SAS® and Relational Databases: What You Must Know

Author: Patricia Hettinger, Data Analyst – Consultant
Oakbrook Terrace, IL

ABSTRACT:

It is a rare shop that has data in just SAS® format. DB2, Teradata, Oracle, SQL Server - all of these relational databases have their own quirks that may not become apparent until your DBA calls. Just because you can assign a libname to a database and use proc sql doesn't mean it will run the same way as it would against a SAS source. This paper will address what you must know before using SAS®/Access for more efficient queries and consistent results.

INTRODUCTION:

The audience for this paper are those who have some experience with native SAS but limited experience with RDBMS databases and SAS/Access. We will address the usual design differences between SAS datasets and RDBMS tables, coding considerations, the libname option versus SQL Pass-Through and some trouble-shooting hints. Although meant to be a generic discussion of RDBMS, there will be some database-specific examples which will be noted.

TERMS USED IN THIS PAPER:

DBMS – General abbreviation for database management system. Includes hierarchical, relational, distributed and dimensional models.

RDBMS – relational database management system. A database system based upon the relational model introduced by E.F. Codd. Data are stored in tables as well as the relationships among the data.

DBA – database administrator. Someone with the authority to create, delete and update database objects like tables. Also sets up and enforces security rules.

SQL – Structured Query Language. The most widely used language for relational databases. RDBMS usually have extensions to the ANSI standard. SQL allows for queries, updates, structure creation and access determination.

ANSI - American National Standards Institute

SAS/Access – generic name for the SAS interface to a RDBMS. SAS has interfaces to several databases including Teradata, Oracle, DB2, SQL Server and PC files like Excel and Access.

Dataset – used here to refer to data in SAS format that can be directly used in SAS data steps and procedures.

Table – used here to refer to the basic data structure in an RDBMS.

ETL – extract, transform and load. The process by which data becomes stored in datasets and tables

Entity – a thing capable of a separate existence and can be uniquely identified. **Customer** is an example of an entity.

Primary key (PK) – a column or combination of columns that uniquely identifies an entity. For example, we may assign a numeric **Customer Id** to each customer.

Referential integrity – the notion that certain entities have mandatory relationships. For example, you wouldn't have an order without a customer.

Foreign key – an identifier in an entity that corresponds with the primary key of another entity. For example, **Customer Id** would be a foreign key in the **Order** table that would identify to whom the order belongs.

Entity-relationship diagram – a graphical rendering of the relationship between entities. The diagrams used in this paper use crows-feet notation.

DESIGN DIFFERENCES BETWEEN DATASETS AND TABLES

Datasets are known for the lack of methodology in creating and documenting them. This is partially due to the ease of creation. If you have enough disk space and write access to that space, you can create a SAS dataset. You can change the structure anytime you want. You might discover a lot of unhappy people whom you were unaware were using your dataset but you can do it.

Not so in an RDBMS. Most RDBMS of any size have a database administrator who controls what goes into the database, the access people have and the procedure to add new structures and elements. The DBA has several tools to monitor the system and see who is accessing what in detail. A more refined shop would have graphical data models so that users can see the relationships at a glance, source to target documents to detail the ETL involved, data element definitions and a central repository to store it all.

The terminology is different for a dataset versus a table also. Datasets have variables and observations. Tables have columns and rows. You will hear terms like 'primary key', 'foreign key', 'referential integrity' and 'index' much more with tables than with datasets.

The structure of the data you would query will probably be drastically different as well. It's not uncommon for datasets to have the information you need in just one or two sources. This is unlikely with an RBMS – you will probably have to connect many tables to get the same information.

For example, let's take San Vicente Center Point. SVCP is a charitable organization offering financial assistance like grocery or gas cards to those in need. The system was originally developed in SAS with data created by reading in Excel spreadsheets or text documents. There was even a rudimentary interface in SAS® SCL (Screen Control Lanugage) for adding new clients on the fly. Customer information, assistance information and San Vicente case worker (Visitor) were all in one dataset. Each client got his/her own record. Up to three assistance requests per client were stored, along with the SVCP visitor who handled the request.

There were several reasons this was redesigned. The major reason was the present economic conditions had many people making more than three requests. More people asked for assistance as well, increasing the caseload for each visitor. Fortunately, SVCP has able to recruit more volunteers. However that caused more visitor information to be entered as well. Even worse, it was very easy to make mistakes with this information, causing the same visitor to be listed more than once in the reports. To illustrate, Gene Marcus was listed as Eugene Marcus IV, Gene Marcus IV and Gene Marcus.

Another reason for the rewrite was that the SVCP became part of a greater charitable group network. They wanted an easier way to share their information, perhaps going to a web-enabled system. It is much easier to do this with an RBMS than SAS. Not too many people are fluent in SCL anymore.

We decided to use codes for the request, outcome and assistance column to give more consistent values. A later model will include family and income information as well. Our query discussion will be limited to information stored in the original SAS dataset as shown in figure 1

