Remote and Open Access to Enterprise Data:
Enhanced Distributed Data Access Through
Multiple Engine Architecture

Jack Wallace, Bill Brideson, and Cheryl Garner,
SAS Institute Inc., Cary, NC

ABSTRACT

Access to data from the SAS® System is constantly being expanded and enhanced. This access is built on a modular system architecture, called Multiple Engine Architecture, that often provides more than one way to accomplish a task that involves distributed data. You can choose the best design for your application if you understand the processing performed by the various components.

This paper concentrates on the variety of data access capabilities provided by Multiple Engine Architecture and how those capabilities are implemented. If you simply want to know how to use the features, read the introduction and selected references at the end of this paper. If your interest extends to optimizations and processing details, you should find the body of this paper informative.

You are encouraged to also read the paper by Cheryl Garner, "The SAS® System: A Complete Client/Server Solution" in the SUGI 20 Conference Proceedings (see the list of references at the end of this paper). That paper compares and contrasts remote library services to other client/server features of the SAS System. For many SAS applications, these other features may provide enhanced performance.

INTRODUCTION

The features and their functionality are overviewed in this introduction. A more thorough understanding of the features will be gained as the architecture of the system and its components is explored in the next part of the paper, "ODBC and Multiple Engine Architecture Basics". Techniques for optimizing distributed data access are primarily covered in "Accessing Distributed Data from Multiple Sources".

Remote Library Services

Remote Library Services allows a SAS program to access a SAS data library on another machine as if the data library were on the local machine. The program accesses the data library through a SAS server on the remote machine.

To access a remote library, the SAS program simply executes the LIBNAME statement it would use on the remote machine with an additional option which specifies the name of the server on the remote machine:

LIBNAME xyz 'physical-library-name' SERVER=server_name;

SAS files in the remote library can now be manipulated by SAS programs on the local machine.

Remote library access has been available through SAS/SHARE® software for many years, but it was restricted to SAS programs running on the same machine architecture. For example, a SAS program on one MVS machine could access a SAS data library through a server on another MVS machine. Since 6.07, the SAS program or server could execute on MVS while the other executed on CMS because both MVS and CMS execute on the same IBM® System/370 machine architecture.

When accessing a SAS data library through a server on a different machine architecture, SAS data sets may be read and written. Note that SAS views, including SAS/ACCESS® views of DBMS tables, are a type of SAS data set, so DBMS tables on remote machines can be accessed when the local and remote machines have different architecture. SAS catalogs and some utility files can not be read and converted to a different machine architecture by Remote Library Services.

Open Access to SAS Data

The ability to access SAS data from other programming languages and tools has been an item of interest for a number of years. Developing a proprietary interface for each language and tool, though, has not been high priority at the Institute because of the volume of interfaces and relatively small market addressed by each individual interface.

SAS Institute has been interested in the open standards work for database access and were members of the SQL Access Group (SAG). We believe that open standards for data access are the most cost-effective way for the Institute and its customers to allow other programming languages and vendor products to interoperate with SAS data. The SAG Call Level Interface (SAG-CLI) specification and its derivatives appear to be the most promising standards for our use.

The SAG-CLI and ODBC specifications are only slowly being adopted in the UNIX market. Because of this, the Institute is developing a more simple and basic C runtime library interface for use under UNIX. Customers interested in this interface can obtain a beta copy from their SAS Institute marketing representative.

Several years ago, Microsoft modified and enhanced an early draft of the SAG-CLI and incorporated it in their Windows Open Systems Architecture (WOSA) as the Open Database Connectivity (ODBC) Interface. This is the first commercially available, standard implementation with significant market penetration and independent Software Vendor (ISV) support. (We'll see later that the ODBC implementation in Windows is very similar to the SAS Multiple Engine Architecture.) Many Windows applications and application tools such as Microsoft Word, Visual Basic, and the SAS System itself can utilize ODBC to access data in other DBMS systems.

An ODBC driver to access SAS data from ODBC applications is available. This driver communicates with a SAS server to access SAS data. The server may execute on the same machine, a remote machine with the same machine architecture, or a remote
machine with a different machine architecture. So, a Microsoft Word document (or other ODBC application) running on Windows is able to access SAS data on any machine in the network, including mainframes, minicomputers, and workstations.

Expanded Use of SQL Passthru

In order to fully implement the features above, we have expanded the use of SQL Passthru. Originally, SQL Passthru was designed to allow the SQL processor in the SAS system to accept SQL statements with the syntax of other DBMS systems and send those statements directly to the DBMS for execution. For example, PROC SQL can send SQL statements directly to a DB2 server for execution.

With Remote Library Services, the SQL processor can send an SQL statement through a SAS server to a DBMS system on the remote machine.

The SAG-CLI and ODBC standards for open access to data are based on the SQL language. The SAS SQL processor has been enhanced to accept SAS SQL statements through the same interface that it uses to pass DBMS SQL statements to a DBMS. Other modules have been added to accept the ODBC API, convert it to appropriate SAS SQL statements, and send it to the SAS SQL processor for execution through this passthru API. That architecture will be explored further below.

**ODBC and Multiple Engine Architecture Basics**

It is interesting to compare and contrast the ODBC and Multiple Engine Architectures. This section overviews the architectures and highlights their similarities and differences.

