An Introduction to SQL in SAS

Pete Lund
Looking Glass Analytics
Olympia, WA

What is SQL?

- Is it “sequel” or “es-que-el”?
- “Structured Query Language”
  - not really structured in the way that other languages are
  - does much more than just queries
  - isn’t really a complete language
- First developed for IBM’s System/R project in the early 1970’s
- Implemented, to some degree, by all database management software

2

The Structure of an SQL Query

SELECT	< the variables you want > (columns)
FROM	< the datasets you want > (tables)
ON	< join conditions are met >
WHERE	< selection conditions are met >
GROUP BY	< summarize by these variables >
HAVING	< these summary conditions met >
ORDER BY	< sort by these variables >

LOOKINGGLASS ANALYTICS

3

Query Basics

- *SELECT* is a *statement* and is required
- All the rest are *clauses* of the *SELECT* statement
  - ***FROM***
  - ***ON***
  - ***WHERE***
  - ***GROUP BY***
  - ***HAVING***
  - ***ORDER BY***
- *FROM* is the only required clause
- The clauses are always ordered as above
- Each clause can appear, at most, once in a query

LOOKINGGLASS ANALYTICS

4

Choosing Your Columns

- The *SELECT* Statement -

- The first step in getting the data that you want is selecting the columns (variables). To do this, use the *SELECT* statement

```
select BookingDate,
       ReleaseDate,
       ReleaseCode
```

- List as many columns as needed, separated by commas

LOOKINGGLASS ANALYTICS

5

Choosing Your Table

- The *FROM* Clause -

- The next step in getting the data that you want is specifying the table (dataset). To do this use the *FROM* clause.

```
select BookingDate,
       ReleaseDate,
       ReleaseCode
from SASclass.Bookings
```

- These two components (*SELECT* and *FROM*) are all that's required for a valid query

LOOKINGGLASS ANALYTICS

6

Using SQL in SAS
- PROC SQL -

- To use SQL in SAS simply enclose the SQL statements inside PROC SQL

```
proc sql;
  select BookingDate,
         ReleaseDate,
         ReleaseCode
  from SASclass.Bookings;
quit;
```

The results will appear in the output window

Booking date	Release Date	Booking Release Code
09/30/2003	09/30/2003	BA
10/02/2003	10/03/2003	TC
10/04/2003	10/04/2003	BA
10/07/2003	10/07/2003	PR
10/09/2003	12/05/2003	CR
10/13/2003	10/14/2003	CR
10/17/2003	10/20/2003	TC
10/20/2003	10/21/2003	BA
10/23/2003	10/24/2003	CR
10/25/2003	10/26/2003	PR
10/28/2003	10/29/2003	PR
10/31/2003	11/03/2003	CR
12/29/2003	.	IN
11/09/2003	11/10/2003	CR

LOOKINGGLASS ANALYTICS

7

Things to Note about PROC SQL

- First, the entire query is a single statement: only one semi-colon
- Second, the procedure is terminated with QUIT rather than RUN: more than one query can be placed inside the procedure
- Finally, SQL statements cannot run outside PROC SQL

```
proc sql;
  select BookingDate,
         ReleaseDate,
         ReleaseCode
  from SASclass.Bookings;
quit;
```

LOOKINGGLASS ANALYTICS

8

What to Do with the Results?

- The *CREATE TABLE* Statement -

- By default, the results of a query are displayed in the output window
- You can create a SAS dataset from the query by preceding the *SELECT* statement with the *CREATE TABLE* statement

```
create table ReleaseCodes as
select BookingDate,
       ReleaseDate,
       ReleaseCode
from SASclass.Bookings;
```

The dataset ReleaseCodes will contain the three columns from the dataset SASclass.Bookings

Ordering Your Columns

- The order of the columns in the *SELECT* statement determines the order that they will appear in the output

```
select BookNum,
       Severity,
       InfractionDate
from SASclass.Infractions;
```

Notice that by default the variable labels are displayed in the output

SAS Explorer – Infractions table columns

BookNum	InfractionCount	INFRACTIONDATE	Severity	INFRACTIONTYPE
197007400	1	08/04/2001	G	OT
197007400	2	08/29/2001	G	OT
197007400	3	01/28/2002	S	OT
197016680	2	12/19/2001	G	OT
197016680	1	12/20/2001	G	OT
197016680	4	04/01/2002	G	OT

