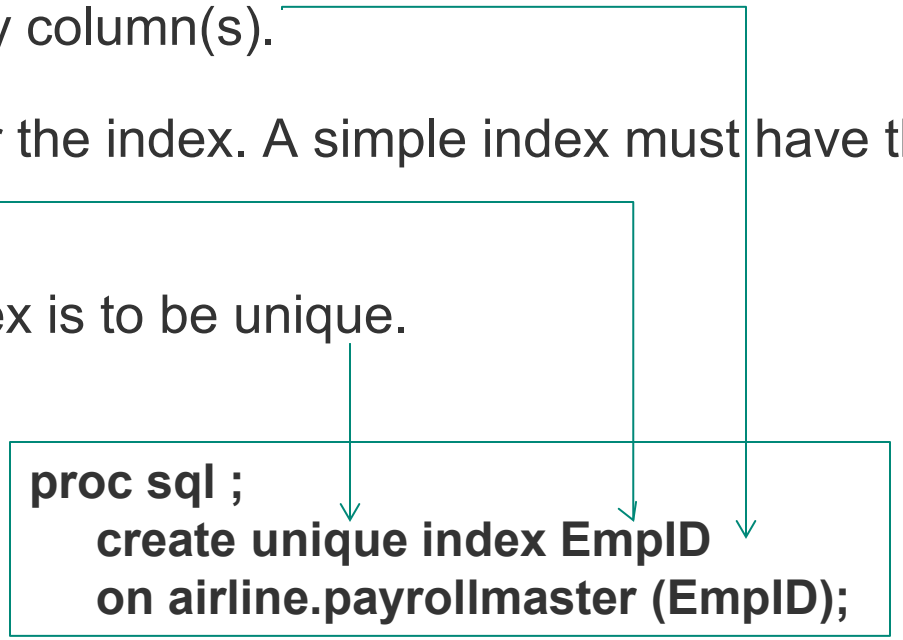
A table can have

- multiple simple and composite indexes
- character and numeric key columns.

Creating Indexes (Cont'd)

Creating an Index

- Designate the key column(s).
- Select a name for the index. A simple index must have the same name as the column.
- Specify if the index is to be unique.



Creating Indexes (Cont'd)

Creating an Index

Gerenal form of the **CREATE INDEX** statement:

```
PROC SQL;
    CREATE <UNIQUE> INDEX index-name
        ON table-name(column-name, column name);
```

Precede the **INDEX** keyword with the **UNIQUE** keyword to define a unique index.

Creating Indexes (Cont'd)

Creating an Index (Cont'd)

Additional notes:

- Indexes can be based on either a character or numeric variable.
- You do not want to create two indexes on the same variable.
- You can achieve improved index performance if you create the index on a pre-sorted data set.
- A composite index cannot have the same name as a variable.

Creating Indexes (Cont'd)

Creating an Index (Cont'd)

Example: This simple index is based on EmpID and allows no duplicate ID numbers in the table.

```
proc sql;
    create unique index EmpID
    on airline.payrollmaster (EmpID);
```

Names must match for a simple index!

Creating Indexes (Cont'd)

Creating an Index (Cont'd)

Example: This composite index named DAILY is based on FLIGHTNUMBER and DATE.

```
proc sql ;
    create unique index daily
       on airline.marchflights(FlightNumber, Date);
```

Creating Indexes (Cont'd)

Indexing and Performance

Example: An index was created for the JobCode column of AIRLINE.PAYROLLMASTER. Use the MSGLEVEL=I system option to determine which queries used the index.

```
options msglevel=I;
proc sql ;
    select *
       from airline.payrollmaster
       where JobCode='NA1';
INFO: Index JobCode selected for WHERE clause optimization.
    select *
        from airline.payrollmaster
        where salary gt 100000;
```

NOTE: To determine if an index is used, specify the SAS system option MSGLEVEL=I. A note appears in the SAS log when an index is selected for processing.

