SAS/ACCESS® 9.1.x Interface to DB2 on the Mainframe Using the LIBNAME Engine

The Power to Know®

A SAS White Paper

Table of Contents

Introduction

This paper provides an overview of the most important features and capabilities of the SAS/ACCESS Interface to DB2. General and technical product information is given for SAS/ACCESS and DB2 RDBMS users who need fast, seamless access to DB2 tables from the SAS System. The methods for accessing DB2 data are presented, along with examples.

Throughout this paper, tips are provided that enable you to optimize a SAS session or job so that the processing is performed by SAS or by DB2, whichever would be most efficient for the job. The objective is to achieve SAS centered processing where the SAS software *manages, but not necessarily performs, the work*. SAS centered processing, given good performance, is preferable to separating the work and performing some tasks in SAS and the rest of the tasks outside SAS; for example, pre-processing or post-processing with DB2 utilities.

The content of this paper ranges from rudimentary to complex. Because sections that are dense with detail can be time-consuming to read, this paper is formatted with main ideas presented as topics. Within a topic, you'll find a technical discussion that often includes a code example. All the examples that are shown in this document were run using SAS 9.1 or higher.

SAS/ACCESS Interface to DB2

Problem: Your site uses DB2 as a data repository. Your DB2 users want to mine, warehouse, and analyze the DBMS data by using SAS software, thereby taking advantage of the broad range of SAS tools: from basic data exploitation — reports and graphs — to the latest technologies.

Solution: The SAS/ACCESS Interface to DB2 enables SAS users to transparently access and manipulate DB2 data. SAS/ACCESS doesn't replace the database functionality but enhances it by adding the power of the SAS language and functions.

What's New in SAS/ACCESS 9.1.x?

The following options and features are new:

- The DEGREE option in the LIBNAME statement determines whether DB2 can use parallelism.
- Multi-volume support for both SMS and non-SMS managed data sets when using the bulk loader via the BL_DB2UNITCOUNT option.
- Threaded reads, which are controlled by the following new options:
  - DBSLICEPARM – enables or disables threaded reads, controls their scope, and controls the number of SAS threads to be generated.

    - DBSLICE – overrides the DBSLICEPARM option when you specify WHERE clauses to partition a table across multiple SAS threads. This option is especially useful when you are familiar with the table content and structure.

- Full support of stored procedures, including the ability to pass and retrieve parameters by using SAS macro variables.
- Support for GLOBAL TEMPORARY tables. Starting with SAS 9.1.2, the DBMSTEMP=YES option in the LIBNAME statement creates DECLARE GLOBAL TEMPORARY tables by using implicit pass-through.
- Starting in SAS 9.1.3, SQL updates use parameter markers that remove the limitation of not being able to update columns with literals when the length of a literal exceeds 255 bytes.

For complete information about the syntax and use of these new options, see the SAS documentation.

Which Releases of DB2 Are Supported?

SAS/ACCESS Interface to DB2 for SAS®9 and higher supports DB2, Version 6, Release 1 and higher.

What SAS Software Is Required to Access DB2?

Base SAS and SAS/ACCESS Interface to DB2 are required in order to access DB2.

Where Should the Required SAS Software Be Installed?

To access DB2, you must install the required SAS software on the same machine on which the DB2 software is installed.

How Does SAS/ACCESS Communicate with the Database?

The SAS/ACCESS Interface to DB2 uses two types of call attachment facilities to connect to DB2. An attachment facility is a part of the DB2 code that allows other programs to connect to and use DB2 to process SQL statements and commands. With the **Call Attachment Facility** (CAF), your application program can establish and control its own connection to DB2. Programs that run in MVS batch, TSO foreground, and TSO background can use CAF. By default, the SAS/ACCESS Interface to DB2 uses CAF. SAS supports multiple CAF connections for a SAS session, therefore, for a SAS server, each SAS client can have its own connection to DB2. However, because CAF does not support sign on, each connection that the SAS server makes to DB2 has the operating environment authorization ID of the server, not the authorization ID of the client for which the connection is made.

If you specify the DB2RRS system option, the SAS/ACCESS interface to DB2 engine uses the **Recoverable Resource Manager Services Attachment Facility** (RRSAF). RRSAF was a new feature in DB2 Version 5, Release 1, and support for it began in SAS 8.

