Paper 79-2019

Getting the Message Across: Cutting Run Times and Updating Databases with SAS/ACCESS®

Elise S. Mara, San Francisco Department of Public Health

ABSTRACT

When SAS® communicates with a database, efficient coding can mean the difference between a few minutes and a few hours! It is often easy to find tips on how to read data into and out of Excel spreadsheets, text files, and the like, but for users without experience in database administration or networking, getting SAS to talk to a database may seem like a trickier endeavor. Fortunately, SAS has user-friendly functionality to establish connections, pull data, alter database contents, and take advantage of server resources. This paper will outline how to use ODBC and LIBNAME statements to connect to databases, issue basic SQL procedure commands to perform data selection and manipulation, and craft pass-through queries to have the database server do the heavy lifting for you.

INTRODUCTION

When I started out as a SAS programmer, I was taught how to read in and output data in various formats. My instructors placed a great deal of emphasis on contending with fixed-width columns, different types of delimiters, and the challenges inherent to ensuring variable types and formats were preserved correctly. Missing from this training was a topic vital to many of our professional lives: what do you do when the data is kept in a database?

Over time, I have tried to tackle this question with varying degrees of success, and I witnessed several more experienced programmers fall into many of the same traps. Relational databases pose unique challenges that strategies designed for flat files often do not effectively address.

Here are some examples of common mistakes I have seen (or have made) when working with database data:

- Reading entire database tables into SAS when you only need a subset of data
- Outputting data into a different format (a Microsoft Excel spreadsheet, for instance) in order to read it into SAS or upload it to a database
- Running complicated data manipulation procedures locally instead of on a database server

If you have the access rights you need on a database you are working with, you should not have to resort to these measures.

One reason why you should try to avoid these measures has to do with the way that SAS uses memory. Large database management systems (DBMS), such as Microsoft SQL Server or Oracle, perform data manipulation in memory, rather than reading from and writing to disk each time a step is carried out.[1,2] SQL Server, for instance, employs a strategy that involves copying data into a memory buffer when it needs to be changed, then gradually writes changes to the data back to disk when that data hasn't been touched in a while.[1] The advantage of relying heavily on memory is that accessing memory is much quicker than disk I/O. SAS, on the other hand, keeps datasets on disk and alters them there.[3] This technical limitation means that making a SAS dataset out of a very large database table or running a program that makes changes to large datasets can take considerably longer than letting a DBMS handle the brunt of the work.

Another reason to avoid these strategies is more obvious: having fewer steps in a program makes your code more portable. Many of us have had the experience of inheriting a convoluted legacy program referencing a plethora of file locations, formats and macros that are anything but transparent. Requiring the user to employ a spreadsheet or text file as a way-station between the database and SAS adds another layer of time, training and potential access issues. If you confine the data transfer steps to SAS, then it will be much easier to share your program with others.

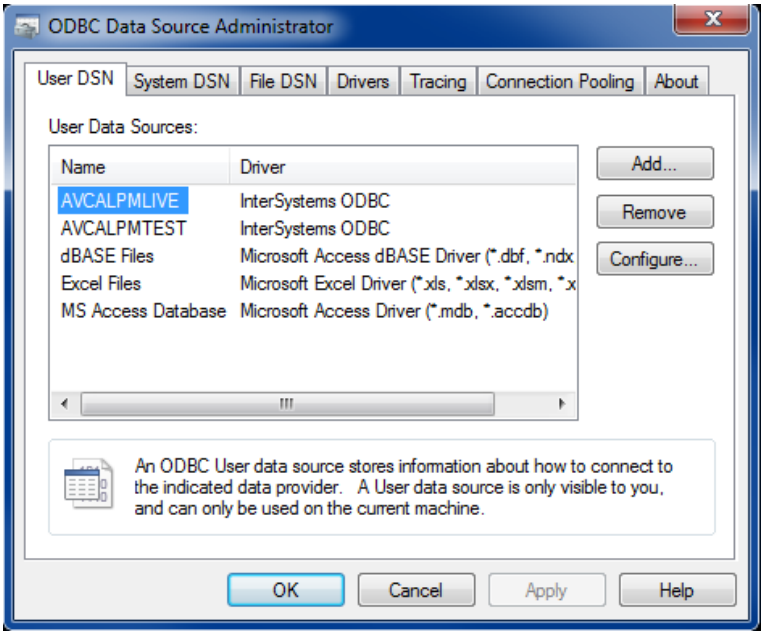
Through the course of this paper, I will describe ways to communicate efficiently, effectively and concisely between SAS and a database. In my own work, these techniques allowed me to transform a legacy data-matching program that took over an hour to run into one that took only minutes, while still ensuring that colleagues without a deep knowledge of SQL could use it. You do not need to be a database administrator or an expert in networking to take advantage of SAS' capacity for database communication.