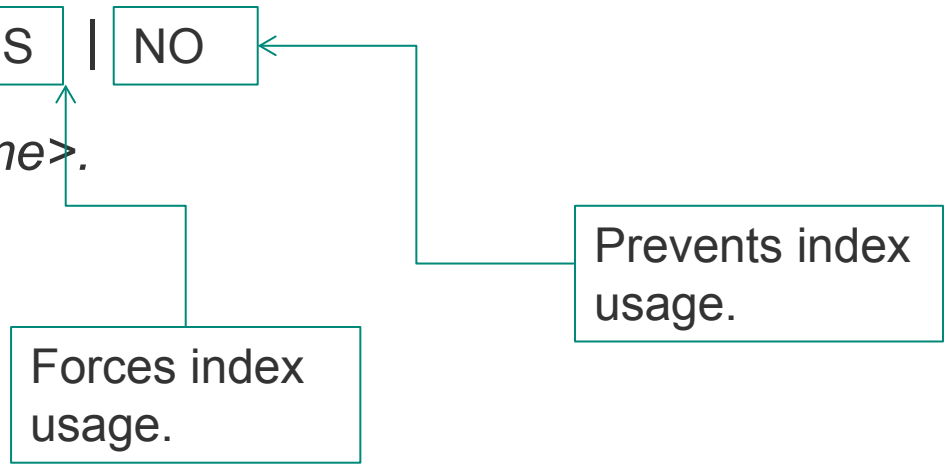
Creating Indexes (Cont'd)

Indexing and Performance (Cont'd)

Control Over Index Usage in a WHERE Expression

Two data set options control the use of indexes:

- IDXWHERE= YES | NO
- INXNAME=<*name*>.



Creating Indexes (Cont'd)

Indexing and Performance (Cont'd)

- When the IDXWHERE=option is

YES SAS uses the best available index to process the WHERE expression, even if SAS estimates that sequential processing is faster.

NO SAS processes the data sequentially even if SAS estimates that processing with an index is better.

- When the IDXNAME=option is

<*name*> SAS uses the named index regardless of performance estimates.

If you do not use the IDXWHERE=option, SAS chooses whether to use an index. You can use either the IDXWHERE=or the IDXNAME=data set option, but not both.

Creating Indexes (Cont'd)

Indexing and Performance (Cont'd)

Suggested guidelines for using indexes:

- Keep the number of indexes to minimum to reduce disk storage and update costs.
- Do not create an index for small tables; sequential access is faster on small tables.
- Do not create an index based on columns with a very small number of distinct values, for example, Male and Female.
- An index performs best when it retrieves a relatively small number of rows, that is, <15%.

Creating Indexes (Cont'd)

Indexing and Performance (Cont'd)

Tradeoffs

Benefits	Costs
• Fast access to a small subset of data (<15%). • Equijoins can be performed without internal sorts. • Can enforce uniqueness. • BY group processing without sorting.	• Extra CPU cycles and I/O operations to create an index. • Extra disk space to store the index file. • Extra memory to load index pages and code for use. • Extra CPU cycles and I/O operations to maintain the index.

Maintaining Tables

You can use PROC SQL to

- **modify values** in a table or view
- **add rows** to a table or view
- **delete rows** from a table or view
- **alter column attributes** of a table
- **add new columns** to a table
- **drop columns** from a table
- **delete** an entire table, SQL view, or index.

Maintaining Tables (Cont'd)

Updating Data Values

Use the **UPDATE** statement to modify column values in existing rows of a table or SAS/ACCESS view.

General form of the **UPDATE** statement:

```
PROC SQL;
      UPDATE table-name
          SET column-name=expression,
          SET column-name=expression,…
          WHERE expression;
```

Careful! If you omit the WHERE expression, all rows are updated.

Maintaining Tables (Cont'd)

Updating Data Values (Cont'd)

x	y
1	a1
2	b1
3	a2
4	b2

```
update one
   set x=x*2
   where y contains 'a';
```

x	y
2	a1
2	b1
6	a2
4	b2

Maintaining Tables (Cont'd)

Updating Data Values (Cont'd)

Example: Give all level 1 employees a 5% raise.

```
proc sql;
      update airline.payrollmaster
         set Salary=Salary * 1.05
         where JobCode like '__1';
       select *
         from airline.payrollmaster;
```

NOTE: You cannot create additional columns using the **UPDATE** statement.