Because only one attachment facility can be used at a time, the DB2RRS system option can be specified only when SAS is invoked. SAS supports multiple RRSAF connections for a SAS session, and RRSAF supports the ability to associate a mainframe authorization ID with each connection at sign on. This authorization ID is not the same as the authorization ID that is specified in the AUTHID data set or in the LIBNAME option. DB2 uses the RRSAF-supported authorization ID to validate a given connection's authority to use both DB2 and system resources, when those connections are made using the System Authorization Facility and other security products such as RACF. Basically, this authorization ID is the user ID that you used for logging on to the host system. With RRSAF, the SAS server makes the connections for each client, and the connections have the client's host authorization ID associated with them. This is true only for clients that are authenticated by the SAS server when the client specifies a user ID and a password. (Usually, servers authenticate their clients when the clients provide their user IDs and passwords.) If a client connects to a SAS server without providing a user ID and a password, then the ID that is associated with the connections is the ID of the server (as with CAF) and not the ID of the client.

Other than specifying DB2RRS at SAS start-up, there is nothing else that needs to be done in order to use RSSAF. The SAS/ACCESS Interface to DB2 automatically signs on each connection that is made to DB2 with either the ID of the authenticated client or the ID of the SAS server for non-authenticated clients. The authenticated clients have the same authorities to DB2 as they have when they run their own SAS session from their own ID.

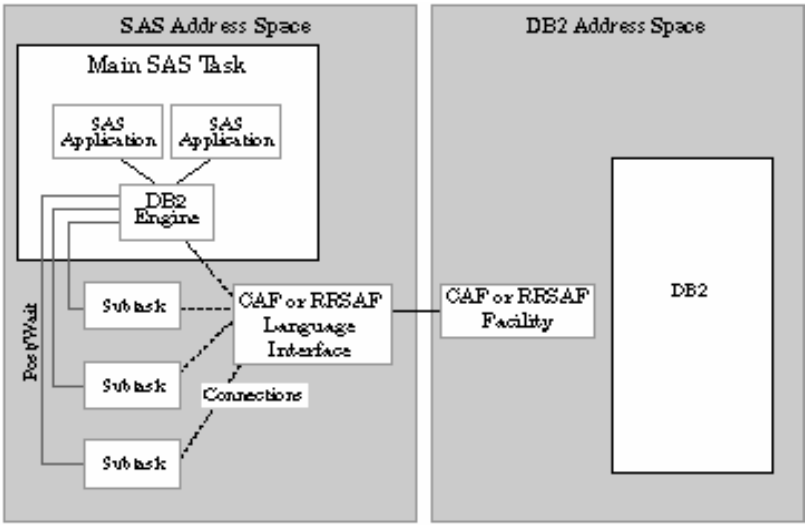


Figure 1. How SAS/ACCESS Communicates with DB2

For more information about CAF and RRSAF, see the DB2 documentation from IBM.

Can SAS/ACCESS Get Data at Remote Locations?

The **Distributed Data Facility** (DDF) is a DB2 component that enables DB2 applications to access data at other DB2 servers and at remote relational database systems that are DRDA compliant. The SAS/ACCESS Interface to DB2 supports both types of distributed database connections, **system-directed** (through private protocol) and **application-directed** (through DRDA).

System-Directed Access

System-directed access enables a DB2 requester to connect to one or more DB2 servers. This type of connection is established by coding three-part names or database aliases in the application. The connection that is established between the DB2 requester and the server does not adhere to the DRDA standards, and cannot be used to connect products that are not DB2 products to DB2. The SAS/ACCESS Interface to DB2 cannot explicitly request a connection. Instead, it performs an implicit connection when a distributed request is initiated by the SAS application. To initiate an implicit connection, you must specify the LOCATION option. When you specify this option, the three-level table name (*location.authid.table*) is used in the SQL statement that is generated by the interface to DB2. When the SQL statement that contains the three-level table name is executed, an implicit connection is made to the remote DB2 subsystem. The primary authorization ID of the initiating process must be validated in order to connect to the remote location.