SETTING UP A COMMUNICATION CHANNEL: ODBC AND DSNS

ODBC, or Open Database Connectivity, is a widely used application programming interface (API) that allows applications to communicate with outside data sources. In ODBC, a driver transmits commands issued by an application to the data source, such as SAS asking to pull in data from a database. A Data Source Name (DSN) stores information on how to connect to the data source, including which driver conducts the communication.

The drivers you need to communicate with a database should come as part of the DBMS installation—if you are not running the DBMS software on your machine, you may need to download them separately. Make sure that the version of SAS you are using has the SAS/ACCESS engine installed. (If you are not sure if SAS/ACCESS is installed, run the SETINIT procedure or the PRODUCT_STATUS procedure to check.)

Here is an example of what the ODBC data sources menu in Windows looks like:



Display 1: Windows ODBC Menu

You will notice that there are three types of DSNs listed:

- *User DSNs* are specific to the current user on the current machine.
- *System DSNs* are specific to the machine itself. (You need administrator privileges on your machine to set up a system DSN.)
- *File DSNs* are files containing data source connection instructions, which are not specific to a user or machine.

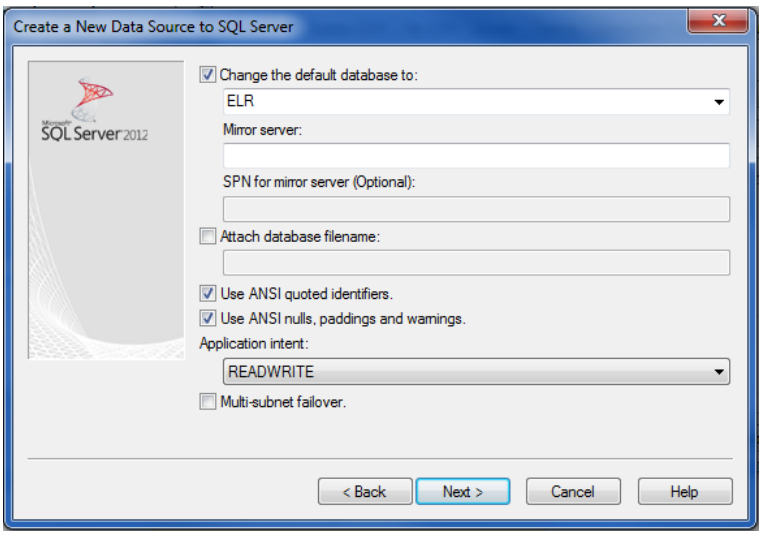
User and system DSNs are collectively known as "machine data sources" and are stored in the Windows registry of your machine. You can save a file DSN as you would any other file, such as in a local folder or on a shared network drive. Bear in mind that if you share a SAS program that references a DSN with a colleague working on a different machine, in the case of a user or system DSN, they will need to create the DSN on their own machine; in the case of a file DSN, they will need access to the file's location.[4]

Once you determine which DSN type is most appropriate for your needs, you can click “Add” on the menu and follow the instructions to create it. Windows will ask you to select a driver, fill in or navigate to the location of the data source, and supply any credentials needed to access the data source.

Something important to note is that Windows stores drivers for 64-bit applications and 32-bit applications in separate menus. For instance, to create a DSN for a 32-bit Microsoft Access database running on a 64-bit operating system, you would need to run this line to open the 32-bit ODBC menu:

```
c:\windows\sysWOW64\odbcad32.exe
```

Some data sources may have additional options associated with them. When creating a DSN pointing to a server running SQL Server, for example, you can specify which database on the server you want to reference by default. If you do not specify one, the DSN will point to the default database associated with your login.