Output Window

Booking Number	Infraction Severity	Infraction Date
197007400	G	08/04/2001
197007400	G	08/29/2001
197007400	S	01/28/2002
197016680	G	12/19/2001
197016680	G	12/20/2001
197016680	G	04/01/2002
197016680	G	04/01/2002
197016680	S	05/05/2002

SELECT Statement Options

- On the *SELECT* statement you can rename, label and format variables
  - Rename: use the *AS* keyword
  - Label: use the *LABEL* keyword
  - Format: use the *FORMAT* keyword
  - Length: use the *LENGTH* keyword

```
create table ReleaseCodes as
select BookingDate as BD,
       ReleaseDate as RD format=monyy7.,
       ReleaseCode label='Rel Code'
from SASclass.Bookings;
```

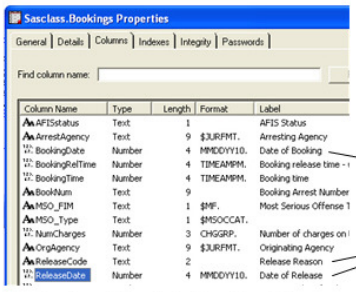
LOOKINGGLASS ANALYTICS

Ex 1

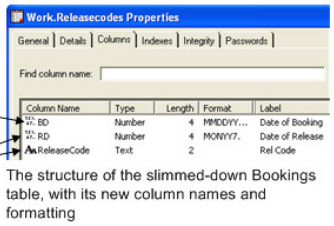
Changing Column Attributes

- The results of the query:

```
create table ReleaseCodes as
select BookingDate as BD,
       ReleaseDate as RD format=monyy7.,
       ReleaseCode label='Rel Code'
from SASclass.Bookings;
```



The structure of the original Bookings table.



The structure of the slimmed-down Bookings table, with its new column names and formatting

LOOKINGGLASS ANALYTICS

Selecting All Columns – A Shortcut

- All the columns in a table can be selected using an asterisk (*) rather than a column list

```
select *
from SASclass.Bookings;
```

- The above query would select all the columns in the Bookings table

Creating New Columns

- Another Use for *AS* -

- You can use any valid arithmetic or logical expression to create a new column

```
select BookingDate,
       ReleaseDate,
       ReleaseDate - BookingDate as LOS
from SASclass.Bookings;
```

- A new column, LOS, will be displayed in the output
  - It would be included in a new table if we'd had a *CREATE TABLE* statement

Creating New Columns

- The new column is added to the output
- Without the *AS,* the column header in output window would be blank
- If we'd created a new table, the new variable would be called _TEMA001 – the next unnamed column would be _TEMA002 and so on
- Lesson: *AS* is important!

Date of Booking	Date of Release	
01/01/2000	01/02/2000	1
01/01/2000	01/24/2000	23
01/01/2000	01/03/2000	2
01/02/2000	01/23/2000	21
01/03/2000	01/04/2000	1
01/03/2000	01/06/2000	3
01/03/2000	01/05/2000	2
01/03/2000	01/04/2000	1
01/05/2000	01/27/2000	22
01/06/2000	06/04/2000	150

LOOKINGGLASS ANALYTICS

15

Creating New Columns
- Using Formats -

- You can use *PUT* and *INPUT* functions to create new columns

```
select SentenceDate,
       put(SentenceFlag,YesNo.) as Sentenced
from SASclass.Charges;
```

- The new column, Sentenced, would now be character (length 3) with values of "Yes" and "No"

LOOKINGGLASS ANALYTICS

16

Conditional Logic in the *SELECT* Statement
- The *CASE* Operator -

- The *CASE* operator can be used in the *SELECT* statement to conditionally set a value to a new column

```
select *,
       case
         when InfractionType eq 'IS' then '*'
         when Severity eq 'S' then '*'
         else ' '
       end as CheckThese
from SASclass.Infractions;
```

- Similar to the datastep SELECT…WHEN

Logical Operators in *CASE*

- You can use logical as well as arithmetic operators with the *CASE* operator
- The *CASE* we saw before…

```
case
  when InfractionType eq 'IS' then '*'
  when Severity eq 'S' then '*'
  else ' '
end as CheckThese
```

- …could be rewritten as

```
case
  when InfractionType eq 'IS' or Severity eq 'S' then '*'
  else ' '
end as CheckThese
```

Sequence of *CASE* Expressions

- Like *IF…THEN…ELSE*, the value returned is from the first condition that is true

```
case
  when InfractionType eq 'IS' and Severity eq 'S' then 1
  when InfractionType eq 'IS' then 2
  when Severity eq 'S' then 3
  else 0
end as CheckThese
```