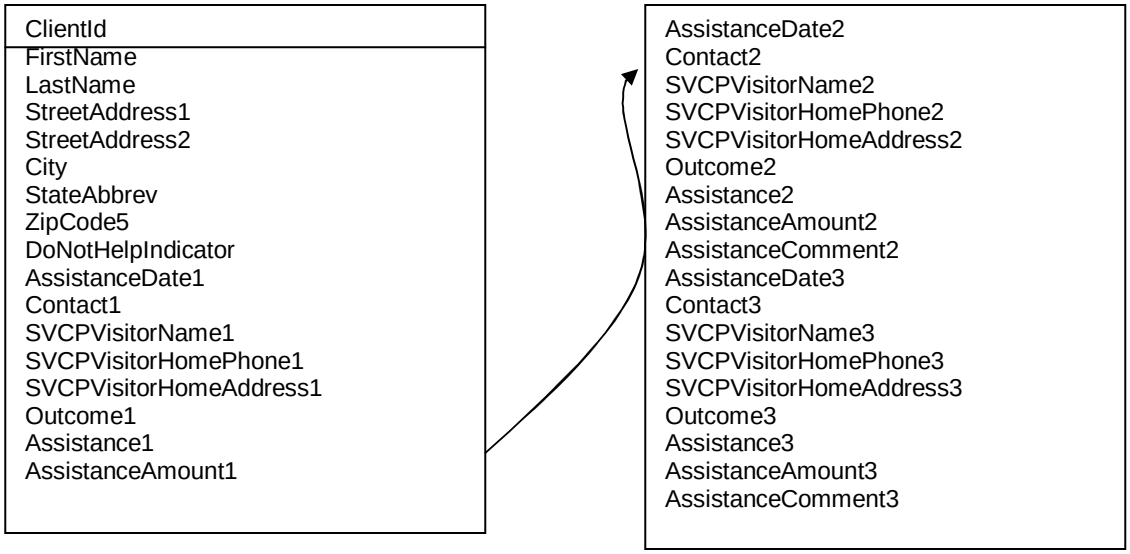
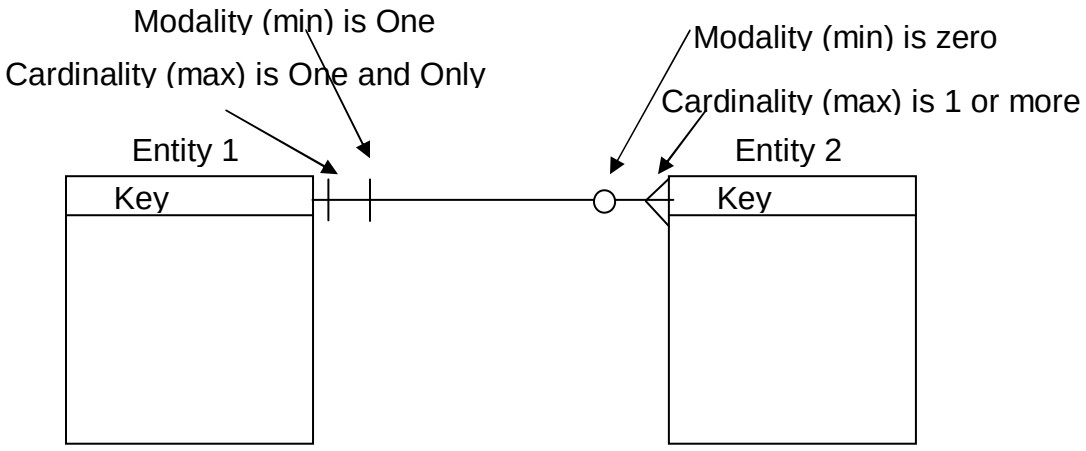


Figure 1: Original SVCP Structure

CROWS FOOT NOTATION

Cardinality is the maximum number of entities that can exist in a relationship. Modality is the minimum number. Figure 2 gives a typical representation of a parent/child relationship:



Entity 1 must exist for Entity 2 to exist
Entity 2 does not need to exist for Entity 1 to exist
More than one Entity 2 may exist for each Entity 1

Figure 2: Parent/child relationship representation

After data modeling, we ended up with the entity-relationship model in Figure 2. Note that we now need to put together six different sources. At least we shouldn't have to repeat any variables:

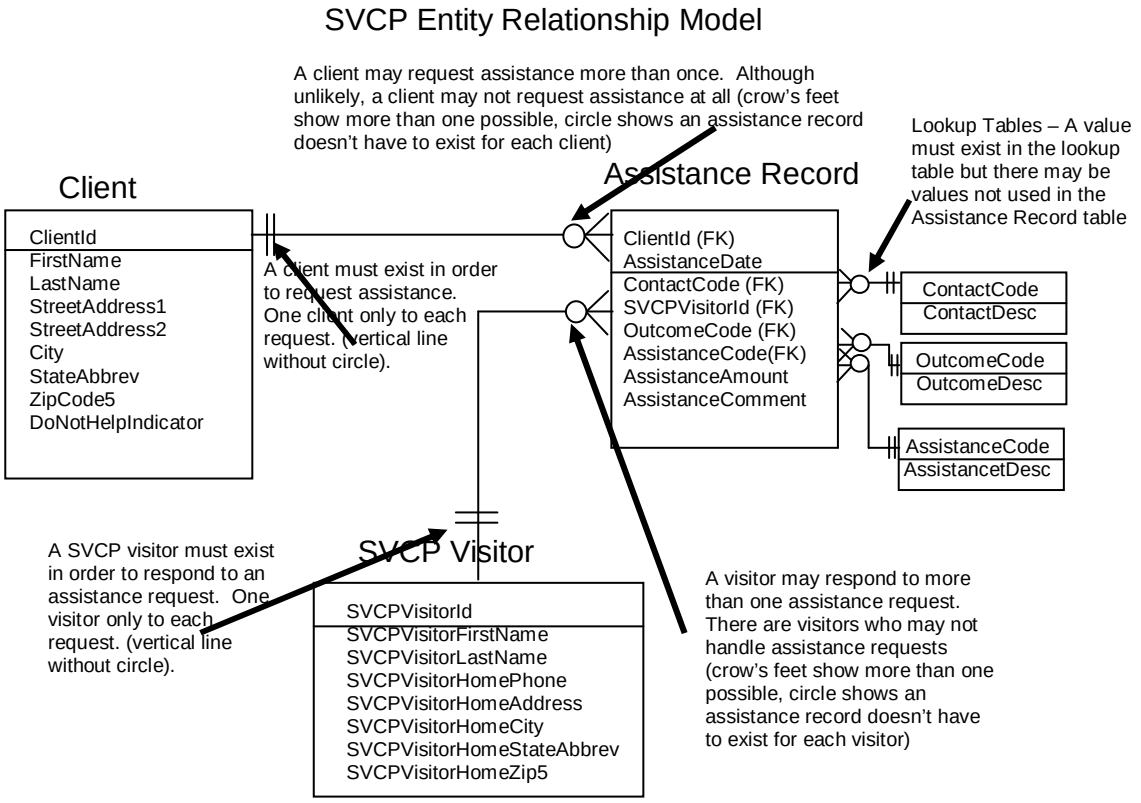


Figure 3

Something you may not be used to is the use of indexes. They comprise a significant part of this system. The original dataset didn't have any, not an uncommon situation in the SAS world. This model has nine currently and well may add more for performance reasons. An index may be used to define the primary key, enforce referential integrity or make retrieval of a subset easier.

The indexes are:

ClientId on Client – primary key
SVCPVisitorId on SVCP Visitor – primary key
ClientID, AssistanceDate on Assistance Record – primary key
ContactCode – primary key on lookup table
OutcomeCode – primary key on lookup table
AssistanceCode – primary key on lookup table
SVCPVisitorId on Assistance Record corresponding to SVCPVisitorId on SVCP Visitor – foreign key
ContactCode corresponding to ContactCode in lookup table – foreign key
OutcomeCode corresponding to OutcomeCode in lookup table – foreign key
AssistanceCode corresponding to AssistanceCode in lookup table – foreign key

IMPACT TO SAS CODE

There are two basic methods to access a database through SAS/Access. One with SQL Pass-Through through which SAS passes the SQL code directly to the database and the newer libname method which assigns a library to a database as if it were a SAS library. All commands are then translated into SQL with varying degrees of success. One advantage of using the libname method is that it lets you use the powerful SAS® Enterprise Guide interface. It also lets you query combinations of tables and datasets with proc sql although not without issues as we will see later.