Display 2: Additional Options on ODBC Menu for SQL Server

The next step is to show SAS where the database is by directing SAS to the DSN. You do this using a LIBNAME statement that, in the case of a user or system DSN, references the name of the DSN. Here, I am referencing the database “ELR” using a DSN of the same name, and giving it the libref “mydata”:

```
LIBNAME mydata odbc dsn="ELR";
```

If I wanted to reference a file DSN containing the connection information instead, I could run this statement, which shows SAS the file location:

```
LIBNAME mydata odbc noprompt="filedsn=d:\share\hereisELR.dsn;";
```

If you are connecting to SQL Server, it is also possible to create a connection without setting up a DSN at all, though you will need to supply all the connection information in your LIBNAME statement. Here is an example of a connection string that uses SQL Server authentication. Note that this may not be a great idea from a security perspective because your password is stored in plain text in your code:

```
LIBNAME mydata odbc noprompt="driver=SQL Server Native Client
10.0;uid=emara;pwd=elisespassword;database=ELR;server=ELR_server";
```

Here, I am connecting to the same database as above with Windows authentication (`Trusted_Connection=yes`). I have also included an additional option: a particular schema. A schema in SQL Server is essentially a collection of objects in a database; if you do not tell SAS specifically which

schema you want to reference, SAS will write any objects you create in the database to the default schema associated with your login:

```
LIBNAME mydata odbc noprompt="driver=SQL Server Native Client
10.0;server=ELR_server;database=ELR;Trusted_Connection=yes" schema=emara;
```

Once your LIBNAME statement runs, you can reference database tables and views as you would any SAS dataset, as in this example where I am listing patients with the last name “Smith”:

```
PROC PRINT data=mydata.tblPatient;
    VAR ln fn dob;
    WHERE ln like '%SMITH%';
RUN;
```

TALKING TO YOUR TABLES: QUERY EXAMPLES

You can use SAS DATA and PROC procedures to interact with database tables. Unsurprisingly, the most efficient way to withdraw data from or alter data in a relational database is with PROC SQL queries. Remember that SAS honors the access rights imposed by the DBMS; you cannot do things outside of the scope of permissions allotted to you on the DBMS.

READING DATA IN AND OUT

An easy way to create a dataset containing the information you want is by using the CREATE TABLE command. In this example, I am pulling a few columns of a patient table into SAS as a dataset called “patients”:

```
PROC SQL;
    create table patients as
    select ln, fn, dob
    from mydata.tblPatient
    ;
QUIT;
```

You can use joins and WHERE conditions to get the subset of data you want. Here, I am interested in getting information on patients that were seen at a clinic in 2018:

```
PROC SQL;
    create table seen as
    select ln, fn, dob, visitdate
    from mydata.tblPatient as p
    inner join mydata.tblVisit as v on v.id = p.id
    where v.visitdate >= INPUT('01/01/2018',mmddyy10.)
    ;
QUIT;
```

Another great use of joins is to take advantage of lookup tables in the database to decode values that may be cryptic. In this construction, I am interested in examining patient race as well, and I have joined the race lookup table so that I will know what the race codes in the patient table mean. The label variable from the race lookup table takes the name “race” in the output:

```
PROC SQL;
    create table seen as
    select ln, fn, dob, visitdate, label as race
    from mydata.tblPatient as p
    inner join mydata.tblVisit as v on v.id = p.id
    left outer join mydata.tlkpRace as r on r.race = p.race
    where v.visitdate >= INPUT('01/01/2018',mmddyy10.)
    ;
```

```
QUIT;
```

A database view (or “query” in Microsoft Access) is a dynamic table defined by a query. The data in a view updates when the underlying tables it draws from are updated. If you use CREATE TABLE with a view, the SAS dataset you create is essentially a snapshot of the data in the view at that given time:

```
PROC SQL;
    create table testresults as
    select testname, result
    from mydata.vwTests
    where testid = 1
    ;
QUIT;
```

You can make views in SAS too, which function similarly. This view references some data tables related to medical test results stored on the database, is updated every time you run the SAS program, and makes the data available to SAS without writing it to a SAS dataset:

```
PROC SQL;
    create view testresults as
    select testid, testname, result
    from mydata.tblTests as t
    left outer join mydata.tlkpTestName as n on n.testid = t.testid
    where t.testid = 1
    ;
QUIT;
```

Because SAS references the database connection as a LIBNAME, you can use libref syntax to easily write tables to the database:

```
PROC SQL;
    create table mydata.patients as
    select ln, fn, dob
    from mydata.tblPatient
    ;
QUIT;
```