- In the above example, serious assaults on staff (IS/S) meets all three conditions – it is the first that sets the value

LOOKINGGLASS ANALYTICS

19

A Column Name with *CASE*

- If only one column is referenced in the CASE logic, it can be included with CASE rather than repeated in WHENs
- The following are equivalent

```
case
  when Race eq 'W' then 'White'
  when Race eq 'B' then 'Black'
  else 'Other'
end as RaceGroup
```

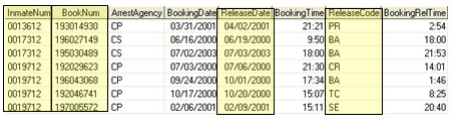
```
case Race
  when 'W' then 'White'
  when 'B' then 'Black'
  else 'Other'
end as RaceGroup
```

LOOKINGGLASS ANALYTICS

20

Choosing Your Rows
- The *WHERE* Clause -

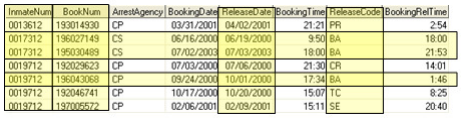
- The *SELECT* statement allows you to choose the columns you want



InmateNum	BookNum	ArrestAgency	BookingDate	ReleaseDate	BookingTime	ReleaseCode	BookingRelTime
0013612	193014930	CP	03/31/2001	04/02/2001	21:21	PR	2:54
0017312	196027149	CS	06/16/2000	06/19/2000	9:50	BA	18:00
0017312	195030489	CS	07/02/2003	07/03/2003	18:00	BA	21:53
0019712	192029623	CP	07/03/2000	07/06/2000	21:30	CR	14:01
0019712	196043068	CP	09/24/2000	10/01/2000	17:34	BA	1:46
0019712	192046741	CP	10/17/2000	10/20/2000	15:07	TC	8:25
0019712	197005572	CP	02/06/2001	02/09/2001	15:11	SE	20:40

```
select InmateNum, BookNum,
       ReleaseDate,ReleaseCode
```

- The *WHERE* clause allows you to select the rows that you want



InmateNum	BookNum	ArrestAgency	BookingDate	ReleaseDate	BookingTime	ReleaseCode	BookingRelTime
0013612	193014930	CP	03/31/2001	04/02/2001	21:21	PR	2:54
0017312	196027149	CS	06/16/2000	06/19/2000	9:50	BA	18:00
0017312	195030489	CS	07/02/2003	07/03/2003	18:00	BA	21:53
0019712	192029623	CP	07/03/2000	07/06/2000	21:30	CR	14:01
0019712	196043068	CP	09/24/2000	10/01/2000	17:34	BA	1:46
0019712	192046741	CP	10/17/2000	10/20/2000	15:07	TC	8:25
0019712	197005572	CP	02/06/2001	02/09/2001	15:11	SE	20:40

```
select InmateNum, BookNum,
       ReleaseDate,ReleaseCode
            :
            :
where ReleaseCode eq 'BA'
```


Choosing Your Rows
- The *WHERE* Clause -

- The *WHERE* clause follows the *FROM* clause and contains conditional logic that determines the rows that will be selected

```
select *
from SASclass.Infractions
where Severity eq 'S';
```

- The query above would select only rows that had a serious infraction

WHERE Clause Operators

- All of the SAS conditional and logical operators can be used in an SQL *WHERE* clause
  - Conditional

    -- EQ (=) -- NE (<>)
    -- GT (>) -- GE (>=)
    -- LT (<) -- LE (<=)
    -- IN -- NOT
  - -- Logical

    -- AND -- OR

Special *WHERE* Clause Operators

- There are a number of operators that are specific to the *WHERE* clause
  - IS MISSING
  - IS NULL
  - BETWEEN
  - CONTAINS
  - LIKE
  - =* (sounds like)

WHERE Clause Operators
- *IS NULL* and *IS MISSING* -

- Use the *IS NULL* or *IS MISSING* operator to return rows with missing values

```
create table NoSentence as
select *
from SASclass.Charges
where SentenceDate is null;
```

- The following WHERE clauses are equivalent to the above:

```
where SentenceDate is missing
where SentenceDate eq .
```