A SAS DATA step equivalent is as follows:

```
data  airline.test;
    modify airline.payrollmaster;
       if substr(JobCode,3)='1' then
         Salary=Salary * 1.05;
run;
```

Maintaining Tables (Cont'd)

Conditional Processing

Use a **CASE** expression to perform conditional processing. Assign new salaries based on job level. Two methods are available.

- Method 1:

```
proc sql;
      update airline.payrollmaster
          set Salary=Salary *
                case substr(JobCode,3,1)
                    when '1' then 1.05
                    when '2' then 1.10
                    when '3' then 1.15
                    else 1.08
                end;
```

Maintaining Tables (Cont'd)

Conditional Processing (Cont'd)

- Method 2:

```
proc sql;
      update airline.payrollmaster
          set Salary=Salary *
                case when substr(JobCode,3,1)='1'
                          then 1.05
                     when substr(JobCode,3,1)='2'
                          then 1.10
                     when substr(JobCode,3,1)='3'
                          then 1.15
                      else 1.08
                end;
```

NOTE: Method 1 is more efficient because the SUBSTR function is evaluated only once. This method also assumes an =comparison operator, which means that if you need a different operator, you must use Method 2.

Maintaining Tables (Cont'd)

Conditional Processing (Cont'd)

You can also use a **CASE** expression in other parts of a query, such as within a **SELECT** statement, to create new columns.

General form of the **CASE** expression within the **SELECT** statement:

```
PROC SQL;
     SELECT column <,column> …
          CASE <case-operand>
          WHEN when-condition THEN result-expression
          <WHEN when-condition THEN result-expression>
           <ELSE result-expression>
END;
```

Maintaining Tables (Cont'd)

Conditional Processing (Cont'd)

Example: Display employee names, job codes, and job levels.

```
proc sql;
    select LastName, FirstName, JobCode,
           case substr (JobCode,3,1)
              when '1' then 'junior'
              when '2' then 'intermediate'
              when '3' then 'senior'
              else 'none'
            end as level
    from airline.payrollmaster,
         airline.staffmaster
    where staffmaster.EmpID=
          payrollmaster.EmpID;
```

Maintaining Tables (Cont'd)

Deleting Rows

Use a DELETE statement to eliminate unwanted rows from a table or SAS/ACCESS view.

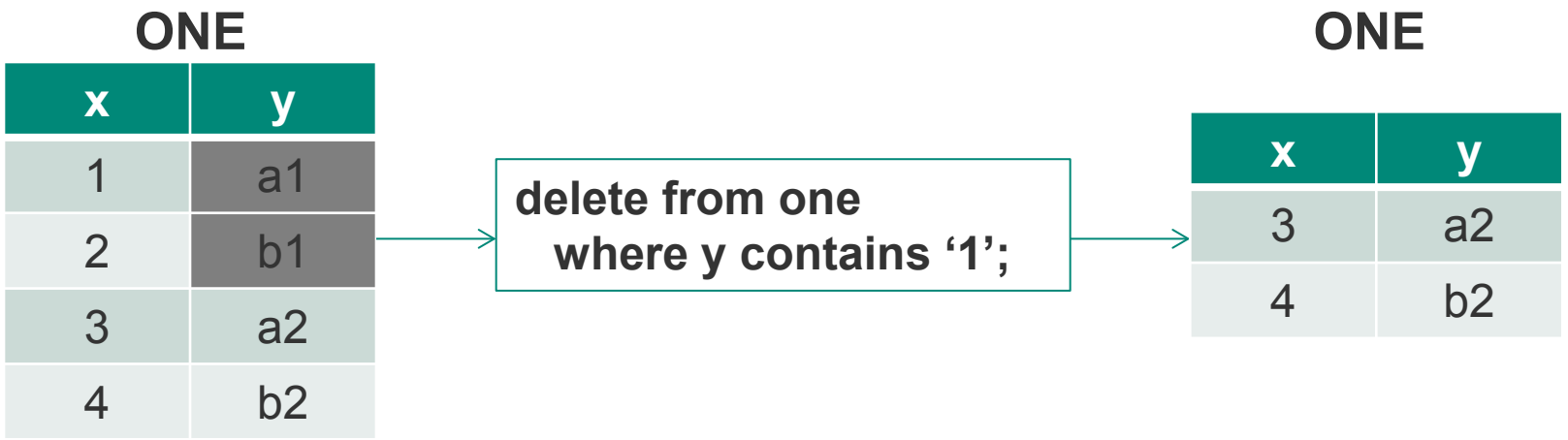
General form of the DELETE statement:

```
PROC SQL;
      DELETE FROM table-name
         WEHRE expression;
```

NOTE: If you do not specify a **WHERE** clause, all rows are deleted.