In the case of our SVCP data, we can assign a library to the RDBMS like this:

libname SVCPdata sqlsvr user=&userid password=&passwd;

You can add any of the tables to your SAS EG project. Just turn off the option to automatically open them when adding them to avoid hanging up your session. You can add these tables much like you can any SAS library:

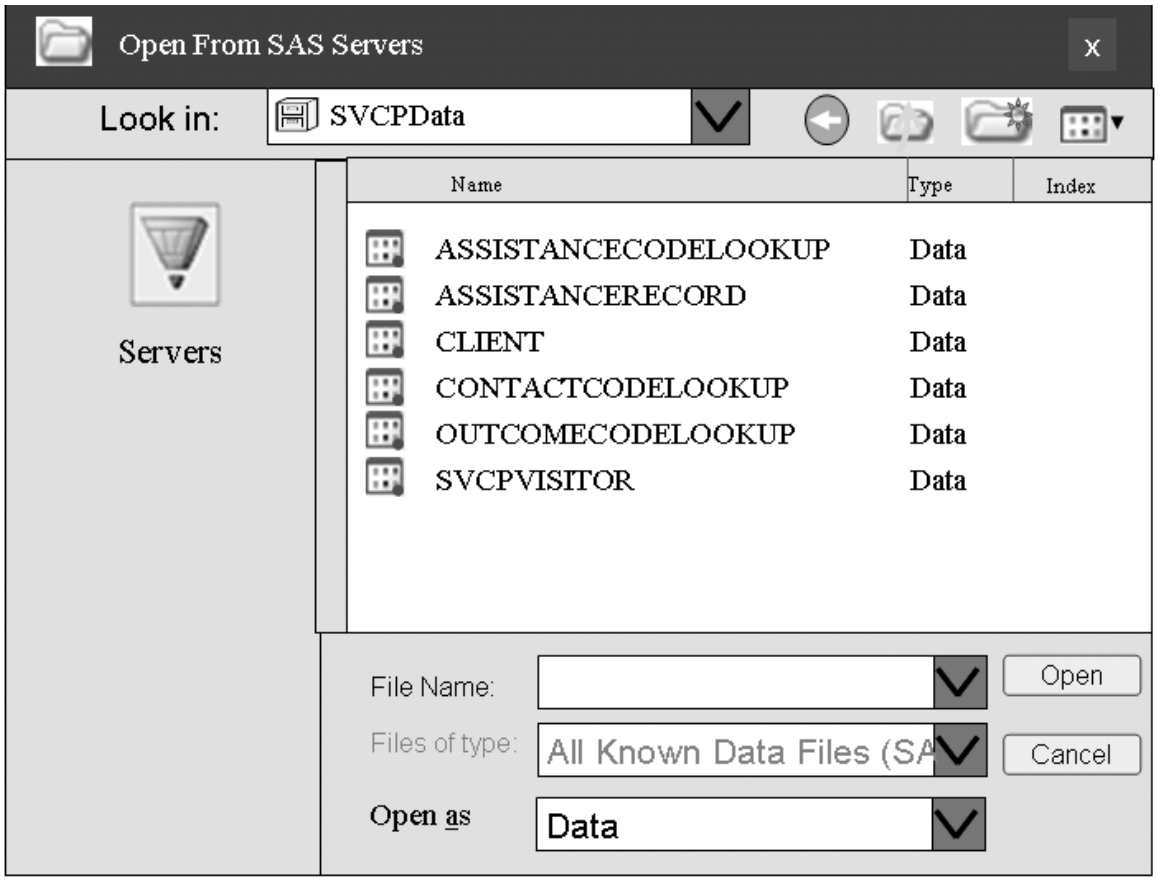


Figure 3: Visualizing SVCP on SQL Server

You may select one or more tables using the Query Builder as shown in figure 4:

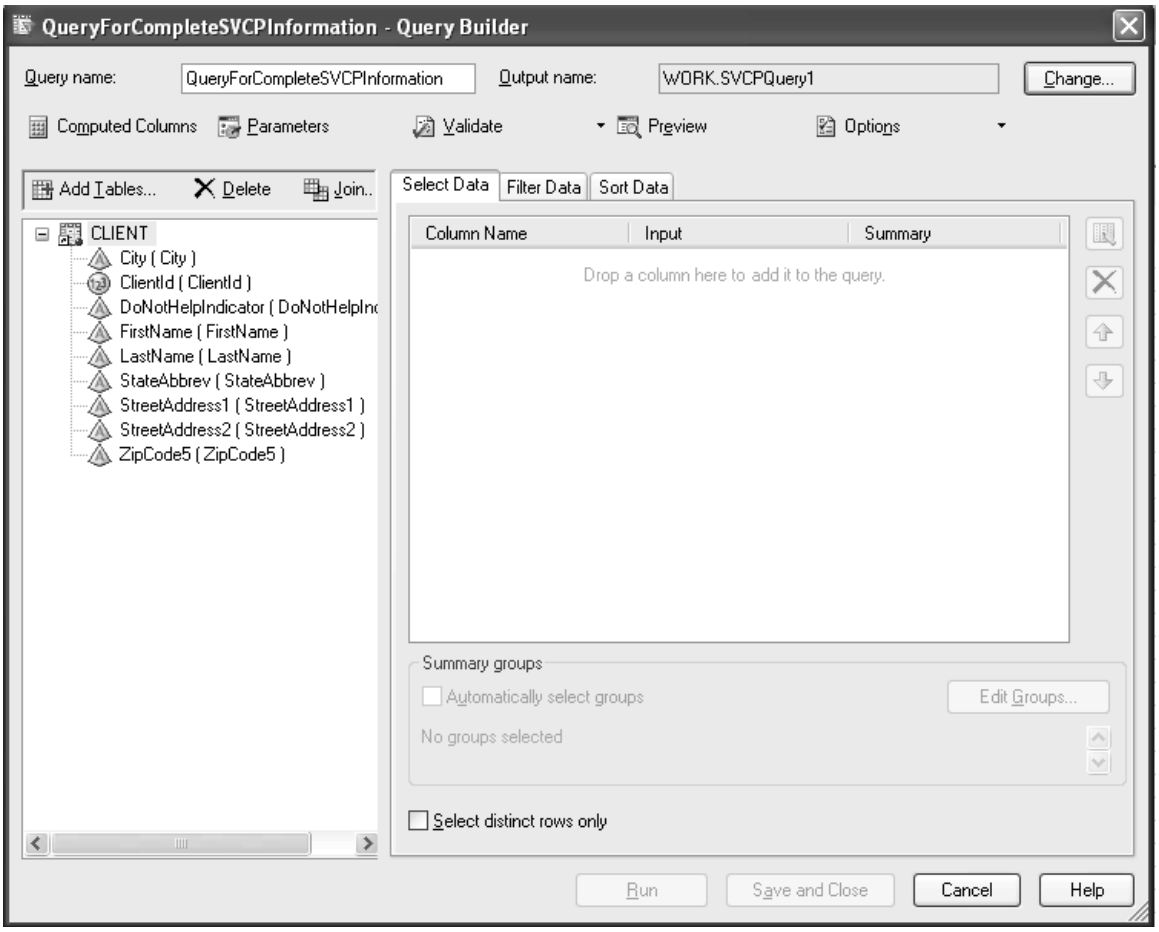


Figure 4 – selecting a table using SAS® Enterprise Guide Query Builder

You may the tables by clicking the 'Join' icon. When the join screen comes up, add a table by clicking the 'Add Table' icon. Note the default is an inner join on like-named fields:

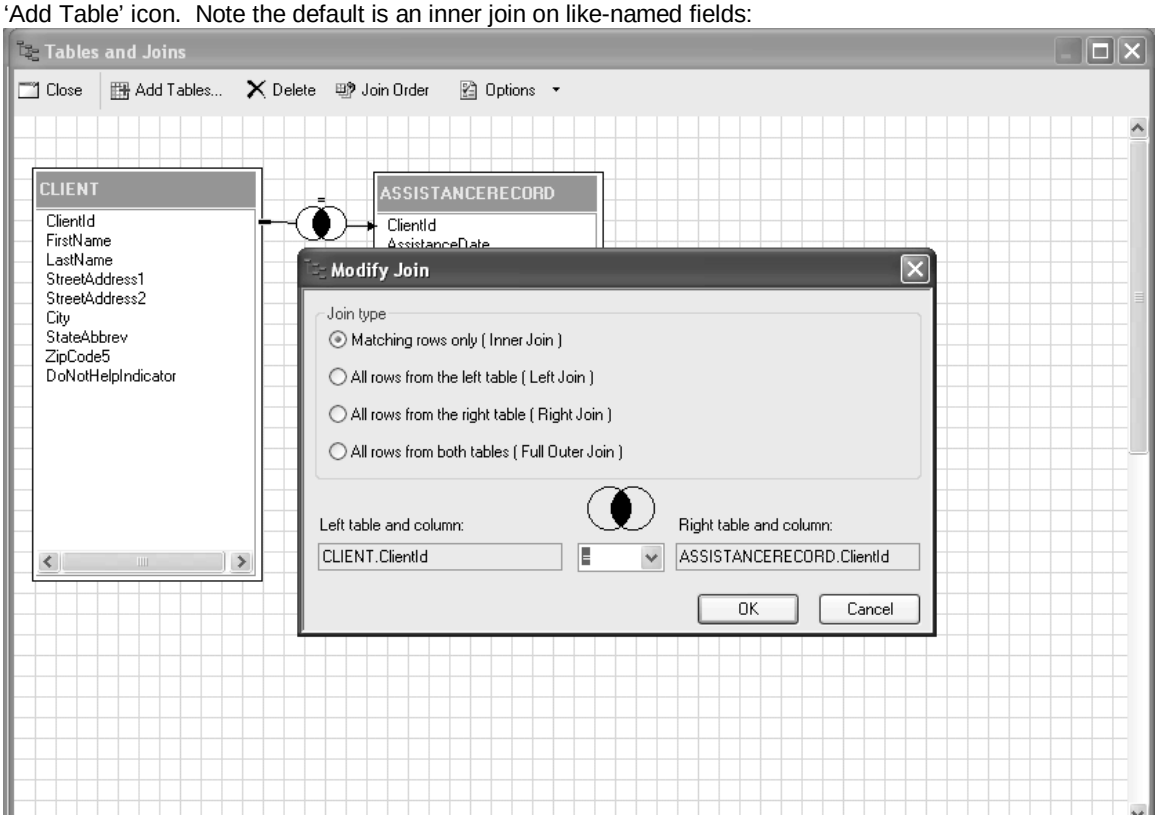


Figure 5: Joining tables in SAS® Enterprise Guide

Adding all of the tables gives us the diagram in Figure 6:

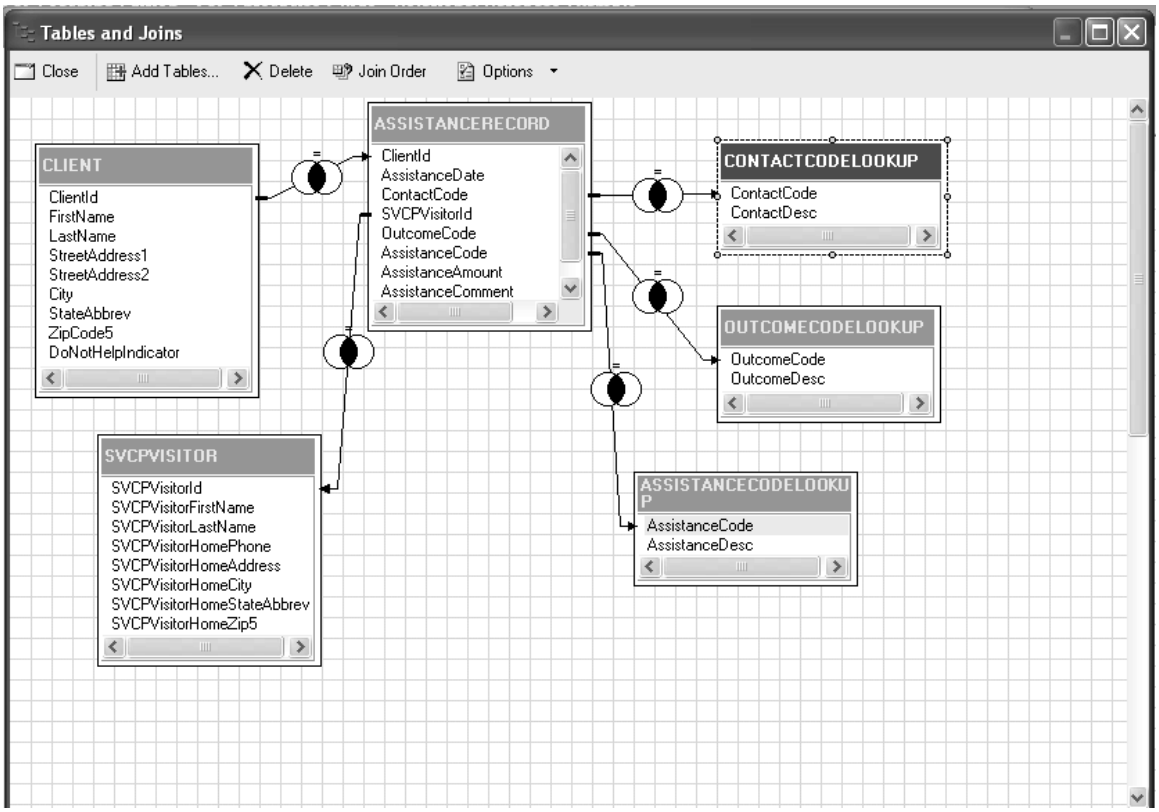


Figure 6: All tables added to join

Closing out of this screen brings up back to the main Query Builder form where we can select the columns we want:

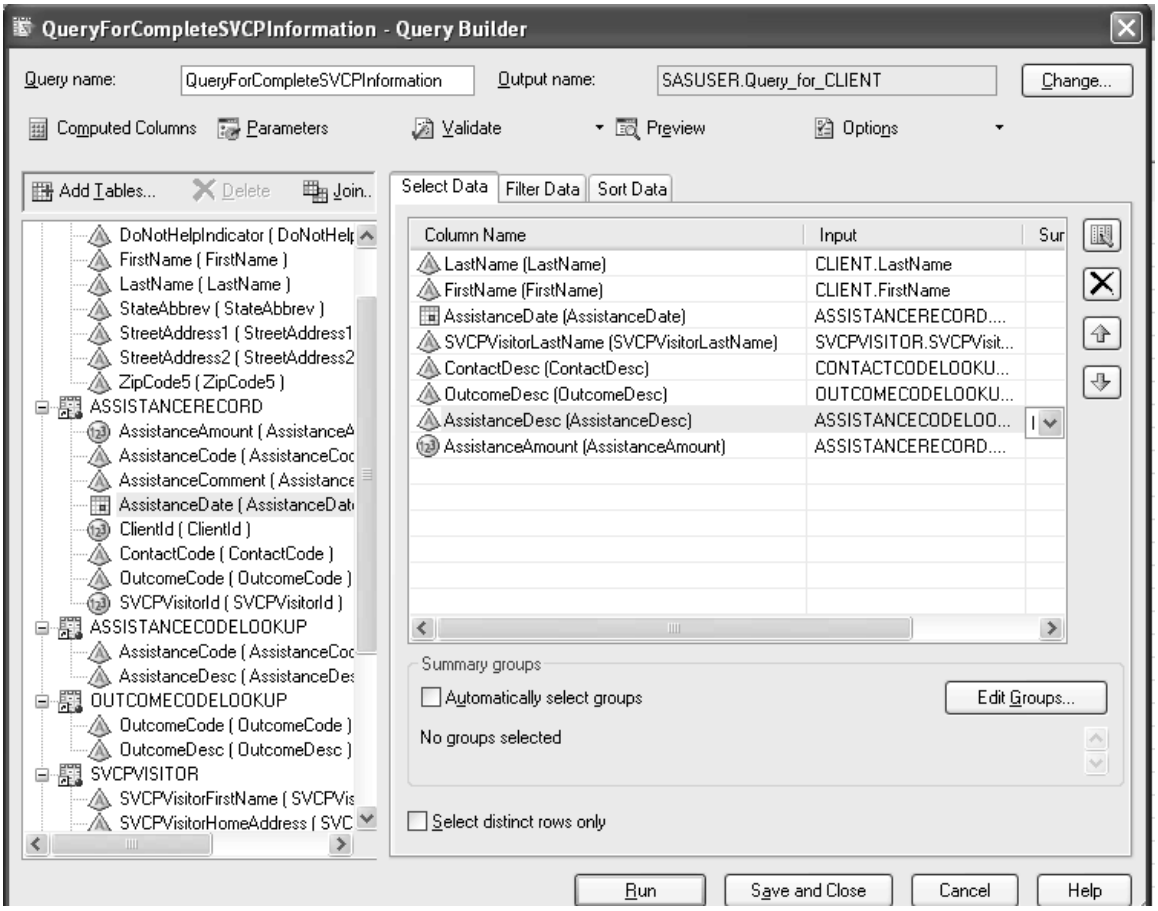


Figure 7: Selection of Columns

Let's sort by client last name and assistance date:

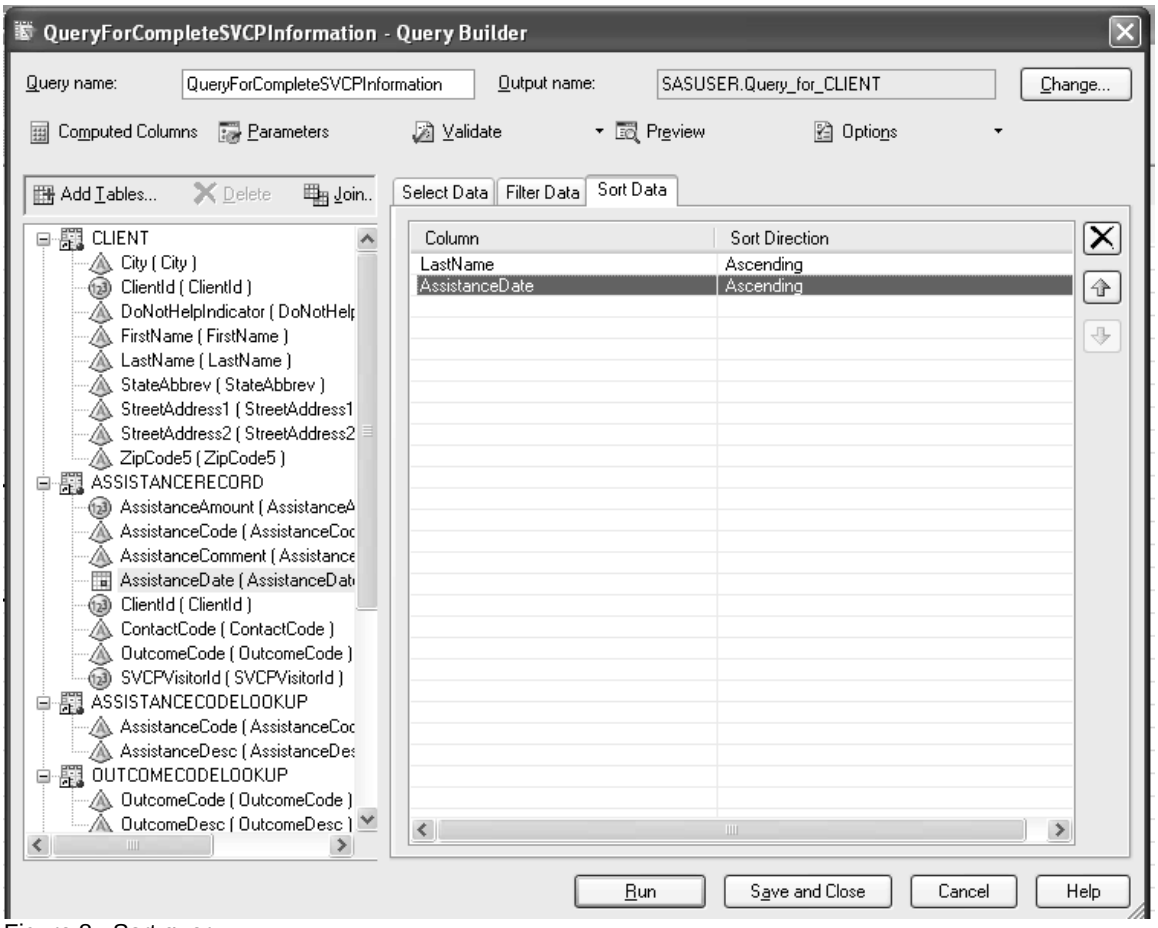


Figure 8: Sort query

The code generated looks like this:

```
PROC SQL;
 CREATE TABLE SASUSER.Query_for_CLIENT AS SELECT CLIENT.LastName,
        CLIENT.FirstName,
        ASSISTANCERECORD.AssistanceDate FORMAT=DATEAMPM22.,
        SVCPVISITOR.SVCPVisitorLastName,
        CONTACTCODELOOKUP.ContactDesc,
        OUTCOMECODELOOKUP.OutcomeDesc,
        ASSISTANCECODELOOKUP.AssistanceDesc,
        ASSISTANCERECORD.AssistanceAmount
 FROM SVCPDATA.CLIENT AS CLIENT
         INNER JOIN SVCPDATA.ASSISTANCERECORD AS ASSISTANCERECORD ON
(CLIENT.ClientId = ASSISTANCERECORD.ClientId)
         INNER JOIN SVCPDATA.ASSISTANCECODELOOKUP AS ASSISTANCECODELOOKUP ON
(ASSISTANCERECORD.AssistanceCode = ASSISTANCECODELOOKUP.AssistanceCode)
         INNER JOIN SVCPDATA.OUTCOMECODELOOKUP AS OUTCOMECODELOOKUP ON
(ASSISTANCERECORD.OutcomeCode = OUTCOMECODELOOKUP.OutcomeCode)
         INNER JOIN SVCPDATA.SVCPVISITOR AS SVCPVISITOR ON
(ASSISTANCERECORD.SVCPVisitorId = SVCPVISITOR.SVCPVisitorId)
         INNER JOIN SVCPDATA.CONTACTCODELOOKUP AS CONTACTCODELOOKUP ON
(ASSISTANCERECORD.ContactCode = CONTACTCODELOOKUP.ContactCode)
 ORDER BY CLIENT.LastName, ASSISTANCERECORD.AssistanceDate;
```

Or if you are fluent in SQL, you could just write similar code and submit it from anywhere, even a batch job on Unix. One thing to keep in mind is that embedded spaces and special characters are more common in some data systems like Access or SAP. If you are writing code referencing fields like that, you prefix the name with a single quote and end with a single quote n. BI0/FICATABLE would be 'BI0/FICATAB LE'n

One difference in querying RDBMS tables from SAS datasets is how duplicate field names are handled. SAS will store the fully qualified name from the tables in the case where columns have the same name. For example, the results of this query would be different depending upon whether customer and org were datasets or tables. It would also depend on upon your installation but you may see results like this:

```
Create table org_sale as
Select customer.name, org.name
From customer
Inner join org
On customer.custid=org.empid
;
```

Both customer and org as tables result set:

```
Customer.name       org.name
John Henderson      HungerDunger, HungerDunger, HungerDunger and McCormack
```

Either customer or org as a dataset result set (only one has to be a dataset for these results):

```
Name
John Henderson
```

In the latter case, you would see a message in the log saying the variable 'name' already exists. When that happens, the original value is the one stored, in this case, the customer name and the second one, organization name from org, dropped.

Now if you joined these by doing a merge, the last value would be the one stored:

```
Data org_sale;
Merge customer(in=a rename=(custid=empid))
Org(in=b); By empid;
If a and b;
```

Merge result set (only one has to be a dataset for these results):

```
Name
HungerDunger, HungerDunger, HungerDunger and McCormack
```

PERFORMANCE CONSIDERATIONS

One very common scenario is to 'join' a native SAS dataset with one or more database tables. If your SAS dataset is fairly large, this would result in full table scans of the database tables, then joining the results to the SAS dataset as the following Oracle SAS trace indicates. This code does a validation of certain fields in a SAS dataset compared to what should be their source in a corresponding Oracle table. We turn on the trace with this statement:

```
OPTIONS FULLSTIMER NOCENTER SAStrace='d,,d' SAStraceloc=SASlog NOSTSUFFIX;
```

The 'd..d' SAStrace setting will send all DBMS calls such as API and client calls, connection information and the row processing you see below to the log. Setting the SAStraceloc to SASlog puts the trace into the regular log. NOTSTSUFFIX gives the short version which is quite long enough as demonstrated below:

```
libname prod Oracle user=&userid pw=”&passwd” ;

PROC SQL;
CREATE TABLE VALFLDS4 AS
SELECT
a.account number
,a.dwelling_type
,A.TERRITORY
,S.TERRITORY as S_territory
,case when S.territory = a.territory then 'MATCH  '
else 'NOMATCH'
end as territory_match_ind
,A.CUST_FIRST_NM
, S.CUST_FIRST_NM as S_CUST_first_nm
,case WHEN S.CUST_FIRST_NM = a.CUST_FIRST_NM then 'MATCH  '
else 'NOMATCH'
end as CUST_first_nm_match_ind
,A.CUST_MIDDLE_I
, S.CUST_MIDDLE_I as S_CUST_middle_i
,case WHEN S.CUST_MIDDLE_I = a.CUST_MIDDLE_I then 'MATCH  '
else 'NOMATCH'
end as CUST_MIDDLE_I_match_ind
,A.CUST_LAST_NM
, S.CUST_LAST_NM as S_CUST_last_nm
,case WHEN S.CUST_LAST_NM = a.CUST_LAST_NM then 'MATCH  '
else 'NOMATCH'
end as CUST_LAST_NM_match_ind
,A.CUST_NAME_SUFFIX
, S.CUST_NAME_SUFFIX as S_CUST_NAME_SUFFIX
,case WHEN S.CUST_NAME_SUFFIX = a.CUST_NAME_SUFFIX then 'MATCH  '
else 'NOMATCH'
end as CUST_NAME_SUFFIX_match_ind

from prod.cust_master as S
inner join list.campaign_curr_cust1 as a
on s.account_number=a.account_number;
```