LOOKINGGLASS ANALYTICS

25

WHERE Clause Operators
- *BETWEEN* -

- The *BETWEEN* operator allows you to search for a value that is between two other values

```
select *
from SASclass.ADPmonths
where ADPmonth between '1jul2001'd and '30jun2002'd;
```

- Note: the end points are included in the results of the query

LOOKINGGLASS ANALYTICS

26

A *BETWEEN* Quirk

- The values referenced in the *BETWEEN* are treated as range end points and are automatically placed in the correct order
- The following two conditions produce the same results:

```
where ADPmonth between '1jul2001'd and '30jun2002'd

where ADPmonth between '30jun2002'd and '1jul2001'd
```


WHERE Clause Operators

- CONTAINS -

- You can search for a string inside another string with the *CONTAINS* operator

```
select *
from SASclass.Charges
where ChargeDesc contains 'THEFT';
```

- Charges like “THEFT2”, “AUTO THEFT” and “THEFT OF PROPERTY” would all be included in the result

CONTAINS is Case Sensitive

- Like other string comparisons in SAS, the *CONTAINS* operator is case-sensitive
- To make the previous query return rows with “Auto Theft”, add an *UPCASE* function to the *WHERE* clause

```
select *
from SASclass.Charges
where upcase(ChargeDesc) contains 'THEFT';
```


WHERE Clause Operators

- LIKE -

- You can do more detailed pattern matching with the *LIKE* operator using pattern matching characters
  - _ (underscore): matches any single character
  - % (percent sign): matches any number of characters (including zero)
  - other characters: matches that character

```
select *
from SASclass.Charges
where ChargeDesc like '%THEFT%';
```

Find "THEFT" with any number of characters before and/or after

Some Examples of *LIKE*

- The following WHERE clauses return…

where ChargeDesc like 'THEFT%';
(anything that begins with "THEFT")

THEFT
THEFT 2
THEFT-AUTO

where ChargeDesc like '%THEFT';
(anything that ends with "THEFT")

THEFT
AUTO THEFT
3RD DEGREE THEFT

where ChargeDesc like '%_THEFT';
(anything that has at least one character before and then ends with "THEFT")

AUTO THEFT
AUTO-THEFT
3RD DEGREE THEFT

LOOKINGGLASS ANALYTICS

31

The Big Difference Between _ and %

- Remember, _ looks for any single character and % looks for any number of characters (including 0)

where ChargeDesc like 'THEFT_FTA'

Charge Desc	Frequency	Percent	Cumulative Frequency	Cumulative Percent
THEFT FTA	3	11.11	3	11.11
THEFT-FTA	1	3.70	4	14.81
THEFT/FTA	23	85.19	27	100.00

where ChargeDesc like 'THEFT%FTA'

ChargeDesc	Frequency	Percent	Cumulative Frequency	Cumulative Percent
THEFT FTA	1	1.32	1	1.32
THEFT /FTA	2	2.63	3	3.95
THEFT 1 / FTA	1	1.32	4	5.26
THEFT 2 FTA	1	1.32	5	6.58
THEFT 2 /FTA	2	2.63	7	9.21
THEFT 2 FTA	1	1.32	8	10.53
THEFT 2/FTA	2	2.63	10	13.16
THEFT 3 FTA	1	1.32	11	14.47
THEFT 3 FTA	1	1.32	12	15.79
THEFT 3 FTA	5	6.58	17	22.37
THEFT 3 /FTA	5	6.58	22	28.95
THEFT 3 FTA	2	2.63	24	31.58
THEFT 3-FTA	1	1.32	25	32.89
THEFT 3/CASH/FTA	3	3.95	28	36.84
THEFT 3/FTA	21	27.63	49	64.47
THEFT FTA	3	3.95	52	68.42
THEFT-FTA	1	1.32	53	69.74
THEFT/FTA	23	30.26	76	100.00

LOOKINGGLASS ANALYTICS

32

WHERE Clause Operators

- =* (Sounds Like) -

- The “sounds like” operator, =*, uses the Soundex algorithm to match like character values

```
select LastName, FirstName
from SASclass.Inmates
where LastName =* 'SMITH';
```

- This query would return all rows where the inmates last name “sounds like” “Smith”

LOOKINGGLASS ANALYTICS

33

A Short History of Soundex

- Developed in the mid-1910’s by Odell and Russell to encode names
- Used extensively in the 1930’s by WPA crews to organize census data from 1880 to 1920
- Works best with American/English names
- Uses a very simple algorithm – remember, it was originally all done by hand!