Maintaining Tables (Cont'd)

Deleting Rows (Cont'd)



Maintaining Tables (Cont'd)

Deleting Rows (Cont'd)

Example: From the AIRLINE.FREQUENTFLYERS table, delete all frequent flyers who have either used up their points or used more than they have.

```
proc sql;
    delete from airline.frequentflyers
        where PointsEarned-PointsUsed <=0;
```

Partial log

NOTE: 11 rows were deleted from AIRLINE.FREQUENTFLYERS.

Maintaining Tables (Cont'd)

Altering Columns

Use the **ALTER** statement to manipulate columns in a table three different ways.

General form of the **ALTER** statement:

```
PROC SQL;
      ALTER TABLE table-name
          ADD column-definition, column-definition,…
          DROP column-name, column-name,…
          MODIFY column-definition, column-definition,…;
```

Maintaining Tables (Cont'd)

Altering Columns (Cont'd)

- Add columns to a table.

```
proc sql;
    alter table airline.payrollmaster
        add Bonus num format=comma10.2,
            Level char(3) ;
```

You are enlarging the table.

NOTE: After adding columns, use the **UPDATE** statement to assign values to those columns. These added columns initially contain missing values.

Maintaining Tables (Cont'd)

Altering Columns (Cont'd)

- Drop columns from a table.

```
proc sql;
    alter table airline.payrollmaster
        drop DestinationType;
```

You are shrinking the table.

An alternative is to use the DROP=data set option as follows:

```
create table airline.flightdelays
     select *
          from airline.flightdelays(drop=destype);
```

Maintaining Tables (Cont'd)

Altering Columns (Cont'd)

- Modifying attributes of existing columns in a table. You can alter a column's length, informat, format, and label.

```
proc sql;
    alter table airline.payrollmaster
        modify Bonus num format=comma8.2,
               Level char(1)
                   label='Emplyee Level';
```

Maintaining Tables (Cont'd)

Altering Columns (Cont'd)

Example: Alter AIRLINE.PAYROLLMASTER as follows:

1. Add a new column named Age.
2. Change the DateOfBirth column to the MMDDYY10. format.
3. Drop the DateOfHire column.

Create the columns here.

```
proc sql;
   alter table airline.payrollmaster
       add Age num
       modify DateOfBirth date format=mmddyy10.
       drop DateOfHire;
    update airline.payrollmaster
        set age=int((today()-DateOfBirth)/365.25) ;
```

Populate the rows here.

Maintaining Tables (Cont'd)

Deleting Table, Indexes, and Views

Use the **DROP** statement to delete an entire table, SQL view, or index.

General form of the **DROP** statement:

```
PROC SQL;
     DROP TABLE table-name, table-name,…;
     DROP VIEW view-name, view-name,…;
     DROP INDEX index-name, index-name,…;
          FROM table-name;
```

Maintaining Tables (Cont'd)

Deleting Table, Indexes, and Views (Cont'd)

Example: Delete the index EmpID from the AIRLINE.PAYROLLMASTER table and delete the temporary table DISCOUNT.

```
proc sql;
    drop index EmpID
        from airline.payrollmaster;

NOTE: Index EmpID has been dropped.
     drop table Discount;

NOTE: Table WORK.DISCOUNT has been dropped.
```

NOTE: When you delete a table, all indexes on that table are automatically deleted. If you copy a table, all indexes are copied.

Maintaining Tables (Cont'd)

In Summary

- UPDATE
  - -SET
  - -CASE

  Modifies values in existing columns, that is, changes row values.
- ALTER
  - -ADD
  - -DROP
  - -MODIFY

  Adds or drops columns, or changes column attributes.
- INSERT
  - -SET
  - -VALUES

  Inserts data rows at end of existing tables.
- DELETE

  Removes rows as specified with a WHERE clause.
- DROP

  Deletes an entire table, view, or index.