**Architecture Layers**

Figure 1 illustrates the basic components of ODBC and MEA.

**Multiple Engine Architecture**

- Procedures and ODBC Manager
  - Call Driver
  - Call Engine
  - Call User-Defined Function

**ODBC Architecture**

- ODBC Application
  - Connect to ODBC Manager
  - Call Driver
  - Call Engine

![Figure 1](image)

An ODBC application calls ODBC functions in the driver manager to submit SQL statements and receive results. The driver manager loads and transfers calls to the appropriate driver. The driver processes the ODBC function calls and translates them into the necessary function calls or actions to access a data source. The data source is the data that the application wants to access along with any operating system, database server, and communications software used to access the data.

SAS applications are composed of, among other things, procedures, DATA steps, and SCL code. These application components in turn access SAS data sets by calling routines in the engine supervisor. Depending on the type of SAS data source, the engine supervisor loads and calls an appropriate engine to access the data source. In figure 1 a tape data source and disk data source are depicted. Many other data source types are possible as we will see.

**Application Programming Interfaces (API)**

The Application Programming Interfaces really define the data model and methods to access data available to application programs from ODBC drivers or SAS engines. A full description of the APIs is beyond the scope of this paper.

Because different data sources are capable of providing different functionality, not all drivers and engines provide the complete functionality of their respective APIs. In both architectures, a part of the API are informational interfaces which the application can use to query the driver and engines about their conformance or support for other API functions.

**ODBC API**

As stated in the introduction, the ODBC interface between the application and the ODBC manager and driver is based on the SQL language. This includes routines to:

- connect to a DBMS,
- obtain descriptions of the tables and columns available,
- execute SQL statements,
- retrieve results,
- disconnect from a DBMS.

The interface between the ODBC manager and the driver is almost the complete ODBC API. The primary purpose of the manager is to load the appropriate driver for a data source. It does provide some initialization services and validates parameters and sequencing for application ODBC calls.

**SAS API**

The API between the application components and the engine supervisor for SAS data set access includes both navigational and SQL set based Interfaces.

The navigational interface includes routines to:

- open a SAS data set,
- obtain descriptions of the variables (columns) and indexes if any,
- read, update, or add records (rows or observations),
- fetch or replace variable values in the record buffer,
- restrict access to records qualified by a WHERE clause,
- position data set or index to a particular record or key value,
- close the SAS data set.

The navigational interface between the engine supervisor and the engine is a subset of SAS data set functionality. Common services in the engine supervisor removes the requirement to
duplicate all functionality in each engine. In particular, routines to
toch and replace variable values in record buffers are common
services and not passed to each engine. Instead, the engine
transfers records between the engine supervisor and the data
source.

Some functions may be executed by common services in the
engine supervisor or by the engine. The engine and engine
supervisor shake hands and give the processing to the engine if it
can optimize the processing. For example, restricting access to
records qualified by a WHERE clause may be performed by the
engine. Common services in the engine supervisor are used if the
engine does not support WHERE clause processing.

The SQL set interfaces are similar to dynamic SQL and the ODBC
API and is referred to as the SQL passthru API since the
application is really communicating directly with the engine.
Passthru interfaces are provided to:

- connect to a DBMS,
- prepare, describe, and open SQL SELECT statements,
- execute other SQL statements,
- retrieve results,
- disconnect from a DBMS.

Driver and Engine Classifications

It's useful to classify drivers and engines into several groups that
share similar attributes. Figure 2 shows basic driver and engine
types:

- single-tier and multiple-tier drivers,
- native, view, and REMOTE engines.

![Figure 2: Single-tier Drivers and Native Engines](image)

Single-tier drivers and native engines are similar in that they each
directly access the data files associated with the data sources.
They each contain the logic to process the API requests through
operating system functions to access the appropriate data files,
extracting the correct data, and delivering the requested results.
While the operating system may use file server technology to
store the files on a different machine, we assume the files are
stored on the machine where the engine executes and do not
consider that a form of remote access in this paper.

Native engines are used to process SQL statements that are
selected from a SAS system-defined format. The most important
native engines in any release is the BASE (or current version) engine. It
implements all the features of the current release SAS
data set model in the most optimal manner for disk access.
Compatibility engines support back release formats of the BASE
engine. Sequential engines store SAS data files on sequentially
accessed media such as tape. The transport engine stores SAS
data sets in a format that can be read by another machine type.
Only the BASE engine is guaranteed to support all the
functionality of the current SAS data set model.

Multiple-tier Drivers and other SAS Engines

Multiple-tier drivers are similar to the REMOTE engine and
SAS/ACCESS engines in that they send requests to another
process called a server for processing. The server process may
execute on the same machine as the application or on a different
machine.

The multiple-tier driver simply converts the ODBC functions to
the appropriate API functions calls for the data source. The data
source is then responsible for communicating the request to a
database server and returning the results.

SAS Engine and SAS Driver Processing

This section discusses the actual processing of SAS engines and
the SAS/ODBC Driver. While it is not necessary to know this
material in order to use the engines or driver, it is useful when
discussing the optimizations for accessing data from multiple
sources.

View Engine Processing

A view is a type of SAS data set that is materialized from other
data sources based on descriptions of the other data or
processing instructions which are stored in a VIEW file. A VIEW
file is processed by a particular view engine to materialize the
SAS data set it represents.