LOOKINGGLASS ANALYTICS

34

The Soundex Algorithm

- Let's follow "SMITH" through the algorithm

SMITH

A. Retain the first letter in the argument and discard the following letters:
- A E H I O U W Y → **SMT**

B. Assign the following numbers to these classes of letters:
1: B F P V
2: C G J K Q S X Z
3: D T → **SM3**
4: L
5: M N → **S53**
6: R

C. If two or more adjacent letters have the same classification from Step B, then discard all but the first.

Sounds-Like Matches to "SMITH" (S53)

```
select LastName, FirstName
from SASclass.Inmates
where LastName =* 'SMITH';
```

The results of our original query:

Last Name	Frequency	Percent	Cumulative Frequency	Cumulative Percent
SCHMIDT	1	3.23	1	3.23
SCHMIT	1	3.23	2	6.45
SMITH	27	87.10	29	93.55
SMITHEE	1	3.23	30	96.77
SNEED	1	3.23	31	100.00

SCHMIDT
Step 1 - retain the first letter ("S") and eliminate the "H" and "I": SCMDT
Step 2 - eliminate the duplicate value letters (S/C, both 2, and D/T, both 3): SMD
Step 3 - "M" gets a value of 5 and "D" gets a value of 3
SCHMIDT="S53"

SNEED:
Step 1 - retain the first letter ("S") and eliminate the "E": SND
Step 3 - "N" gets a value of 5 and "D" gets a value of 3
SNEED = "S53"

One More Word About *WHERE*

- Columns in the *WHERE* clause do not need to be in the *SELECT*

```
select BookNum,
       BookingDate,
       ReleaseDate,
       ArrestAgency,
       MSO_Type
from SASclass.Bookings
where ReleaseCode eq 'BA';
```

- This query would return all bailed bookings, even though the release code is not in the select list

Dataset Options in the *FROM* Clause

- The *WHERE=* Option -

- SAS dataset options can be used on the "tables" in the *FROM* clause
- This includes the *WHERE=* dataset option

```
select *
from SASclass.Inmates
where Sex eq 'M'
```

is equivalent to

```
select *
from SASclass.Inmates(where=(Sex eq 'M'));
```

Why Use Dataset Options?

- Some SQL reserved words cannot be used as column names
- This query would fail because *CASE* is not a valid column name

```
create table WithCaseNumber as
select *
from ChargeData
where year(BookingDate) eq 2004 and
      Case ne '';
```

LOOKINGGLASS ANALYTICS

39

Using Dataset Options

- We could rewrite the earlier query in a couple different ways:

```
select *
from ChargeData(rename=(Case=CaseNumber))
where year(BookingDate) eq 2004 and
      CaseNumber ne '';
```

Rename the column and use the new name in the *WHERE* clause

```
select *
from ChargeData(where=(Case ne ''))
where year(BookingDate) eq 2004;
```

Use the *WHERE* dataset option to subset the data

Notice we can have both a dataset *WHERE* and a SQL *WHERE* in the query

LOOKINGGLASS ANALYTICS

40

Sorting Your Data
- The *ORDER BY* Clause -

- You can use the ORDER BY clause to sort the results of your query

```
select *
from SASclass.Charges
where ChargeType eq 'A'
order by WarrantType;
```

- This query would select all the assault charges and sort them by the type of warrant (felony, investigation, misdemeanor)

Alphabetic Sorting
- Functions in the *ORDER BY* Clause -

- You can take advantage of using functions in the *ORDER BY* clause to get a case-insensitive sort

```
select ChargeDesc
from MixedCase
order by upper(ChargeDesc);
```

- This query would return a result in alphabetical order, regardless of the case of the charge description

You Can't Do This with PROC SORT!

```
select ChargeDesc
from MixedCase
order by ChargeDesc;
```

	ChargeDesc
1	AGGRESSIVE BEGGING
2	AGGRESSIVE BEGGING
3	ALCOHOL BVG ON METRO
4	ALCOHOL TRANSIT F
5	ALCOHOLIC BEV IN
6	ANIMAL ON PR
7	ANTI-HAC
⋮	⋮
6016	WORK REL SUSPENSION
6017	achol in park cash
6018	aggressive begging
6019	alcohl in park
6020	alcohol restric area
6021	amended to prop dest

```
select ChargeDesc
from MixedCase
order by upper(ChargeDesc);
```