Maintaining Tables (Cont'd)

Updating Views

You can update the data underlying PROC SQL views using the INSERT, DELETE, and UPDATE statements, but

- you can only update a single table through a view. It cannot be joined or linked to another table, nor contain a subquery.
- you can update a column using the column's alias, but not a derived column.
- you cannot update the table through a summary query.
- you cannot update a view containing an **ORDER BY** clause.

Maintaining Tables (Cont'd)

Updating Views (Cont'd)

Creating a view…

```
proc sql;
    create view airline.raise as
        select EmpID, JobCode, Salary,
               Salary/12 as MonthlySalary format=dollar12.
          from airline.payrollmaster;
```

… and then update the view.

```
proc sql;
    update airline.raise
        set Salary=Salary * 1.20
         where JobCode='PT3';
```

Additional SQL Features

- Setting PROC SQL Options
- Dictionary Tables and Views
- Interfacing PROC SQL with Macro Language
- Program Testing and Performance

Setting PROC SQL Options

Controlling Processing

The SQL procedure offers a variety of options and statements that affect processing.

General form of the PROC SQL statement:

PROC SQL *options;*

Selected options:

INOBS=*n*	sets a limit of *n* rows from each source table that contributes to a query.
OUTOBS=*n*	restricts the number of rows that a query outputs (displays or writes to a table).
LOOPS=*n*	restricts the number of iterations of the inner loop of PROC SQL.

Setting PROC SQL Options (Cont'd)

Controlling Processing (Cont'd)

NOPROMP|PROMPT	modifies the effect of the INOBS=, OUTOBS=, and LOOPS= options so that you are prompted to stop or continue when a specified limit is reached.
PRINT|NOPRINT	controls whether the results of a SELECT statement are displayed.
NONUMBER|NUMBER	controls whether the row number is printed as the first column in the output.
NODOUBLE|DOUBLE	double-spaces the report.
NOFLOW|FLOW| FLOW=*n*|FLOW=*n m*	controls the appearance of wide character columns. The FLOW option causes text to be flowed in its column rather than wrapping the entire row. Specifying *n* determines the width of the flowed column. Specifying *n* and *m* floats the width of the column between the limits to achieve a balanced layout.

Setting PROC SQL Options (Cont'd)

Controlling Processing (Cont'd)

Example: Display the AWARDS table with flowed character columns and double-spacing.

```
proc sql flow= 1 3
double;
    select *
        from awards;
```

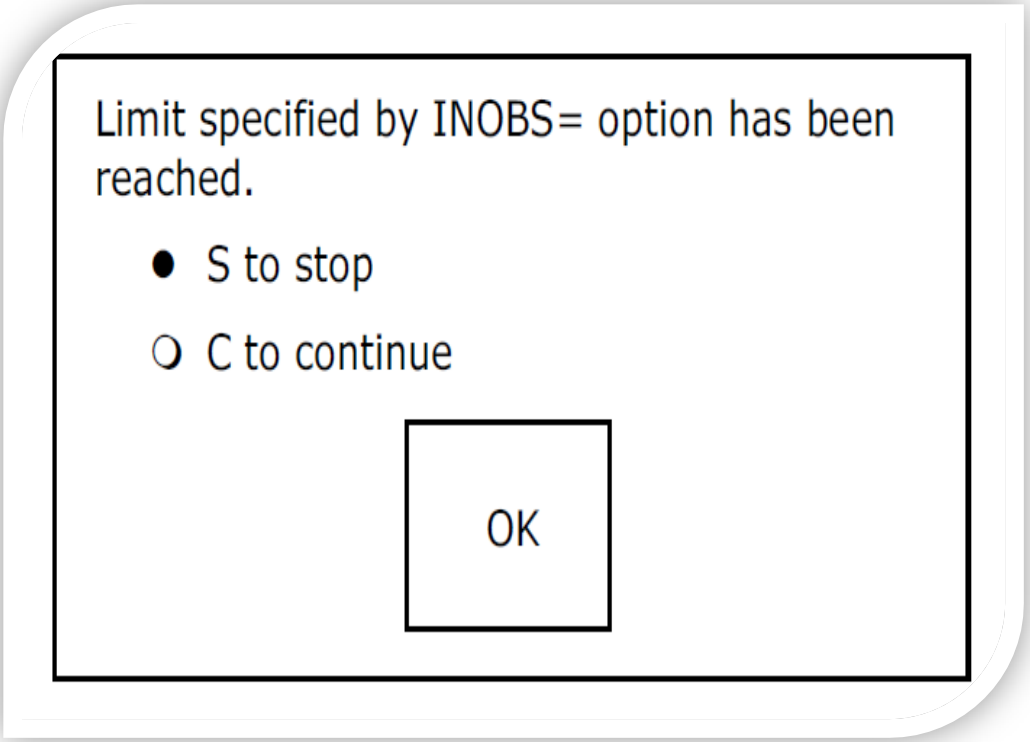
Points Required	Rank	Award
2000	1	free night in hotel
10000	2	50% discount on flight
20000	3	free domestic flight
40000	4	free international flight