Common processing of view engines is depicted in figure 3.

![Figure 3: View Processing](image)
The application requests that the engine supervisor open a SAS data set in a library that is accessed by a native engine.

The engine supervisor determines that the name refers to a file of type VIEW instead of type DATA. It calls the native engine to open the view file and read the view header record. The header contains the name of the engine that interprets the view. The supervisor loads and calls the view engine to complete opening the SAS data set. The view engine reads data descriptions or instructions from the view file and closes it. It opens other data sources to complete the open processing.

All subsequent access to the open SAS data set is passed to the view engine. It accesses data from the other data sources and materializes SAS data records for the engine supervisor.

In the remaining discussions of view engines, access to the VIEW file is ignored except where it plays a particular role in remote access.

In this paper, the term logic engine is introduced to refer to view engines that programmatically access data in other sources according to stored program logic in a VIEW file. Figure 4 depicts and describes the two logic engines in the SAS System today.

The DATA step view engine does not optimize access according to WHERE clauses specified for a view. The engine supervisor always filters the records that are returned from the DATA step view.

SAS/ACCESS engines are used to access data in other formats. In most cases, the data are formatted by a DBMS and the engine uses the DBMS to access the data as depicted in figure 5.

The view file itself contains descriptions of the data that is in the DBMS table. The engine converts requests to access records from the engine supervisor to the appropriate DBMS system function calls. The DBMS function calls usually communicate the request with a DBMS server to access the database. Data (row) are returned through the DBMS functions to the engine, engine supervisor, and application program.

REMOTE Engine Processing

The REMOTE engine is used to access SAS files through a SAS server for one of two reasons:

- to share update access to the SAS file with other SAS programs reading or updating the same file,
- to access a SAS file on a remote machine.

As depicted in figure 6, the REMOTE engine communicates the
Systems Architecture

Accessing a SAS file through a server is always slower than directly accessing the file because of the extra code to communicate the request and pass it to the engine in the server. The response time cost of this communication increases when the server is executing on a different machine than the application program and varies amongst different communications protocols and network topologies. Optimizing the communications costs is essential in remote data access applications.

The REMOTE engine does optimize access time by reducing the number of communications between the REMOTE engine and the server. Not all engine calls result in calls to the server. If the call is requesting descriptive information retrieved when the SAS file is opened, no new communications are required. In addition, the server returns information about each record with the retrieval call. It returns multiple records in one call for sequential input access (and for random input access if the TOSBSSNO= option was specified). Subsequent requests for information or records which the REMOTE engine already has in its control blocks or buffers, do not result in communications to the server.

The application program or user can also reduce the number of communications by utilizing WHERE processing. The REMOTE engine always tells the engine supervisor that it can optimize WHERE processing and passes the WHERE clauses to the server. The server then shales hands with the real engine to determine if the engine or the server will qualify records with the WHERE clause. In either case, only records that qualify are ever sent back across the communications link to the REMOTE engine.

As mentioned in the introduction, the REMOTE engine and server are required at first to execute on the same machine architecture. Now, the REMOTE engine and server exchange information on each other's architecture when communication is established. If the architectures differ, machine code is generated to translate the data streams with the following priorities:

1. the REMOTE engine assumes responsibility (translates directly to and from the server representation) if it can,
2. the server assumes responsibility (translates directly to and from the REMOTE engine representation) if it can,
3. they share responsibility (each side translates to and from a transport format).

Only those pieces of data which need to be translated are translated. By generating machine code, the fastest translation path length is produced.

SAS ODBC Driver Processing

ODBC application access to SAS data is depicted in figure 7. It's easy to see that the SAS ODBC driver is a multiple-tier driver. The data source components of the SAS implementation consist of a SAS server accessed through a Remote Engine Emulator and Communications Access Method Emulators.

The SAS ODBC driver converts the ODBC API to the passthru API and calls the REMOTE engine emulator. The REMOTE engine emulator formulates the correct communications messages and calls the appropriate communications access method emulator to send the request to the SAS server and receive the response.

Figure 7

In the SAS server, the communications and requests look almost identical to requests from a SAS application using the passthru API. The request is passed to the Passthru API of the SQL processor in the SQL engine. It then processes the SQL query or statement as it would from any other source. The results are returned from the server through the emulators to the driver.

Accessing Data from Multiple Sources

Multiple Engine Architecture and the many engines in the SAS system make it possible to access data from multiple sources in many different ways. Each topic in this section will be approached from three points of view:

- A picture that shows schematically the software components and the flow of the data.
- A SAS program that shows programming techniques you would use to implement the arrangement of processing and data flow.
- A narrative description of what you must consider for the arrangement of processing and data flow. Particular attention will be paid to the rule that says "Move data selection and manipulation logic as close to the data sources as possible."

This section will begin with a single SAS session that uses multiple sources of data and will progress to multiple SAS sessions. The section will finish with a discussion of SQL passthru and Remote SQL passthru and the very fine level of control afforded by those facilities.

Joining/Merging Data in a Single SAS session

Two interesting cases are joining SAS data with DBMS data and joining DBMS data with DBMS data.