Application-Directed Access

Application-directed access enables a DB2 requester to connect to one or more DB2 or non-DB2/mainframe application servers such as DB2 Universal Database for AIX and DB2 Universal Database for iSeries (AS/400) using DRDA protocols. This type of connection is established by coding SQL CONNECT statements in the application. To connect to a DRDA remote server or location, the SAS/ACCESS Interface to DB2 uses an explicit connection. To establish an explicit connection, the interface to DB2 first connects to the local DB2 subsystem via an attachment facility (CAF or RRSAF). Then it issues an SQL CONNECT statement to connect from the local DB2 subsystem to the remote DRDA server prior to accessing data. To initiate a connection to a DRDA remote server, you must specify the connection option SERVER. The SAS application uses a separate connection (specifying the SERVER option) for each remote DRDA location.

What Is DRDA?

DRDA is an abbreviation for Distributed Relational Database Architecture. DRDA distributes relational data among multiple platforms and enables similar and dissimilar platforms to communicate with one another. For example, a DB2 subsystem can communicate with another DB2 subsystem (similar). Alternatively, a DB2 subsystem can communicate with a third-party RDBMS (dissimilar). As long as both platforms conform to the DRDA specifications, they can communicate. DRDA can be considered to be a "universal, distributed data protocol."

DRDA provides methods of coordinating communication among distributed locations. This capability enables applications to access multiple remote tables at various locations and have them appear to the end user as if they were a logical whole.

However, a distinction should be made between the architecture and the implementation. DRDA describes the architecture for distributed data and nothing more. DRDA defines the rules for accessing the distributed data, but it does not provide the actual application programming interfaces (APIs) to perform the access. So DRDA is not an actual program but is more like the specifications for a program.

When a DBMS is said to be DRDA compliant, it means that the DBMS follows the DRDA standards. DB2 is a DRDA-compliant RDBMS product.

DRDA versus Private Protocol

It is recommended that the DRDA protocol be used to access distributed data. IBM has announced that private protocol will not be enhanced in the future, and it might not be supported for future releases of the database.

Here are some other factors relating to private protocol:

- Private protocol does not support many SQL features, including user-defined functions, LOBs, and stored procedures.
- Only DB2/mainframe subsystems can communicate using this protocol.
- When a static SQL statement is passed to the server, it is dynamically bound and then executed. The statement is dynamically bound only the first time that it is executed in a unit of recovery. Subsequent executions of the SQL statement in the unit of recovery do not pay the cost of the dynamic bind. However, if the SQL statement is executed again after a COMMIT or a ROLLBACK statement, another dynamic bind occurs, unless dynamic SQL is being cached either globally or locally.
- In a unit of work, updates can be made to an unlimited number of DB2 subsystems. An application can also read at several sites in a unit of work.

Distributed processing using DRDA has the following characteristics:

- The application can access data at any server that supports DRDA. It is not limited to accessing data at a DB2 server on the mainframe.
- The application can use remote BIND to bind SQL into packages at the serving RDBMS.
- The application can connect to other RDBMSs in the network and execute packages at those DBMSs.

How SAS/ACCESS Works – Methods to Access DB2

SAS/ACCESS software for relational databases is a family of interfaces (each interface is licensed separately) that enable you to interact from SAS with data that is stored in RDBMSs. SAS/ACCESS provides the following methods for accessing relational DBMS data:

- The LIBNAME statement, which enables you to assign SAS library references (librefs) to database schemas, local or remote. After a database schema is associated with a libref,

two-level names can be used to access any table or view in the schema and work with it as if it were a SAS data set.

- The MODIFY and UPDATE statements in a DATA step.
- The Pass-Through Facility, which enables you to interact with the database by using its SQL syntax without leaving your SAS session. The SQL statements are passed directly to the data source for processing.

SAS/ACCESS can use either implicit or explicit pass-through. The former uses the LIBNAME engine to generate and run SQL statements against the database.

- The ACCESS, DBLOAD, and DBUTIL procedures, which support indirect access to database objects. These are legacy procedures that continue to be supported for the database interfaces and environments on which they were available in releases of SAS 6. The ACCESS procedure does not support SAS names that are longer than 8 characters or database fields that are longer than 200 characters.