The trace shows the code translated by SAS and passed to Oracle. Note there is an index on account_number but this request will not use it.

```
ACCESS ENGINE:  Exiting dbrqsub with SQL Statement set to
           SELECT  "ACCOUNT_NUMBER", "TERRITORY",  "CUST_FIRST_NM",
"CUST_MIDDLE_I", "CUST_LAST_NM", "CUST_NAME_SUFFIX”  FROM
PROD.CUST_MASTER
```

Instead it will do a full-table scan indicated by the trace showing the number of rows fetched as below:

```
ORACLE:  Rows fetched : 250
ORACLE:  Rows fetched : 250,
```

repeats hundreds of times until we exit the Access engine:

```
ACCESS ENGINE: Exit yoeclos with rc=0x00000000
NOTE: Table WORK.VALFLDS created, with 9461 rows and 65 columns.
NOTE: PROCEDURE SQL used (Total process time):
      real time           20:20.00
      user cpu time       3:41.19
      system cpu time     1:17.42
      Memory                           1867k
      Page Faults                      0
      Page Reclaims                    0
      Page Swaps                       0
      Voluntary Context Switches       195607
      Involuntary Context Switches     275468
      Block Input Operations           0
      Block Output Operations          60
```

We can run the exact same query with a smaller dataset The MULTI_DATASRC_OPT is set to 'IN_CLAUSE' (MULTI_DATASRC_OPT =IN_CLAUSE). The IN_CLAUSE constructs a 'IN' statement automatically which will facilitate use of the index:

First translation to SQL:

```
ACCESS ENGINE:  Exiting dbrqsub with SQL Statement set to
           SELECT  "ACCOUNT_NUMBER", "TERRITORY", "TDSP", "CUST_FIRST_NM",
"CUST_MIDDLE_I", "CUST_LAST_NM", "CUST_NAME_SUFFIX",
FROM
PROD.CUST_MASTER

ACCESS ENGINE: Exiting yoepnt() current_rid=1, next_rid=1
ACCESS ENGINE:  yoeget before read, current row=1, next row=1
ORACLE: orqsub()
ACCESS ENGINE: Entering dbrqsub
ORACLE: orqacol()
```

Addition of 'IN CLAUSE' to the SQL being passed to the database

```
ACCESS ENGINE: Add PROC where clause of  ( ("ACCOUNT_NUMBER" IN  (
'60113720001022077' , '60113720001110731' , '60113720001201137' ,
60113720001216732' , '60113720001261073' , '60113720001272777' ,
60113720007721773' ) ) )
ORACLE: orqacls()
ACCESS ENGINE:  Exiting dbrqsub with SQL Statement set to
           SELECT  "ACCOUNT_NUMBER", "TERRITORY", "TDSP", "CUST_FIRST_NM",
"CUST_MIDDLE_I", "CUST_LAST_NM", "CUST_NAME_SUFFIX",
FROM
PROD.CUST_MASTER  WHERE  ( ("ACCOUNT_NUMBER" IN  ( '60113720001022077' ,
'60113720001110731' , '60113720001201137' , '60113720001216732' ,
'60113720001261073' , '60113720001272777' ,
'60113720007721773' )
) )
ACCESS ENGINE: Return prep-only, with WHERE_BY option
ORACLE: orlock()
ORACLE:  READBUFF option value set to 250.
ORACLE:  Rows fetched : 100

NOTE: Table WORK.VALFLDS created, with 100 rows and 65 columns. NOTE: PROCEDURE
SQL used (Total process time):
      real time           0.00 seconds
      user cpu time       0.07 seconds
      system cpu time     0.02 seconds
      Memory                           1699k
      Page Faults                      0
      Page Reclaims                    0
      Page Swaps                       0
```

```
Voluntary Context Switches        21
Involuntary Context Switches      117
Block Input Operations            0
Block Output Operations           0
```

Note that the FETCH message appears only once for the 100 records returned. If you rewrote this query to have a right join because you wanted to bring back all of the dataset observations regardless of whether they match the table or not, you'll go back to a full table scan and fetch all of the records from that table before putting them with your dataset even though you only get 100 records back:

```
from prod.cust_master as S
right join list.campaign_curr_cust1 as a
on s.account_number=a.account_number

ORACLE:  Rows fetched : 250 (repeated a few hundred times)
ORACLE:  Rows fetched : 151
NOTE: Table WORK.VALFLDS created, with 100 rows and 65 columns.
```

In this case, it's probably better to do this in two stages. The first stage you would use the inner join to bring back all of the records that matched and the second do a merge to add back the ones that didn't.

SAS DICTIONARY TABLES

If you have RDBMS databases allocated to your session using the libname option, querying the dictionary tables will result in a dynamic call to the RDBMS for metadata. This can take some time. For example, if you have SAP allocated to your session via the SAS R/3 interface with trace turned on, you will see calls very much like this:

```
r3prep().
Entering send_metadata_request_and_load_metadata()
ACCESS ENGINE: Table returned is AEOI (Revision Numbers)
```

The above message will repeat until all of the SAP table information has been returned. Once all of the SAP metadata has been collected, your R3 session will close like this:

```
Entering r3close.
Leaving r3close.
```

Processing Oracle directory entries will look much like this:

```
ACCESS ENGINE: Entering ydedopn
ACCESS ENGINE: Using a utility connection for directory open
ACCESS ENGINE: Entering dbiopen
ORACLE: oropen()
ORACLE: ordesca()
ORACLE: ordesc()
ORACLE: DESCRIBE on ORACLE_0
ORACLE: ordtfmt()
ACCESS ENGINE: Entering ydedrd
ACCESS ENGINE: Table returned is ALL_OBJECTS
ACCESS ENGINE:  Exiting ydedrd with rc=0x00000000
```

The above message will repeat until all of the Oracle tables have been listed. Once that happens, your trace would show these calls:

```
ACCESS ENGINE: Entering dbiclose
ORACLE: orclose()
```

If you don't need to do a search for RDBMS information, clearing the libname with a statement like this will make the dictionary run a lot faster: `Libname clear` *`rdbms_name`* `;`

LIBNAME OR PASS-THROUGH?

Besides the ability to graphically set up a query in SAS® Enterprise Guide, there are other benefits to using the libname option.