	ChargeDesc
1	achol in park cash
2	AGGRESSIVE BEGGING
3	aggressive begging
4	AGGRESSIVE BEGGING
5	alcohl in park
6	ALCOHOL BVG ON METRO
7	alcohol restric area
8	ALCOHOL TRANSIT PROP

LOOKINGGLASS ANALYTICS

Things to Note about the Alphabetic Sort

- The results are now in the desired order
- The original case of the values has not been changed
- There is a "randomness" to the groups of values in the result
- Officers don't spell very well ("achol", "alcohl")

	ChargeDesc
1	achol in park cash
2	AGGRESSIVE BEGGING
3	aggressive begging
4	AGGRESSIVE BEGGING
5	alcohl in park
6	ALCOHOL BVG ON METRO
7	alcohol restric area
8	ALCOHOL TRANSIT PROP

LOOKINGGLASS ANALYTICS

Changing the Sort Order
- The *DESC* Option -

- The default sort order is ascending – to get a descending sort, use the *DESC* option following the column name

```
select OrgAgency format=$Agency.,
       BookNum,
       WarrantType
from SASclass.Charges
where ChargeType eq 'A'
order by WarrantType desc,
         OrgAgency;
```

Note the differences with PROC SORT:
- the option is DESCENDING
- the option precedes the variable name

LOOKINGGLASS ANALYTICS

45

Summarizing Your Data
- Aggregate Functions -

- We've seen how SAS functions can be used in SQL queries to act on a single value in a table
- You can use aggregate functions to act on a group of columns or rows

- avg	- max	- prt	- sumwgt
- count	- mean	- range	- t
- css	- min	- std	- uss
- cv	- n	- stderr	- var
- freq	- nmiss	- sum	

LOOKINGGLASS ANALYTICS

Ex 1 Ex 2 Ex 3

46

Summary Functions Across Columns

- If more than one column is referenced in a function, it works across columns on each row

```
select *,
       sum(GoodTime,CreditDays) as OffDays
from SASClass.Charges;
```

	BookNum	ChargeNum	Bail	ChargeDesc	FIM	ChargeType	GoodTime	CreditDays
1	192000028	1	1050	RECKLESS DRIVING $$$	MB	T	0	0
2	192000096	1	.	VUCSA/ SODA	FR	D	0	0
3	192000096	2	.	ATTEMT/VUCSA	MR	D	0	0
4	192000146	1	550	D.W.I. FTA	MR	U	0	0
5	192000168	1	500	DRIVING DURING SUSPE	MD	T	0	0
6	192000168	2	500	HAB TRAFFIC OFFENDER	MD	T	0	0
7	192000168	3	.	ATTEMP VUCSA	MC	D	180	160
8	192000168	4	.	PROB HOLD	FC	N	30	10
9	192000168	5	.	PROB HOLD	FC	N	30	10
10	192000352	1	150	SIMPLE ASSAULT	MD	A	0	0
11	192000352	2	2000	ASSAULT	MB	A	0	0
12	192000352	3	300	DR LIC SUS/REV/NOINS	MB	T	0	0
13	192000402	1	2050	D.W.I./HTO	MB	U	0	0

Σ across columns

GoodTime	CreditDays	OffDays
0	0	0
180	160	340
30	10	40
30	10	40
0	0	0
0	0	0

Summarization Across Rows

- If a single column is referenced in a function, it works across all rows in the table

```
select sum(Bail) as TotalBail,
       mean(Bail) as MeanBail,
       max(Bail) as MaxBail,
       nmiss(Bail) as NoBailSet
from SASclass.Charges;
```

Σ down rows

VIEWTABLE: Work.Bailsummary

	TotalBail	MeanBail	MaxBail	NoBailSet
1	31197597	4490.8013531	1000000	5040

Summarizing Your Data
- The *GROUP BY* Clause -

- By default, summary functions work across a whole table – you can summarize groups of data with the *GROUP BY* clause

```
select BookNum,
       sum(Bail) as TotalBail format=dollar15.
from SASclass.Charges
group by BookNum;
```

- We will now have one row per value per booking with total bail for al charges on that booking

Using the *GROUP BY* Clause

```
select BookNum,
       sum(Bail) as TotalBail format=dollar15.
from SASclass.Charges
group by BookNum;
```

Original table

	BookNum	ChargeNum	Bail
1	192000028	1	1050
2	192000096	1	.
3	192000096	2	.
4	192000146	1	550
5	192000168	1	500
6	192000168	2	500
7	192000168	3	.
8	192000168	4	.
9	192000168	5	.
10	192000352	1	150
11	192000352	2	2000
12	192000352	3	300
13	192000402	1	2050