If your data has gone through a lot of post-processing in SAS, passing the data back into the database may be easiest with a simple DATA step. Bear in mind that in a large database environment you may have relatively broad read permissions, but you may only be able to write data tables to a specific location. This example writes data to my personal schema on a database:

```
DATA emara.people;
    SET seen;
RUN;
```

INSERTIONS AND UPDATES

Appending new records to an existing table that you have permission to write to is easy with PROC SQL. The example below adds medical record numbers to a database table from a SAS dataset. A join to the database table is included to prevent SAS from trying to insert records that already exist in the table:

```
PROC SQL;
    insert into mydata.tblMedRec (id, medrecno)
    select s.id, s.medrecno
    from sasdataset as s
    left outer join mydata.tblMedRec as m on m.id = s.id and m.medrecno =
s.medrecno
    where m.id is null
```

```
    ;
QUIT;
```

Alternatively, if you only need to insert a few rows and are confident that they will not cause a problem, you can directly specify the values you want to insert without referencing a SAS dataset at all:

```
PROC SQL;
    insert into mydata.tblMedRec (id, medrecno)
    values (1111, 22222)
    values (2222, 33333)
    ;
QUIT;
```

Row deletions are also possible with PROC SQL, though since this sort of command can be destructive, be sure to test your code and make sure it will work as intended. Not specifying a WHERE clause deletes all the rows in the table, while the WHERE in this example filters the command to rows that contain certain values:

```
PROC SQL;
    delete from mydata.tblMedRec
    where id=1111 and medrecno=22222
    ;
QUIT;
```

To change data in existing rows, use an UPDATE command. Here is a simple example that applies the update for one or more rows with a particular id value:

```
PROC SQL;
    update mydata.tblMedRec
    set medrecno=22222
    where id = 1111
    ;
QUIT;
```

If you want to apply an update based on values in a SAS dataset, this syntax becomes somewhat more complicated. Unlike the INSERT command, which allows tables or datasets to be referenced through joins, the UPDATE command in SAS requires that they be referenced through nested SELECT statements. In addition, if every row in the database table is not accounted for in the SAS dataset—which is usually the case—you need to include a WHERE clause to limit the update to only the matching rows. If you do not include the WHERE, *the variable you are trying to update will be set to null in the non-matching rows.*

Below, I am updating the medrecno variable in tblMedRec. The first nested SELECT statement tells SAS to retrieve the value from the dataset “sasdataset” where the id in “sasdataset” matches the id in tblMedRec. The second nested SELECT statement tells SAS to only update matching rows and ignore everything else in tblMedRec. If I omitted the second statement, the value of medrecno in non-matching rows would be lost.

```
PROC SQL;
    update mydata.tblMedRec as m
    set medrecno= (
          select medrecno
          from sasdataset as s
          where s.id = m.id)
    where exists (
          select 1
          from sasdataset as s
          where s.id = m.id)
```

```
    ;
QUIT;
```

A slightly different version of the above code can accomplish the same thing using “in” instead of “exists”:

```
PROC SQL;
    update mydata.tblMedRec as m
    set medrecno= (
          select medrecno
          from sasdataset as s
          where s.id = m.id)
    where id in(
          select id
          from sasdataset as s)
    ;
QUIT;
```

WHY DID MY QUERY FAIL?

Aside from more obvious issues related to basic syntax, there are various reasons why a query to the database may raise an error message in the log. Here are some examples:

```
ERROR: Value 1 of VALUES clause 1 does not match the data type of the corresponding column in the
       object-item list (in the SELECT clause).
ERROR: Value 2 of VALUES clause 1 does not match the data type of the corresponding column in the
       object-item list (in the SELECT clause).
ERROR: Value 3 of VALUES clause 1 does not match the data type of the corresponding column in the
       object-item list (in the SELECT clause).
```

Output 1: Error from Type Mismatch in INSERT Query

Data types of variables in your SAS dataset must correspond to those of the variables in the database table; broadly speaking, you cannot write a numeric value to a variable defined as a string, and you cannot write a character value to a variable defined as an integer or binary. Depending on the DBMS, numeric variables formatted as dates may sometimes produce garbled output without an accompanying error message. For instance, dates without a time component have to be reformatted when passed into a Microsoft Access datetime variable, as in the code below:

```
PROC SQL;
    insert into mydata.tblPatient (ln, fn, dob)
    select ln, fn, DHMS(dob,0,0,0)
    from sasdataset
    ;
QUIT;
```

```
ERROR: Execute error: [Command::Execute failed. : The statement has been terminated.: Violation of
       PRIMARY KEY constraint 'PK_[redacted]'. Cannot insert duplicate key in object
       'dbo.[redacted]'.
NOTE: This insert failed while attempting to add data from VALUES clause 1 to the data set.
ERROR: ROLLBACK issued due to errors for data set [redacted].
```