Once a dataset gets to about 5000 observations, it's a good idea to get user space on your database and then load a user-defined table. The author has found this effective if a dataset's observations are 10% or lower than the number of rows in the smallest RDBMS table involved in the query. Libnaming a database is a comparatively simple way to accomplish this. The libname statement below assigns the name s_box to a Teradata database.

```
libname s_box  Teradata database = s_box server = viptdrop
dbcommit=0 override_resp_len=YES  connection = shared
user = &userid       password=&passwd;
```

Breaking this down into its components

`libname s_box`	- the name we will assign to the database
`Teradata`	- type of database, Teradata in this case
`database = s_box`	- actual name of database (optional)
`server = viptdrop`	- server through which we are accessing the database (required if you have more than one server)
`dbcommit=0`	when to commit changes – 0 means after each data step or query is completed
`override_resp_len=YES`	- the default length of the buffer used to hold data returned from Teradata to SAS will be set by Teradata not SAS
`connection = shared`	- used to eliminate any deadlocks loading tables within a single tablespace
`user = &userid`	- your userid on this system
`password=&passwd;`	- your password

We drop the table if it is present.

```
proc sql noprint;
drop table s_box.send_curr;
quit;
```

Then we set up the load statement. Fastload = yes loads in blocks not record by record. It ignores the referential integrity and other constraints that really slow down inserts. We explicitly describe the fields in the new table and create a primary index on acct. We use the usual set statement to load the table but find we need the output statement.

```
data s_box.send_curr(fastload=yes
dbtype = (acct = 'char(16)'             -explicit definition of fields in new
table
          lname= 'char(25)'
          fname = 'char(20)'
          ssn = 'char(9)'
)
Dbcreate_table_opts= 'primary index(acct/nomiss));   - create index on acct
Set sload;                                      - your load file
By acct;
Output s_box.send_curr;
Run;
```

Once the table is loaded, you may use either the libname or Pass-Through option to join it to your other RDBMS tables. To use your finder table in Pass-Through you'd write code like this:

```
connect to teradata(database = s_box user=&usrid password=&passwd.);
 create table curr_info as
 Select * from connection to teradata(
select a.acct, a.lname, a.fname,b.last_name,b.first_name,
```

```
b.curr_acct_bal, b.last_payment date
from s_box.send_curr inner join
custprod.account_info
on a.acct=b.acct_nbr
)
;
quit;
```

One disadvantage to the libname option is depending on your RDBMS, you may find yourself timing out if you keep your SAS session option long enough. Then when you attempt to use the libname, it will give you login errors and probably suspend your RDBMS account. There's no real way around this but to execute your libname statement occasionally.

Pass-Through is usually better if you are doing a complex query using tables only in the database or using a function or syntax that is specific to that database system. Oracle, DB2 and Teradata will generally allow you to create temporary tables within the same query with the WITH construct. The example below is a simulation of proc freq which may perform better than a proc freq using the libname:

```
with total_count as
(select
count(*) tot_count
from leads.campaign_records
)
,Per_value
as
(select count(*) cnt_value,
Channel_code
From leads.campaign_records
Group by channel_code
)
Select
Channel_code
,cnt_value
,tot_count
,cast(100*(cnt_value/tot_count) as number(6,2)) as percent_cnt
From total_count,
Per_value
;
```

SAS simply cannot translate this as this construct doesn't exist in its syntax:

```
___
           180
ERROR 180-322: Statement is not valid or it is used out of proper order
```

Since SAS is being translated to native SQL code, the more complex a query is the more likely using libnamed tables will result in strange code, perhaps not being executable or even worse possibly hanging up both your SAS session and the RDBMS.

Another advantage to using Pass-Through is the majority of RDBMS's will let you see how a query will execute before actually running it. Native SAS has no such facility. You will generally have another interface available to run an EXPLAIN PLAN statement, even if something rather primitive like SQLPlus or Beteq. Explaining the plan for the previous query might result in something like this:

```
Explain plan for
WITH SUM_CONTACT AS
(SELECT
COUNT(*) AS NO_OF_CONTACTS
, MIN(CONTACT_DATE) AS FIRST_CONTACT_DATE,
MAX(CONTACT_DATE) AS LAST_CONTACT_DATE
, CUSTOMER_ID
FROM LEADS.CAMPAIGN_RECORDS GROUP BY CUSTOMER_ID)
SELECT
```

```
A.CUSTOMER_LAST_NAME
, A.CUSTOMER_TELEPHONE
, A.CUSTOMER_BIRTH_DT,
SUM_CONTACT.NO_OF_CONTACTS
, SUM_CONTACT.FIRST_CONTACT_DATE
, SUM_CONTACT.LAST_CONTACT_DATE
FROM LEADS.CAMPAIGN_RECORDS A,
SUM_CONTACT
WHERE A.CUSTOMER_ID = SUM_CONTACT.CUSTOMER_ID;
```

Your plan might look like this:

```
--SELECT STATMENT HINT*ALL ROWS
       - HASH JOIN
              - VIEW
                     - HASH GROUP BY
                            --TABLE ACCESS FULL              CAMPAIGN_RECORDS
                     -TABLE ACCESS FULL                     CAMPAIGN_RECORDS
```

This is possibly one of the most expensive queries you could run. Hash joins do not use indexes and perform full table scans. It will build a hash structure for the second input before reading each row from the first input one at a time. If you find people need the SUM_CONTACT table, it might be worthwhile creating it regularly, either as dataset or a table.

The general rule of thumb is that it is better to use an index than scanning the entire table. However, that depends upon the columns of the index being used and more importantly their selectivity. The more selective the index is, the more efficient. In our query using account number, selecting 100 out of the millions of accounts made using an index very efficient. Attempting to retrieve five hundred thousand records may make the index worse than useless. Your DBA can help you understand these access paths and tune your query.

MACRO VARIABLES AND PROC SQL

The automatic macro variable SQLOBS is set to the number of observations returned from PROC SQL whether executed against datasets or tables. This value can be stored and used elsewhere such as this macro for creating an empty report:

```
%macro empty_report(numobs);
       %if &numobs = 0 %then %do;
              data _null_;
              file print;
              put 'Nothing to Report';
              run;
       %end;
%mend empty_report;
proc sql;
Select *
from SVCPdata.client
where lastname = 'Buffet';
quit;
%empty_report(&sqlobs);
run;
```

Output when there are no observations:

Nothing to Report

If you are using Pass-Through, two other informative automatic variables are available to you, SQLXMSG and SQLXRC. SQLXRC will return the RDBMS return code and SQLXMSG the associated message. These will give you the RDBMS specific error codes which you can look up, then take to your DBA if you need more help.

CONCLUSION

Understanding data models should aid you in understanding your data and therefore in turning data into good information. Remembering to check your code, deciding when to use libname or Pass-Through, creating a finder table in your RDBMS when appropriate and allowed and understanding RDBMS access paths should go a long way in making your code run efficiently with little intervention needed from either your DBA or system administrator.

RECOMMENDED READING

PROC SQL by Example – Howard Schreier, especially if you are new to PROC SQL in general
Online SAS Manual – SAS Insitute, particularly the SAS/Access section for more details on your particular RDBMS.

ACKNOWLEDGEMENTS:

Thanks to Joe and Paul Butkovich for your encouragement and support.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. The author is often on the road but can be contacted at

Patricia Hettinger
Email: patricia_hettinger@att.net
Phone: 331-462-2142