Selecting a SAS/ACCESS Method

There are several methods that you can use in order to access DBMS data. The advantages and limitations of each method are described in the next sections. Before processing complex or data-intensive operations, you might want to test several of these methods in order to determine which one is the most efficient for your specific task.

In general, using the LIBNAME statement in SAS/ACCESS is usually the fastest and most direct method of accessing your DBMS data, and the only method that supports bulk loading. An exception to this is when you need to use non-ANSI standard SQL. ANSI standard SQL is generated when you use the SAS/ACCESS LIBNAME engine in PROC SQL, but the Pass-Through Facility, using explicit pass-through, accepts all the extensions to SQL that are provided by your DBMS.

Using the LIBNAME statement in SAS/ACCESS has the following advantages:

- Significantly fewer lines of SAS code are required in order to perform operations on your DBMS. For example, a single LIBNAME statement establishes a connection to the database for easy table access.
- You do not need to know SQL in order to access and manipulate data on your DBMS. You can use SAS procedures or DATA step programming on any libref that references DBMS data. You can read, insert, update, delete, and append data, as well as create and drop DBMS tables by using SAS statements.
- The LIBNAME statement provides more control over database operations such as locking, spooling, and data type conversion by using LIBNAME and data set options.
- The LIBNAME engine can optimize the processing of joins and WHERE clauses by passing these operations directly to the DBMS. This takes advantage of database indexing and other processing capabilities.
- The LIBNAME engine can pass some functions directly to the DBMS for processing.

Using the Pass-Through Facility has the following advantages:

- Pass-Through Facility statements enable the DBMS to optimize queries, especially when you join tables.
- Pass-Through Facility statements enable the DBMS to optimize queries when they have summary functions (such as AVG and COUNT), GROUP BY clauses, or columns that are created by expressions (such as the COMPUTED function).
- Pass-Through Facility statements can be used with SAS/AF[1] applications to handle the transaction processing of the DBMS data. Using a SAS/AF application gives a user complete control of COMMIT and ROLLBACK transactions. The Pass-Through Facility accepts all the extensions to SQL that are provided by your DBMS.

How the LIBNAME Statement Works in SAS/ACCESS

SAS/ACCESS enables you to read, update, insert, and delete data from a DBMS object as if it were a SAS data set. You invoke a SAS/ACCESS interface by specifying a DBMS engine name and the appropriate connection options in a LIBNAME statement. Then, enter SAS requests as you would when accessing a SAS data set. SAS/ACCESS generates DBMS-specific SQL statements for the SAS requests that you entered, and submits the generated SQL to the DBMS.

Example 1 Browse a table by using the LIBNAME statement.

```
libname db2lib db2 ssid=db2a
    authid=dsn8710;
proc print data=db2lib.emp;
run;
```

Example 2 Add data to a table by using the LIBNAME statement in conjunction with PROC APPEND.

```
libname db2lib db2 ssid=db2a
    authid=dsn8710;

proc append base=db2lib.manager;
    data=db2lib.emp;
  where job = 'MANAGER';
run;
```

(See the *Base SAS Procedures Guide* for details about the APPEND procedure.)

Example 3 Create and populate a table from an existing table by using the LIBNAME statement.

```
libname db2lib db2 ssid=db2a
    authid=dsn8710;

proc sql;
    create table db2lib.managers
    as select * from db2lib.emp
      where job = 'MANAGER';
quit;
```

[1] SAS/AF set of tools that develop applications in the SAS System.

How the Pass-Through Facility Works

When you read and update DBMS data by using the Pass-Through Facility, SAS/ACCESS passes SQL statements directly to the DBMS for processing. To enhance performance, SAS/ACCESS can pass queries, joins, and data functions to the DBMS for processing (instead of retrieving the data from the DBMS and processing it in SAS).

Here is how the Pass-Through Facility works:

1. The Pass-Through Facility invokes PROC SQL and submits a CONNECT statement that can include the DBMS name (the default is DB2) and the options that are required in order to establish a connection with a specific database.
2. PROC SQL examines each query to determine whether it would be advantageous to send all or part of the query to the DBMS for processing.
3. PROC SQL translates queries (or query fragments) into DBMS-specific SQL syntax.
4. The queries (or query fragments) are processed by the DBMS, which returns the results to SAS. Any queries or query fragments that cannot be passed to the DBMS are processed in SAS.