Summarized table

	BookNum	TotalBail
1	192000028	$1,050
2	192000096	.
3	192000146	$550
4	192000168	$1,000
5	192000352	$2,450
6	192000402	$2,050
7	192000422	$500
8	192000427	$750

We now have one row per booking with the total bail summarized

The Special Case of *COUNT()*

- Most summary functions take numeric column(s) as arguments
- An exception is *COUNT* which can take any of three types of arguments
  - Numeric column
  - Character column
  - *
    - Note: the *FREQ* and *N* functions are the same as COUNT
- Note: N, *NMISS, MIN and MAX* can also take a character column as an argument

LOOKINGGLASS ANALYTICS

Ex 1 51

Using *COUNT()*

- If the argument is a column name, the function returns the number of non-missing values
- If the argument is "*", the function returns a count all the rows in the table

```
select count(GoodTime) as GoodTime_Count format=comma7.,
       count(ChargeDesc) as ChargeDesc_Count format=comma7.,
       count(SentenceDate) as SentenceDate_Count format=comma7.
       count(*) as TotalRow_Count format=comma7.
from SASclass.Charges;
```

GoodTime_ Count	ChargeDesc_ Count	Sentence Date_Count	TotalRow_ Count
11,987	11,987	2,750	11,987

LOOKINGGLASS ANALYTICS

52

Mixing the "Scope" of *COUNT()*

- You can use both column-specific and row-count *COUNT* functions in an expression

```
select ChargeType,
       count(ChargeDesc) as ChargeDesc_Count,
       count(ChargeDesc) / count(*) as ChargeDesc_Pct,
       count(SentenceDate) as SentenceDate_Count,
       calculated SentenceDate_Count / count(*) as SentenceDate_Pct
from SASclass.Charges
group by ChargeType;
```

Charge Type	ChargeDesc_ Count	ChargeDesc_ Pct	Sentence Date_Count	Sentence Date_Pct
Assault	941	100.00%	203	21.57%
Prostitution	230	100.00%	90	39.13%
Drug	1,355	100.00%	147	10.85%
Homicide	16	100.00%	0	0.00%
Non-Compliance	1,425	100.00%	548	38.46%
Other	1,826	100.00%	258	14.13%
Property	2,045	100.00%	494	24.16%
Traffic (non-alcohol)	2,515	100.00%	635	25.25%
DUI	851	100.00%	251	29.49%
Domestic Violence	783	100.00%	124	15.84%

LOOKINGGLASS ANALYTICS

53

Summarizing Your Data

- The *DISTINCT* Operator -

- You can get a list or count of unique values for a column using the *DISTINCT* operator

```
select distinct Facility
from SASclass.Bookings;
```

Facility
4
5
H
M
P
R
W

```
select count(distinct Facility)
from SASclass.Bookings;
```

Facility Count
7

- The first query would return one row for each value of facility, the second a single value containing a count of the unique values of facility

LOOKINGGLASS ANALYTICS

54

DISTINCT with Formatted Values

- You can use formatted columns with the *DISTINCT* operator

```
select distinct Race format=$Race.,
                Sex format=$Gender.
from SASclass.Inmates;
```

Race	Gender
Asian	Female
Asian	Male
Black	Female
Black	Male
Native American	Female
Native American	Male
Other/Unknown	Female
Other/Unknown	Male
White	Female
White	Male

Note: when using multiple columns with DISTINCT only the actual value combinations are returned - not all possible combinations.

A Caution with *DISTINCT* and Formats

- There is a danger when using formatted values and *DISTINCT*. The distinct value list is built on the actual, unformatted data and then the formats are applied when the data is presented.
- If there are multiple raw values assigned to the same formatted value your distinct list may not look so distinct.

```
select distinct Facility format=$Secure.
from SASclass.Bookings;
```

Facility
Secure
Secure
Alternative
Secure
Secure
Secure
Alternative

We still have the seven rows that the original *DISTINCT* returned earlier

Referencing Created Columns
- The *CALCULATED* Option -

- You can reference columns that were created in the *SELECT* statement with the *CALCULATED* option

```
create table OffDays as
select *,
       sum(GoodTime,CreditDays) as OffDays
from SASClass.Charges
where calculated OffDays gt 0;
```


Why Use *CALCULATED*?