Setting PROC SQL Options (Cont'd)

Controlling Processing (Cont'd)

Example: Prompt the user to stop or continue processing after 10 rows are read from AIRLINE.MARCHFLIGHTS.

```
proc sql inobs=10 prompt;
    select FlightNumber,
           Date
      from airline.marchflights;
```



Setting PROC SQL Options (Cont'd)

Resetting Options

You can use the RESET statement to add or change PROC SQL options without re-invoking the procedure.

General form of the RESET statement:

RESET *options;*

Setting PROC SQL Options (Cont'd)

Resetting Options (Cont'd)

Example: Display two rows from the payroll table and print the row number. Then display the rows without printing the row number.

```
proc sql outobs=2 number;
   select * from airline.payrollmaster;
```

Row	Emp ID	Gender	Job Code	Salary	DateOfBirth	DateOfHire
1	1919	M	TA2	$48,126	16SEP1958	07JUN1985
2	1653	F	ME2	$49,151	19OCT1962	12AUG1988

Dictionary Tables and Views

Overview

You can retrieve information about SAS session metadata by querying *dictionary tables* with PROC SQL. Dictionary tables are

- created at initialization
- updated automatically
- limited to read-only access.

The metadata available in dictionary tables includes

- SAS files
- external files
- system options, macros, titles, and footnotes.

Dictionary Tables and Views (Cont'd)

Overview (Cont'd)

SAS File Metadata

DICTIONARY.MEMBERS	general information about data library members
DICTIONARY.TABLES	detailed information about data sets
DICTIONARY.COLUMNS	detailed information on variables and their attributes
DICTIONARY.CATALOGS	information about catalog entries
DICTIONARY.VIEWS	general information about data views
DICTIONARY.INDEXES	information on indexes defined for data files

Dictionary Tables and Views (Cont'd)

Overview (Cont'd)

Other Metadata

DICTIONARY.EXTFILES	currently assigned filerefs
DICTIONARY.OPTIONS	current settings of SAS system options
DICTIONARY.MACROS	information about macro variables
DICTIONARY.TITLES	text assigned to titles and footnotes

NOTE: SAS librefs are limited to 8 characters. DICTIONARY is an automatically assigned, reserved word.

Dictionary Tables and Views (Cont'd)

Exploring Dictionary Tables

describe table dictionary.tables;

Partial Log

```
create table DICTIONARY.TABLES (
      libname char(8) label='Library Name',
      memname char(32) label='Member Name',
      memtype char(8) label='Member Type',
      memlabel char(256) label='Dataset Label',
      typemem char(8) label='Dataset Type',
      create num format=DATETIME informat=DATETIME
label='Date Created',…);
```

Dictionary Tables and Views (Cont'd)

Using Dictionary Information

Example: Display information about the files in the AIRLINE library.

```
options nolabel nocenter;
select memname format=$20.,
       nobs,
       nvar,
       crdate
    from dictionary.tables
    where libname='AIRLINE';
```

Example: Determine which tables contain the EmpID column.

```
select memname
    from dictionary.columns
    where libname='AIRLINE'
        and name='EmpID';
```

Dictionary Tables and Views (Cont'd)

Using Dictionary Information (Cont'd)

To use session metadata in other procedures or in a DATA step, you can

- create a PROC SQL view based on a dictionary table
- use views provided in the SASHELP library that are based on the dictionary tables.