Joining SAS Data with DBMS Data

A logic engine can be used to join or merge data from multiple SAS data sets. Some of the SAS data sets may themselves be SAS/ACCESS views of DBMS data. Figure 8 illustrates a logic engine accessing data from a native engine SAS data file and a SAS/ACCESS view of a DBMS table.
This PROC SQL step creates a view that will be interpreted later by the SQLVIEW engine (the logic engine which is at the heart of this example).

This SELECT statement selects rows from the DBMS whose 'last name' value is 'Hicks', and uses the company code to combine each of those with information from the company's row of the SAS data file.

```
proc sql;
create view viewlib.cohicks as
  select lname, o.compcode, company
  from sasdata.company as s,
  viewlib.people as o
  where s.compcode = o.compcode &
  lname = 'Hicks';
quit;
```

PROC PRINT prints a SAS data set, and this particular SAS data set is a SAS data view that was created by the PROC SQL step above.

To supply the data to PROC PRINT, the SQLVIEW engine serves as a logic engine, interpreting the instructions in the view.

This particular view instructs the SQLVIEW engine to read data from a DBMS and a SAS data file. The SQLVIEW engine will use a SAS/ACCESS engine to read rows from the DBMS whose 'last name' value is 'Hicks'. The SQLVIEW engine will also use the SAS base engine to read observations from a SAS data file whose company code matches the company code that was read from the DBMS.

The data from the two sources will be combined by the SQLVIEW engine and presented to PROC PRINT as observations of an (abstract) SAS data set.

```
proc print data=viewlib.cohicks;
run;
```

This example does the best it can to put the data selection and manipulation logic as close to each data source as possible. The SQLVIEW engine automatically passes selection criteria to underlying engines whenever possible. In this example the SAS/ACCESS engine will be given a WHERE clause by the SQLVIEW engine and the base engine will participate in optimizing selection depending on heuristic determinations made in the SQLVIEW engine. So the data selection in this example is performed by the SAS base engine and by the DBMS. The data manipulation (combining the data into observations) is performed by the SQLVIEW engine.

The SQL view engine uses indexes to optimize join processing whenever possible, but in version 6 only the SAS base engine is able to surface indexes to the SQL view engine. This means that the SQL view engine can not optimize access with knowledge of DBMS indexes in version 6.

This example used PROC SQL to create an SQL view, but an alternative solution might use a SAS DATA step to create a DATA step view. (A DATA step view is interpreted by the DATA step view engine, which is a logic engine like the SQLVIEW engine.) A DATA step view would be appropriate for more complex selection or manipulation criteria, or logic that's easier to represent as a
DATA step program. A DATA step program might either:

- reference the SAS data file and the SAS/ACCESS view in a
  MERGE statement with a BY statement to specify the merging
  criteria, or

- programmatically access data from each source through
  multiple SET statements.

**Joining DBMS Data with DBMS Data**

Logic engines are an effective solution for joining or merging data from two or more SAS/ACCESS views of database tables when:

- the database tables are in different DBMS servers, or
- the DBMS cannot duplicate the logic in the view.

**Figure 9**

An example of combining data from two DBMS tables is:

```sql
PROC ACCESS;  /*
  Situation: The demographic information for each
  ZIP code is stored in one DBMS table, and the names
  and addresses of the people participating in a
  study are in a second DBMS table.

  Task: Given the name of a person, obtain the
  demographic information for the person's ZIP code.
  */

libname vienlib 'dxms_views';

/* When the system was set up, PROC ACCESS was used to
   create a set of SAS/ACCESS views. This example uses
   two of those: "zipinfo," which is associated with a
   ZIP code table stored in an Oracle DBMS, and
   "subjects," which is associated with a subject
   information table that is stored in a DB2 DBMS.
   */

/* This PROC SQL step creates a view that will be
   interpreted later by the SQLVIEW engine (the logic
   engine which is at the heart of this example). */

PROC PRINT prints a SAS data set, and this particu-
lar SAS data set is a SAS data view that was
created by the PROC SQL step, above.

To supply the data to PROC PRINT, the SLOVIEW
engine serves as a logic engine, interpreting the
instructions in the view.

This particular view instructs the SQLVIEW engine
to read data from two DBMSes. The SQLVIEW engine
uses a SAS/ACCESS engine to read from each DBMS.
The SQLVIEW engine will instruct the SAS/ACCESS
engine that reads from the subject information
table to retrieve only rows whose "last name" value
is "Hicks".

The data from the two sources will be combined by
the SQLVIEW engine and presented to PROC PRINT as
observations of an (abstract) SAS data set.

Since the data sources in this example are different DBMSes,
putting the data selection and manipulation logic as close to each
data source as possible is a challenge. The SQLVIEW engine
automatically passes WHERE selection criteria to underlying
engines whenever possible, and in this example the
SAS/ACCESS interface to DB2 will be given a WHERE clause to
pass on to the DB2 DBMS. Data selection in this example is
performed by one of the DBMSes but, depending on how the data
in the Oracle DBMS is organized, some of the data selection
might also be done by the SQLVIEW engine as it combines data
from the two DBMSes. The data manipulation (combining the data
into observations) in this example is performed by the SQLVIEW
engine.

If the tables are accessed via the same DBMS and the necessary
data selection and manipulation can be performed by the DBMS,
then it is usually far more efficient to allow the DBMS to process
the join than it is to bring data back to a logic engine in the SAS
system to process the join.

To make a DBMS perform a join, define a view in the DBMS itself
that joins the base tables, and create a SAS/ACCESS view that
refers to that DBMS view (of the joined tables). Refer to the
SAS/ACCESS view in your SAS program without an intervening
logic engine (i.e., without defining an SQL view or a DATA step
view).

Of course, there is overhead to defining a view in a DBMS which

---

1520
you may not want to incur, or you may find you are not allowed to
define views in your DBMS. In these situations, you may be able
to use SQL paseinh, which is discussed below (in the section
"SQL Paseinh").

Joining/Merging Data that is Remote

The examples above paid close attention to the rule, "Move data
selection and manipulation logic as close to the data sources as
possible." When more than one SAS session is involved, it is even
more important to keep that rule in mind.

Local SAS Data with Remote SAS Data

Joining local SAS data with remote SAS data can be
accomplished in the same manner as joining local SAS data with
DBMS data described in figure 8. This is illustrated in figure 10.

Figure 10

An example of combining local and remote SAS data files is:

```/*
Situation: The title, author, and other informa-
tion about a book is stored in a SAS data file that
is accessed via a SAS server (because the file is
updated as new books are published). Each clerk is
given their own SAS data file that contains ISBN
numbers of books whose information must be checked.

Task: Given an ISBN, obtain the title, author,
publisher, and publication date of the book.

The first LIBNAME statement associates the libref
MYISBNs with a user's personal SAS library.

The second LIBNAME statement associates the libref
BOOKS with a library of SAS data files that is
accessed via a SAS server on a remote machine.

The third LIBNAME statement associates the libref
VIEWLIB with a library of SAS data views.
*/

libname ishns 'my_assignments';
libname books 'book_library' server=bookserv;
libname viewlib 'isbn_views';

/*
When the book information file was created, PROC
DATASETS was used to create an index on its ISBN
variable. That index is automatically maintained
by the SAS System as the file is updated.
*/

/*
This DATA step was used to create the SAS data
view MYBOOKS in the VIEWLIB library. It reads all
the observations in the SAS data file ISHNS.MYLIST
and, as each observation is read, the value of its
ISBN variable is used to index into the remote
master book information file to retrieve all of
the information about that book.
*/

data viewlib.mybooks / view=viewlib.mybooks;
set ishns.mylist;
set books.info key=isbn;
run;

/*
This PROC PRINT step prints the SAS data set
VIEWLIB.MYBOOKS, which is actually a DATA step
view that reads observations from a local SAS
data file and from a remote SAS data file via
a SAS server.

As the DATA step view engine interprets this view,
it will read an observation from the local SAS data
file and then it will read an observation from the
remote SAS data file. The view will terminate when
the end of the local SAS data file is reached.
*/

proc print data=viewlib.mybooks;
run;

In this example, both the data selection and manipulation logic are
in the local SAS session, which puts it close to the local SAS data
but far from the remote SAS data. So at first glance, this example
appears to violate the rule that says data selection and
manipulation logic should be as close to the data as possible.

But this view is still efficient, because searching the remote data is
not required. Instead, the KEY= option of the SET statement is
used to indicate that observations from the BOOKS.INFO data file
are to be retrieved according to the value of an indexed variable.
This allows the local SAS session to request from the SAS server
only the observation whose value is desired, and the SAS base
engine in the server's session can use its index to locate
the observation with the desired value. The entire process requires
very few communication or processing resources.

You should always very seriously consider how your application
can best put data selection and manipulation as close to the data
as possible, but at the same time it is very important to
understand the resource requirements of the algorithms available
available to you to select and manipulate the data. The best solution is the
one that minimizes resource consumption.

All Remote Data

When all the data to be joined is stored in libraries accessed by
a SAS server, it is often most efficient to have the SAS server select
and combine the data. The REMOTE engine-server architecture
and MEA makes it easy to have a SAS server execute the SQL
view engine, the DATA step view engine, and other SAS engines.
Figure 11 shows a SAS server executing a logic engine.

1521"
Joining Remote Data in a SAS Server

An example of a SAS server executing a logic engine is:

```sas
*--------------------*
* Situation: A variety of information about people, places, things in a company is stored in SAS data files that are accessed via a remote SAS server because they are updated every so often. A set of SAS views is available, and each view selects some combination of information from two or more of the SAS data files. *--------------------*

* Task: Obtain the descriptive information about the Hazardous Waste Management department, and get a list of the employees who work in that department. *--------------------*