In the following example, the CONNECTION TO option is used to read data from a DBMS table or view. Notice that the SQL statement is stored in a SAS view for later processing (optional). The SELECT statement in parenthesis uses SQL that is native to DB2. SAS/ACCESS passes the SQL statements directly to the DBMS for processing. If the SQL syntax that you enter is correct, the DBMS processes the statement and returns the result to SAS.

The syntax of a SQL statement is not checked immediately when the statement is used to create a SAS view. The syntax will be checked at run- time.

```
proc sql;
   connect to db2 (ssid=db2a);
     create view managers as
     select *
       from connection to db2
    (select select empno, firstnme,
           lastname, workdept,
           salary
       from dsn8710.emp
       where job = 'MANAGER');
 quit;

proc print data=managers;
run;
```

Alternatively, the Pass-Through Facility uses PROC SQL to pass SQL statements other than SELECT statements (such as INSERT, DELETE, and UPDATE) to the database. As with the CONNECTION TO option, all EXECUTE statements are passed to the DBMS exactly as you enter them. INSERT statements must contain literal values. Terminate the connection with the DISCONNECT statement. For example:

```
proc sql;
  connect to db2 (ssid=db2a);
  execute (
  create view managers as
  select select empno, firstnme,
    lastname, workdept,
    salary
   from dsn8710.emp
   where job = 'MANAGER') by db2;

  execute (
  grant select on managers to
    public) by db2;
  disconnect from db2;
quit;
```

Notice that, in the first example, the CREATE VIEW resulted in the creation of a SAS PROC SQL view, but in the latter example, the view is actually created in DB2.

SQL Pass-Through Method	User action in SAS/ACCESS Interface to DB2	PROC SQL …	DBMS response…
EXPLICIT	Uses PROC SQL to specify SQL that is specific to DB2	Passes the SQL exactly as written to DB2	Performs the SQL request if the syntax is correct, otherwise, the request fails
IMPLICIT	Uses PROC SQL to specify SAS SQL	Converts SAS SQL to DB2-specific SQL on your behalf, and passes the SQL to the DBMS; however, it executes the SQL portions of the queries, joins, etc. that cannot be converted in SAS	Performs the SQL request if the functionality is supported, otherwise, it returns an error condition that triggers SAS processing

Table 1. Explicit vs. Implicit SQL Pass-Through

Explicit Pass-Through Method – No LIBNAME statement

Using this method, you invoke the SQL procedure and provide parameters to connect to the DB2 server, then enter the SQL requests. PROC SQL submits your SQL statements to DB2 exactly as specified. Assuming that your SQL syntax is correct, DB2 performs the processing that's requested.

Implicit Pass-Through Method – LIBNAME statement

Using a LIBNAME statement, PROC SQL can generate SQL statements on your behalf. This behind-the-scenes processing is known as *implicit pass-through*. SAS, via PROC SQL, passes as much work as possible to the underlying DBMS (in this instance, to DB2). However, implicit pass-through is subject to the rules that are discussed in the next section "Understanding the Rules of Implicit Pass-Through". Additional examples of explicit and implicit pass-through are given later in this section.

Understanding the Rules of Implicit Pass-Through

Basic Eligibility – To be eligible for implicit pass-through, a query must meet the following requirements:

- refer to a single LIBNAME statement in SAS/ACCESS (versions prior to SAS®9)
- be legal according to the ANSI SQL2 standard
- be recognized by PROC SQL

Effect of DBMS SQL2 Conformance – The DBMS can reject a valid SQL2 query that is passed by SAS because the DBMS might not support all three levels of SQL2 conformance. Describing a legal SQL2 query for DB2 is beyond the scope of this paper, however, SQL keywords are listed that trigger PROC SQL to pass the query (or query fragment) to DB2. Also listed are the elements within the query that cause PROC SQL to disqualify it.