Dictionary Tables and Views (Cont'd)

Using Dictionary Information (Cont'd)

Example: Use SASHELP.VMEMBER to extract information from DICTIONARY.MEMBERS in a PROC TABULATE step.

```
proc tabulate
      data=sashelp.vmember format=8.;
   class libname memtype;
   keylabel N='  ';
   table libname, memtype/
            rts=10 misstext='None';
run;
```

Interfacing PROC SQL with Macro Language

Resolving Symbolic References

Macro variable references embedded within PROC SQL code are resolved as the source code is tokenized.

```
%let datasetname=payrollmaster;
%let bigsalary=100000;
```

SYMBOL TABLE	
Name	Value
Datasetname	payrollmaster
bigsalary	100000

```
select *
   from airline.&datasetname
   where Salary>&bigsalary;
```

Interfacing PROC SQL with Macro Language (Cont'd)

Creating Macro Variables

- SQL allows a query to pass data values to variables in the host software system. The SAS System chose to implement these host variables as macro variables.
- **PROC SQL** can create or update macro variables using an **INTO** clause. This clause can be used in three ways.
- **PROC SQL** can create or update macro variables in either local or global symbol tables.
- The **INTO** clause occurs between the **SELECT** and **FROM** clauses. It cannot be used in a **CREATE TABLE** or **CREATE VIEW** statement. Use the **NOPRINT** option if you do not need a display of the query result.

Interfacing PROC SQL with Macro Language (Cont'd)

Creating Macro Variables: Method 1

General form of the **SELECT** statement with an **INTO** keyword:

```
SELECT col1,col2,…
    INTO :mvar1, :mvar2,…
    FROM …
```

Method 1 extracts values **only** from the **first** row of the query result.

NOTE: This method is often used with queries that return only one row.

Interfacing PROC SQL with Macro Language (Cont'd)

Creating Macro Variables: Method 1 (Cont'd)

Example

```
reset  noprint;
select avg(salary),
          min(salary),
          max(salary)
     into :mean, :min, :max
     from airline.payrollmaster;
  %put &mean &min &max;
```

Interfacing PROC SQL with Macro Language (Cont'd)

Creating Macro Variables: Method 1 (Cont'd)

Example: Calculate the average salary of employees with a particular job code. Store the average in a macro variable and use the average to display all employees in that job code who have a salary above the average. Place the average in a title.

```
%let code=NA1;
select avg(Salary) into :mean
    from airline.payrollmaster
    where JobCode='&code';

reset print;
title1 '&code Employees Earning Above-Average Salaries';
title2 'Average Salary for &code Employees Is &mean';
select *
     from airline.payrollmaster
     where  Salary>&mean and JobCode='&code';
```

Interfacing PROC SQL with Macro Language (Cont'd)

Creating Macro Variables: Method 2

General form of the **SELECT** statement to create a macro variable:

```
SELECT a, b,…
    INTO :a1- :an, :b1- :bn
    FROM …
```

Method 2 extracts values from the **first** *n* rows of the query result, and puts them into a series of *n* macro variables.

Interfacing PROC SQL with Macro Language (Cont'd)

Creating Macro Variables: Method 2 (Cont'd)

Example: How many frequent flyers are in each of the three member types (GOLD, SILVER, BRONZE)?

```
reset print;
select MemberType,
          count (*) as Frequency
     into :memtype1- :memtype3,
          :freq1- :freq3
     from airline.frequentflyers
     group by MemberType;

%put Member types: &memtype1 &memtype2 &memtype3;
%put Frequencies: &freq1 &freq2 &freq3;
```

Interfacing PROC SQL with Macro Language (Cont'd)

Creating Macro Variables: Method 3

General form of the **SELECT** statement to create a macro variable:

```
SELECT col1,col2,…
    INTO :mvar1, :mvar2,…
       SEPARATED BY 'delimiter'
    FROM …
```

Method 3 extracts values from all rows of the query result, and puts them into a single macro variable, separated by the specified delimiter.