```

```sas
/*-----------------------*/
/* This LIBNAME statement associates the libref VIEWLIB with a SAS library that is accessed via a SAS server. This SAS library contains all of the views that access the data files that contain the information of the company's people, places, and things. */
libname viewlib server=hrserver;

/*-------------------------*/
/* This PROC SQL step was used when the system was initially set up to create the HMGTA view (Hazardous Waste Management, all information) in the EMPLIB library. */
proc sql;
create view viewlib.hmgta as
    select *
    from emplib.phonelst,
        deptlib.descr
    where dept='Hazardous Waste Management';
quit;

/*-----------------------*/
/* This DATA step does some analysis that is beyond the scope of this example. Note, though, that the view HMGTA returns all of the department's information along with each individual employee's information on every observation generated by the view. For some applications this would be inefficient, but in this case it simplifies the logic of the analysis program because there is no need to access more than one data set at once. RETAIN logic is used to save values across iterations of the DATA step. */
data work.report;
    set viewlib.hmgta;
    /... other logic .../
run;

Having a logic engine used by a SAS server may increase the server's CPU consumption, but will usually decrease communications between the local SAS session and the server. This decrease is especially significant when the view produces a small result set from a large amount of data.

A programming technique that is useful in certain situations is creating a view that reads data from another view. This is known by various names, including "a view of a view" and "view stacking." When used with distributed SAS, this technique brings to light a subtlety that is safely ignored in other situations:

When the SQL view engine finds a data source specified in a view, it determines if the data source is itself an SQL view and, if so, combines the two views in an attempt to optimize the access strategy. When the two views are in the same SAS session (either both in your local SAS session or both in the server's SAS session), this happens transparently. But when the two views are in different SAS sessions, you can control whether the views are combined.

Or, to put it another way: if you read data from an SQL view in your SAS session, and that view reads data from a SAS server, and the data source in the SAS session is another SQL view, the data transmitted from the server to your SAS session can be either the result set of the view in the server's session or it can be the view itself (the instructions that say how to produce the result set).

The default value of the RMTVIEW= option of the LIBNAME statement is RMTVIEW=YES. This value causes the result set of the server's view to be sent to your SAS session. Alternatively, you can specify RMTVIEW=NO on the LIBNAME statement when you assign a libref to a library that's accessed through a SAS server. That value will cause the instructions for producing the result set (the view itself) to be sent to your SAS session, where the views will be combined and the optimized access strategy interpreted.

Accessing Remote DBMSs

SAS applications may be able to access remote DBMS data in two ways. If the DBMS system facilities support remote data access, then it's possible that a SAS/ACCESS view accessing the DBMS local functions can access a remote DBMS server. Alternatively, the SAS application can use the REMOTE engine and server architecture to execute a SAS/ACCESS engine in a SAS server to access a DBMS on the server machine.

DBMS Facilities for Remote Data Access

It's beyond the scope of this paper to describe all the capabilities of each DBMS system for remote data access. There are two basic models: direct client to remote server communication, or client to local server to remote server communication.

The first model is similar to what used by the SAS REMOTE
engine and SAS server. Client applications directly establish communications links to the SAS server without any intervening local database server. Most DBMSes implement this model and require specific client software for networked access through proprietary communications message formats and protocols to their DBMS server. The ODBC and MEA architectures allow for this by translating APIs in the drivers and engines, allowing the DBMS functions to handle communications.

The intervening server model is illustrated in figure 12.

Remote DBMS Access through Local Server

The application program uses DBMS functions to access a "local" server. The local server determines that the desired database is on a remote server machine and routes the access request to the remote machine. Note that the definition of local versus remote server may differ by environment. In a LAN environment, the term local server may include any server on the local LAN with the same basic machine architecture as the client application. The remote server is generally a server other than a different machine architecture or accessed across a WAN.

An example of this model is implemented by IBM's relational database systems. A client application on OS/2® typically accesses a local DB2/211 server which may route the request to a remote DB2 or SQL/DS® server.

Remote Data Access Through a SAS server

A SAS application can access a DBMS through a SAS server by defining the SAS/ACCESS view in a library that is accessed by the SAS server:

The combination is useful when:

- a SAS/ACCESS engine cannot be used by your local SAS session because underlying functionality is not available on your local machine, or
- you don't license the necessary software (SAS/ACCESS and/or DBMS network access software) on the local machine, or
- you want to join or merge data from the DBMS with other data on the remote machine as depicted in figure 14.

Joining Remote Data and Remote DBMS Data

An example of a SAS server executing a logic engine and accessing a DBMS is:

```c
/* Situation: The employee data base is stored in an Ingress DBMS that runs on a UNIX workstation named calvin. The registration data for a conference is stored in a SAS library that is accessed by a SAS server which is running on a UNIX workstation named hobbies. You are using a third UNIX workstation named spiff. */

Task: Report how many people are registered for the conference from each department.

/* This LIBNAME statement associates the libref VIEWLIB with a library accessed via a SAS server. This library was defined to the SAS server by the server administrator, so no physical name is needed in the LIBNAME statement. */

libname viewlib server=sasserv1;

/* The conference registration data is stored in a SAS data library that is accessed by the SAS server using libref CONFLIB. You will not need to access this library directly. */

A SAS/ACCESS view has been stored in the VIEWLIB library that retrieves the name of an employee's department from the Ingress DBMS; the name of
```
this view is EMPDEPT.

You create an SQL view that will be executed by the SQL logic engine in the SAS server’s session that will select rows from the Ingress table that have employee names that match observations in the conference registration file, group that result set into departments, and then count the number of employees in each department:

```sql
proc sql;
create view viewlib.mynamel as
   select count(empname) as deptcnt, deptname
   from conlib.register as r,
   viewlib.empdept as e
   where r.empname = e.empname
   group by deptname;
quirt;
```

You execute PROC PRINT in your local SAS session, specifying the SQL view you just created as the input SAS data set.

The SAS server will use the SQL view engine to interpret the SQL view. The SQL view engine will in turn use the SAS base engine to read the conference registration data file, and it will use the SAS/ACCESS interface to Ingress to retrieve the department name for each registered employee.

The only data transmitted from the SAS server to your SAS session is an observation for each department that has employees attending the conference. Each observation will contain the name of the department and the number of its employees attending the conference.