- The *WHERE* and *GROUP BY* clauses both act on rows as they are coming into the query and so need to be present in the incoming table
- We can either repeat the expression or use the *CALCULATED* keyword

```
create table OffDays as
select *,
       sum(GoodTime,CreditDays) as OffDays
from SASClass.Charges
where sum(GoodTime,CreditDays) gt 0;
```

These two queries are equivalent

```
create table OffDays as
select *,
       sum(GoodTime,CreditDays) as OffDays
from SASClass.Charges
where calculated OffDays gt 0;
```

CALCULATED in the SELECT Statement

- You can use *CALCULATED* in the *SELECT* statement to reference a new column created in the *SELECT*

```
select *,
       sum(GoodTime,CreditDays) as OffDays,
       case
         when calculated OffDays gt 0 then '*'
         else ' '
       end as OffDayFlag
from SASClass.Charges;
```

Note: the new column must be created before it is referenced by CALCULATED

A *GROUP BY* Shortcut

- Relative Column Referencing -

- Instead of using column names in the *GROUP BY* clause you can use the column position

```
select WarrantType,
       ChargeType,
       mean(Bail) as TotalBail format=dollar15.
from SASclass.Charges
group by 2,1;
```

- The above *GROUP BY* is equivalent to

```
group by ChargeType, WarrantType;
```

Ex 1

GROUP BY and *ORDER BY*

- Using Them Together -

- *GROUP BY* has an implied sort and results are displayed in that order
- You can use *ORDER BY* in the same query to override the *GROUP BY* sorted order

```
select FIM,
       mean(Bail) as MeanBail
from SASclass.Charges
group by FIM
order by MeanBail desc;
```

```
group by FIM;
```

FIM	MeanBail
F	$13,152
I	$18,890
M	$1,066

GROUP BY only - sorted by FIM

```
group by FIM
order by MeanBail desc;
```

FIM	MeanBail
I	$18,890
F	$13,152
M	$1,066

GROUP BY sort is overridden by *ORDER BY* - sorted by mean bail (descending)

Ex 1

Incomplete *GROUP BY* Clauses

- If the *GROUP BY* clause does not reference all the non-summary columns in the query the result is added to all the rows

Warrant Type	TotalBail
FB	$11,957
FC	.
FN	$10,750
FO	$4,974
FR	$16,233

	FIM	ChargeType	TotalBail
1	FB	D	$11,957
2	FB	N	$11,957
3	FB	N	$11,957
4	FB	N	$11
5	FB	P	
6	FB	D	
953	FC	O	.
954	FC	O	.
955	FC	D	.
956	FC	N	.
957	FN	D	$10,750
958	FN	A	$10,750
959	FN	V	$10,750
960	FN	D	$10,750
961	FN	P	$10,750
962	FN	P	$10,750
963	FO	D	$4,974
964	FO	N	$4,974
965	FO	D	$4,974

```
select FIM,
       ChargeType,
       mean(Bail) as TotalBail
from SASclass.Charges
group by FIM;
```

Selecting Summary Rows
- The *HAVING* Clause -

- To select rows based on the results of a summary function, use the *HAVING* clause

```
select BookingMonth,
       sum(NumCharges) as TotalCharges
from SASclass.Bookings
group by BookingMonth
having TotalCharges gt 275;
```

- *HAVING* works on rows as they "leave" the query – no *CALCULATED* needed

Creating Macro Variables in SQL
- The *INTO* Operator -

- The *SELECT* statement can produce macro variables with the *INTO* operator

```
select mean(Bail) into :MeanBail
from SASclass.Charges;
```

- Use a colon (:) to prefix the macro variable name
- The query above would create a macro variable &MeanBail containing the mean bail amount

Macro Variables with More than One Value
- The *SEPARATED BY* Option -

- Macro variables can be created from queries that return multiple rows

```
select distinct race into :RaceList
  separated by ','
from SASclass.Inmates;
```

- Each value of Race returned by the query will be placed in the macro variable &RaceList – each separated by a comma

The Structure of an SQL Query

SELECT	< the variables you want > (columns)
FROM	< the datasets you want > (tables)
ON	< join conditions are met >
WHERE	< selection conditions are met >
GROUP BY	< summarize by these variables >
HAVING	< these summary conditions met >
ORDER BY	< sort by these variables >

Contact Information

Pete Lund

Looking Glass Analytics

Olympia, WA

pete.lund@lgan.com

(360) 528-8970

LOOKINGGLASS ANALYTICS

67