Output 2: Error from Primary Key Violation in INSERT Query

You cannot make changes in a database that violate existing constraints on database objects. Thankfully, these violations usually result in descriptive error messages, as in the above example. The database table involved had a primary key constraint on it, which mandated that a particular variable in the table had to have a value and that that value needed to be unique to each row. When I tried adding a row with a value for that variable that already existed in the table, the DBMS prevented me from doing so and returned this error to SAS.

```
ERROR: The OLEDB table [redacted] has been opened for OUTPUT. This table already exists, or there is a
       name conflict with an existing object. This table will not be replaced. This engine does not
       support the REPLACE option.
```

Output 3: Error from Attempted Table Overwrite

Unlike a SAS dataset, you cannot simply overwrite a database table; you have to drop it and recreate it. You can easily accomplish this through PROC SQL:

```
PROC SQL;
    drop table emara.temporary
    ;
QUIT;
```

```
NOTE: SAS variable labels, formats, and lengths are not written to DBMS tables.
ERROR: Error attempting to CREATE a DBMS table. ERROR: Execute error: ICommand::Execute failed.
       : CREATE TABLE permission denied in database '[redacted]'..
```

Output 4: Error from Permission Denial

If you do not have sufficient privileges on a database to carry out an action, you will receive an error message like this. In this example, I attempted to write a SAS dataset to a schema where I did not have permission to create or drop database objects.

LET THE SERVER HANDLE IT: PASS-THROUGH QUERIES

The real power of the SAS/ACCESS engine rests in its ability to run commands on the DBMS without ever having to leave SAS. These commands are called “pass-through queries”, and while they are introduced with PROC SQL, they use the DBMS’ native language and are carried out as though they were sent through the DBMS console itself.

PASS-THROUGH EXAMPLES

The connection information takes a similar form to the LIBNAME statement. Place code you want to run on the DBMS between the innermost parentheses.

Here is a simple example of how to connect via a user or system DSN:

```
PROC SQL;
connect to odbc as ELR
    (DSN="ELR");
execute(


)
by ELR;
QUIT;
```

Here is the construction for a file DSN:

```
PROC SQL;
connect to odbc as ELR
    (noprompt="filedsn=d:\share\hereisELR.dsn;);
execute(


)
by ELR;
QUIT;
```

And here is the DSN-less construction using a connection string:

```
PROC SQL;
connect to odbc as ELR
    (noprompt="driver=SQL Server Native Client
10.0;server=ELR_server;database=ELR;Trusted_Connection=yes");
execute(


)
by ELR;
QUIT;
```

Pass-through queries can be faster than referencing tables through a libref when you are running a SELECT query that involves many joins or complicated aggregations. A compact way to do this uses the syntax below, which carries out an aggregation on the DBMS and ports the results into a SAS dataset called “datainеed”:

```
PROC SQL;
connect to odbc (noprompt="driver=SQL Server Native Client
10.0;server=ELR_server;database=ELR;Trusted_Connection=yes");

create table datainеed as
select *
from connection to odbc (

    select id, max(visitdate) as recentvisit
    from tblVisit
    group by id

);
QUIT;
```

Perhaps the most useful application of pass-through queries, however, is their ability to call functions and stored procedures through the DBMS. This includes functions and procedures standard to the DBMS software—which may meet your needs more directly than their SAS counterparts—as well as those created by users that are stored on the DBMS. This example invokes a user-defined stored procedure in a SQL Server database, whose output you can retrieve in a later step:

```
PROC SQL;
connect to odbc as ELR
    (DSN="ELR");
execute(

    exec spMakeReportCard
)
by ELR;
QUIT;
```

You can even embed a stored procedure or function call in a SAS macro, which can be especially useful when you need to pass values to it. Here, I create a macro that allows me to define a date range used by a procedure:

```
%macro calldata (startdate, enddate);

PROC SQL;
connect to odbc as ELR
    (DSN="ELR");
execute(
```

```
    exec spGetSummary &startdate,&enddate

)
by ELR;
QUIT;

%mend;
```

A REAL PASS-THROUGH APPLICATION

If you are scratching your head about how to deal with a large volume of database data—think pass-through!