```sql
proc print data=viewlib.mynamel;
run;
```

By moving processing done by a logic engine closer to the data, network communication activity is reduced and you honor the rule that says to keep data selection and manipulation as close to the data as possible.

That rule becomes more and more important as data are distributed among more and more servers. If you do not organize your data and design your selection logic to be as efficient as possible, you will have more data flowing over your network than you intended, and your application will not perform as well as you intended.

**SQL Passthru**

SQL and DATA step view engines provide a very rich set of ways to select and manipulate data. In addition to those, the SQL procedure provides a way to send SQL statements directly to a SAS server or DBMS without storing them in a view. This is SQL passthru:

```sql
/*
   SQL Passthr
   SQL Processor sends statements
direct to DBMS server
   Passthr API resembles
   dynamic SQL
   SELECT statements:
   Prepared
   Described
   Rows fetched
   Other statements executed
 */

Figure 15

An example of using SQL passthru is:

```sql
/*
   To use SQL Passthru, invoke the SQL procedure.
 */

proc sql;
/*
   The CONNECT statement uses a SAS engine to make a
   connection between your SAS session and a server
   that will execute your SQL passthru statements.
   That "server" could be a SAS server or a DBMS.

   This example will consist of two parts, the first
   showing remote SQL passthru to a Sybase DBMS, and
   the second part showing remote SQL passthru to a
   SAS server.
 */

connect to sybase;
/*
   A regular SELECT statement is used to retrieve rows:
   "from connection to" specifies that the rows will be retrieved from the connection
   that was established by the CONNECT statement.

   A subquery (in parentheses) is used to specify the
   SELECT statement that is transmitted to the DBMS.
   Here, the server is told to return only those rows
   for which the value of "leaf_shape" is "Pointed".
 */

select * from connection to sybase
   (select genus, species
    from deciduous_trees
    where leaf_shape = 'Pointed');
/*
   The EXECUTE statement can be used to pass an SQL
   command (in parentheses) to the server. Here, the
   Sybase DBMS is being asked to create a view that
   retrieves rows from the "vegetables" table whose
   value of "eaten_part" is "Root".
 */

execute (create view tubers as
*/
select * from vegetables
where eaten_part = 'Root';

/*
 * The DISCONNECT statement reverses the effect
 * of the CONNECT statement.
 */

disconnect from sybase;

/*
 * You can also use the CONNECT statement to connect
 * to a remote SAS server. This CONNECT statement
 * tells your SAS session to connect to a remote SAS server. The server's
 * option tells the remote engine which SAS server
 * you want to connect to.
 */

connect to remote (server=bigiron.sasserv1);

/*
 * This SELECT statement also specifies all of the
 * selection criteria in the subquery that is sent to
 * the server. This subquery tell the SAS server to
 * return rows from the CONIFER member of the TREELIB
 * library, for which the value of the specified
 * variable SCENT is 'pungent'.
 */

select * from connection to remote
  (select genus, species
    from treeLib.conifers
    where scent = 'pungent');

/*
 * An EXECUTE statement can also be used to pass an
 * SQL command to a remote SAS server. This EXECUTE
 * statement asks the SAS server to create an SQL
 * view CITRICE in the yellowlib library that selects
 * all rows from the FRUITS member of the FOODLIB
 * library, for which the value of the variable
 * is 'citrus'.
 */

execute (create view yellowlib.citrice as
  select * from foodlib.fruits
  where type = 'citrus');

/*
 * The DISCONNECT statement reverses the effect
 * of the CONNECT statement, here disconnecting the
 * user's SAS session from the SAS server.
 */

disconnect from remote;

SQL passthru statements can be used either directly from PROC
SQL or stored in SQL views for execution by the SQL view engine
when the view is accessed as a SAS data set by other SAS
applications.

Note that the SQL statements sent to the remote server are
specified as a subquery. Since subqueries can be specified in
different SQL expressions including the WHERE and HAVING
expressions, a result set from a remote server can be used in

much more complex and dynamic queries than illustrated in these
examples. See the SAS Guide to the SQL Procedure (see the
"References" section at the end of this paper) for more information
on subqueries.

**Remote SQL Passthru**

Just as it is useful in a single machine environment to pass join
processing to the DBMS if it can do it (see DBMS data with DBMS
data above), it is best to do so in a remote server environment.
Figure 16 illustrates using the SQL procedure in the local SAS
session to send DBMS syntaxed statements through the
REMOTE engine, SAS server, and SAS/ACCESS Interface
engine in the server to the DBMS for execution.

![Diagram of SQL passthru](image1)

**Figure 16**

An example of using Remote SQL passthru is:

```sql
/*
 * Invoke the SQL procedure.
 */

proc sql;

/*
 * In remote SQL passthru, the CONNECT statement makes
two connections. The first is between your SAS
session and a SAS server, and the second is between
the SAS server and a DBMS. Your SQL passthru
statements will be executed by the DBMS. Here, the
REMOTE engine is used to connect to a SAS server
named SASSERV1.