Interfacing PROC SQL with Macro Language (Cont'd)

Creating Macro Variables: Method 3 (Cont'd)

Example: Put the unique values of all international destinations into a single macro variable.

```
select distinct Destination
      into :airportcodes
            separated by ' '
      from airline.internationalflights

%put &airportcodes;
```

Interfacing PROC SQL with Macro Language (Cont'd)

Automatic Macro Variables

Execution of a PROC SQL query or non-query statement updates the following automatic macro variables:

SQLOBS	records the number of rows output or deleted
SQLRC	contains the return code from each SQL statement
SQLOOPS	contains the number of iterations processed by the inner loop of PROC SQL.

Interfacing PROC SQL with Macro Language (Cont'd)

Automatic Macro Variables (Cont'd)

Macro Program Example: Write a macro that accepts a state code as a parameter and creates a table containing employees from that state. Display a maximum of 10 rows from the table.

```
%macro state(st);
proc sql;
create table &st as
select LastName, FirstName
     from airline.staffmaster
     where State='&st';
%put NOTE: The table &st has
&sqlobs rows. ;
```

```
title1 '&st Employees';
%if &sqlobs>10 %then %do;
     %put NOTE: Only the first 10
          rows are displayed.;
     title2 'NOTE: Only 10 rows
          are displayed.';
     reset outobs=10;
%end;
select * from &st;
quit;
%mend state;
%state(NY) ;
```

Program Testing and Performance

Testing and Performance Options

PROC SQL statement options are available to aid in testing programs and evaluating performance. The following are selected options:

- EXEC|NOEXEC controls whether submitted SQL statements are executed.
- NONSTIMER|STIMER reports performance statistics in the SAS log for each SQL statement.
- NONERRORSTOP|ERRORSTOP is used in batch and noninteractive jobs to make PROC SQL enter syntax-check mode after an error occurs.

NOTE: To use the STIMER SQL option, the system option STIMER or FULLSTIMER must also be in effect.

Program Testing and Performance (Cont'd)

Testing and Performance Options (Cont'd)

Example: Display the columns that are retrieved when you use SELECT * in a query and display any macro variable resolutions, but do not execute the query.

```
%let datasetname=payrollmaster;

proc sql
     feedback
     noexec;
  select *
        from airline.&datasetname;
```

Program Testing and Performance (Cont'd)

Testing and Performance Options (Cont'd)

Example: This is a log from a **PROC SQL** step with the **STIMER** statement option, executing a single query. The first note concerns the invocation of **PROC SQL**.

```
NOTE: The SQL statement used the following resources:
      CPU     time -          00:00:00.01
      Elapsed time -          00:00:00.68
      EXCP count    - 28
      Task memory   - 110K (20K data, 90K program)
      Total memory - 864K (760K data, 104K program)
```

Program Testing and Performance (Cont'd)

Testing and Performance Options (Cont'd)

The second note concerns the query itself.

```
NOTE: The SQL statement used the following resources:
      CPU      time -           00:00:00.23
      Elapsed time -           00:00:03.61
      EXCP count    - 157
      Task  memory - 1213K (828K data, 385K program)
      Total memory - 2258K (1840K data, 418K program)
```

The third note reflects the totals for the procedure.

```
NOTE: The SQL procedure used the following resources:
      CPU      time -           00:00:00.25
      Elapsed time -           00:00:04.34
      EXCP count    - 186
      Task  memory - 1213K (828K data, 385K program)
      Total memory - 2258K (1840K data, 418K program)
```

Program Testing and Performance (Cont'd)

General Guidelines for Benchmarking Programs

- Never use elapsed time for comparison because it may be affected by concurrent tasks.
- Benchmark two programs in separate SAS sessions. If benchmarking is done within one SAS session, statistics for the second program can be misleading because the SAS supervisor might have loaded modules into memory from prior steps.
- Run each program multiple times and average the performance statistics.
- Use realistic data for tests. Program A could be better than program B on small tables and worse on large tables.

Question?