I inherited a SAS program written to match new test results received from laboratories with existing records in our laboratory data management system (LDMS). The program cleaned and reformatted the new data and carried out a series of matches via different sets of criteria to determine which patients were new to the system and which were known. It used a LIBNAME statement that mapped to a DSN and a DATA step to communicate with the primary table in the database.

The communication with the database took place in this form:

```
LIBNAME ldms odbc dsn='LDMS';

DATA labdata;
    SET ldms.alldata;
RUN;
```

The trouble with this program was that over the years this primary table had grown to include over a million records. Writing that table to SAS and joining to it repeatedly strained the resources of my desktop and stretched the run time of the program to over an hour.

I realized that since the DBMS handled complex operations in a more efficient way than SAS, offloading the matching routine to the DBMS should improve the program's performance significantly. Fortunately, I was administrating LDMS and was able to create stored procedures on the database, so I crafted a stored procedure that received the clean data from the SAS program, did the match, and passed a table updated with the match results back into SAS:

```
/* Match 1: true match on fn, ln, dob */

insert into #all (criteria, id, accession_no, ln, fn, mn, dob, newtestdate, recordid, ln_m, fn_m, dob_m, dov_m, result_m
,   site_m, mrno_m, last_disp)
  select 'FN+LN+DOB' as [criteria], b.id, b.accession_no, b.ln, b.fn, b.mn, b.dob, b.collection_date as newtestdate
  , l.recordid, l.ln as [ln_m], l.fn as [fn_m], l.dob as [dob_m], l.collection_date as [dov_m], l.result as [result_m]
  , l.site as [site_m], l.mrno as [mrno_m], l.last_disp
  from temp.tmpbatch as b
  left outer join vwlabmatch as l on l.ln = b.ln and l.fn = b.fn and l.dob = b.dob
  where l.recordid is not null
```

Output 5: Block of Code from Stored Procedure to Match Test Data

All that SAS had to do for this part of the program was transfer the clean data onto the database, call the procedure in a pass-through query, and receive the results table. The code looked something like this:

```
DATA tmp.tmpbatch;
    SET next2;
RUN;

proc sql;
connect to odbc as ldms
    (DSN="LDMS");
execute(
```

```
    exec splabmatch
 )
by ldms;
quit;

DATA matchall;
    SET tmp.tmpall;
RUN;
```

Instead of an hour, this block of code took less than a minute to run, and it reduced the overall execution time of the program to only a few minutes.

CONCLUSION

Determining how to handle data from a database in SAS should never be an afterthought. As in many programming languages, there is a variety of ways to approach a task in SAS, but rather than falling back on the familiar, it is best to think through how the communication ought to happen. Doing so will make your code easier to understand and share, and—if you have been guilty of dealing with database tables the same way as flat files—save you plenty of processing time in the future.

REFERENCES

1. 2019. “Memory Management Architecture Guide”. *Microsoft SQL Documentation.* Accessed June 25th, 2019. https://docs.microsoft.com/en-us/sql/relational-databases/memory-management-architecture-guide?view=sql-server-2017.
2. 2019. “Oracle Database: Database Concepts.” *Oracle Help Center.* Accessed June 25th, 2019. https://docs.oracle.com/en/database/oracle/oracle-database/19/cncpt/database-concepts.pdf.
3. Ruegsegger, Steve. 2011. “Programming Techniques for Optimizing SAS Throughput.” *Proceedings of the NorthEast SAS Users Group Conference.* Portland, ME. Available at https://www.lexjansen.com/nesug/nesug11/ld/ld04.pdf.
4. 2019. “Administer ODBC data sources.” *Access Help Center.* Accessed June 26th, 2019. https://support.office.com/en-us/article/Administer-ODBC-data-sources-B19F856B-5B9B-48C9-8B93-07484BFAB5A7.

ACKNOWLEDGMENTS

Many thanks to Paul Stutzman and Bob Kohn for their support and assistance in the preparation of this paper.

CONTACT INFORMATION

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

Elise S. Mara, Epidemiologist
San Francisco Department of Public Health
(415) 437-6379
elise.mara@sfdph.org