The DBMS option tells the SAS server to use the
SAS/ACCESS Interface to RDB to connect to the
RDB DBMS.
 */

connect to remote (server=sasserv1
                     dbms=rdb);

/*
 * A regular SELECT statement is used to retrieve
 * rows from a remote server.
 */

A subquery (in parentheses) is used to specify
the SELECT statement that is transmitted to the
server. Here, the subquery combines data from two
tables. Since the SAS/ACCESS Interface will pass
the subquery on to the DBMS, the DBMS will join the

**1525**
tables and return the result set of the join to the SAS server which will return it to your SAS session.

```
select * from connection to remote
   (select dt genus, dt.species, hm.region_number
    from deciduous_trees as dt,
    habitat_map as hm
    where dt.leaf_shape = 'Pointed' &
    dt.genus = hm.genus &
    dt.species = hm.species
    );
```

The EXECUTE statement can be used to pass an SQL command (in parentheses) to a SAS server that in turn uses the SAS/ACCESS interface to pass the SQL command on to the DBMS. Here, the DBMS is being asked to create a view that retrieves rows from the "vegetables" table whose value of "eaten_part" is "Root".

```
execute (create view tubers as
   select * from vegetables
   where eaten_part = 'Root');
```

The DISCONNECT statement reverses the effect of the CONNECT statement.

```
disconnect from remote;
```

**Update Access and User Identification**

A multi-user server appears to a DBMS as a single "user," so DBMS authorization is based on the access rights of the SAS server. You can still provide security for the DBMS in this environment by limiting access to the server with server access password and/or utilizing passwords on SAS/ACCESS views in the server.

Update access to a DBMS via a multi-user SAS server is not allowed because the DBMSes do not treat each user of the SAS server as a separate transaction environment that can be independently committed or rolled back.

However, remote update access to a DBMS is allowed via the Remote Library Services feature of SAS CONNECT software. Remote Library Services includes a component that provides the data access functionality of the multi-user SAS server, but only within a remote SAS/CONNECT session and only to the local SAS session that created the remote SAS session. Since the remote SAS/CONNECT session runs with the authorization of the user that created it and since only the local SAS/CONNECT session can communicate with the remote SAS/CONNECT session, DBMS authorization controls are effective when Remote Library Services is used.

For a thorough treatment of SAS server security issues, please see the paper Security for SAS Servers, presented at this conference by Brian Parkinson.

**CONCLUSION**

MEA is Extremely Flexible

The modular structure of Multiple Engine Architecture allows you to accomplish tasks in many ways by various combinations of the engines. Figure 17 combines most of the previous SAS figures into one. I hope you can reference this diagram and paper to effectively combine the facilities available today in the SAS System.

Figure 17

**SAS Server and MEA extend capabilities of ODBC Applications**

While the primary motivation of the SAS/ODBC Driver implementation is to allow OPEN access of SAS formatted and managed data from non-SAS applications such as some of our users have requested, it's important to remember that Multiple Engine Architecture in a server environment allows you to join and merge data from many sources in many ways. Figure 18 illustrates all the additional capabilities from an ODBC application environment.

Figure 18

From the SQL logic engine you can:

- access SAS data files through the native engine,
Systems Architecture

- access external files through DATA step processor with a DATA step view,
- access DBMS data through SAS/ACCESS views,
- access DBMS data through SAS SQL views containing SQL pass thru statements for the DBMS, and
- join and merge data from these sources through SQL statements or DATA step programming logic.

The architecture will also allow an ODBC application to send SQL statements through the server and a SAS/ACCESS pass thru Interface to a DBMS without using the SQL processor in the server execution as depicted by the line from the user task directly to the SAS/ACCESS pass thru API. However, the SAS/ODBC Driver does not provide the same level of ODBC conformance when by-passing the SAS SQL processor. We've found many ODBC application packages need level 1 conformance. Therefore, we have not expended resources testing this capability. You may find it adequate for writing custom ODBC applications for known databases and tables. If you are interested in trying this, contact Technical Support at SAS Institute.

Future Plans

We plan to support more standard APIs for open data access to SAS data as they are adopted by the software community and requested by our customers. Also, we are in the process of providing a proprietary interface that is callable from C programs in environments where no standard is emerging. If you have an interest in a particular standard and operating system - please convey your interest to both the author and the SAS Marketing division at SAS Institute.

We also intend to study ways to improve our processing and so that we can make more intelligent processing decisions for you. For example:

1. We want to be able to retransform and optimize SQL access strategies when a new WHERE clause is given. However, we don't want to do that if we've already accessed all the data and have a relatively small set of results sets that could be fitted with the new WHERE clause. Different deciles are appropriate for different cases and may produce different result sets in a concurrent access environment.

2. We are considering adding another key type to the SAS data model in version 7 that could be used for equality matching only. Some SAS/ACCESS engines will then identify DBMS indexes as these restricted use keys. The SQL processor will then use them and the engine will convert these restricted key interface to the appropriate WHERE clause. Again, there are cases where this is more efficient and others where it is less efficient. Some of the criteria will depend on the DBMS.

Ultimately, we need to give the programmer the ability to force one decision or another even if we improve our ability to optimize the default behavior.

REFERENCES


ACKNOWLEDGEMENTS

SAS, SAS/ACCESS, SAS/SHARE, and SAS/CONNECT are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. © indicates USA registration.

IBM, DB2, DB2/2, OS/2, and SQL/DS are registered trademarks or trademarks of International Business Machines Corporation.

Other brand and product names are registered trademarks or trademarks of their respective companies.

1527