The following SQL keywords trigger PROC SQL to pass the query to DB2:

- DISTINCT
- Aggregate functions:
    - COUNT(*)
    - COUNT(x)
    - FREQ(x)
    - N(x)
    - AVG(x)
    - MEAN(x)
    - MAX(x)
    - MIN(x)
    - SUM(x)
    - JOIN
- UNION

PROC SQL disqualifies any query that includes one or more of the following elements:

- CONNECTION TO
- Re-merging
- DATA step options
- One or more truncated comparisons
- INTO clause
- One or more ANSI MISS/NOMISS inner or outer joins
- A SAS function that is not in the preceding list of aggregate functions or in the SAS data function list (see the next section "Data Functions That Are Passed to DB2".

Therefore, PROC SQL does not pass a query to DB2 that contains a WHERE clause with an unsupported SAS function, for example, the FUZZ[2] function. Instead, PROC SQL returns the query to SAS for processing.

Data Functions That Are Passed to DB2

Beginning with SAS/ACCESS 8.2, the following SAS SQL functions are passed to DB2 for processing. If the DB2 function name is different from the SAS function name, the DB2 name is shown in parenthesis:

ABS	LOWCASE (LCASE)	TAN
ARCOS (ACOS)	LOG	TANH
ARSIN (ASIN)	LOG10	UPCASE (UCASE)
ATAN	MOD	SUM
CEIL	SIGN	COUNT
COS	SIN	AVG
COSH	SINH	MIN
EXP	SQRT	MAX
FLOOR	YEAR	MONTH
DAY		

Table 2. SAS Data Functions That Are Passed to DB2

Some of these functions didn't exist in DB2 prior to DB2 Version 6, so they are not passed to the DBMS in DB2 Version 5. Also, these functions are not passed to the DBMS when you connect by using DRDA, because there is no way to determine what location that you are connected to and which functions are supported there.

Example 1 Implicit Pass-Through of the SAS YEAR() Function

```
libname db2lib db2 ssid=db2a
        authid=dsn8710;
proc print data=db2lib.emp
     (keep=empno, firstnme, lastname,
           workdept, salary, hiredate);
     where year(hiredate)>=1980;
run;
```

The SAS YEAR() function is equivalent to the DB2 YEAR() function. Therefore, implicit pass-through generates the following DB2-specific SQL for the preceding SAS WHERE-clause code:

```
SELECT "EMPNO","FIRSTNME","LASTNAME"
       "WORKDEPT","SALARY","HIREDATE"
  FROM DSN8710.EMP
 WHERE (YEAR("HIREDATE") >= 1980)
   FOR FETCH ONLY
```

[2] The FUZZ function returns the nearest integer if the argument is in 1E-12.

Example 2 Explicit and Implicit Pass-Through: Side-by-Side Code

```
/* Setup: Create a small DB2 sample table */

libname db2lib db2 ssid=db2a;

data db2lib.customers;
     input custname $  1-10
           custnum    18-20
           custcity $ 22-36;

datalines;
Beach Land    16     Ocean City
Coast Shop     3     Myrtle Beach
Coast Shop     5     Myrtle Beach
Coast Shop    12     Virginia Beach
Coast Shop    14     Charleston
Del Mar        3     Folly Beach
Del Mar        8     Charleston
Del Mar       11     Charleston
New Waves      3     Ocean City
New Waves      6     Virginia Beach
Sea Sports     8     Charleston
Sea Sports    20     Virginia Beach
Surf Mart    101     Charleston
Surf Mart    118     Surfside
Surf Mart    127     Ocean Isle
Surf Mart    133     Charleston
run;
```

Explicit SQL	Implicit SQL

`option db2dbug nodate sastrace=',,,d';`
`title2 'Customer Cities';`
 	
`proc sql noerrorstop;`
`connect to db2 (ssid=db2a);`
`  select * from connection to db2`
`  (select distinct custcity from customers);`
`quit;`
	`option db2dbug nodate`
	`sastrace=',,,d';`
	`title2 'Customer Cities';`
	`libname db2lib db2 ssid=db2a;`
	`proc sql noerrorstop;`
	`  select distinct custcity from`
	`db2lib.customers;`
	`quit;`

continued

Table 3. Side-by-Side SQL Pass-Through Examples

SAS Log

```
1   options db2dbug nodate sastrace=',,,d';
2   title2 'Customer Cities';
3   proc sql noerrorstop;
4      connect to db2 (ssid=db2a);
5      select * from connection to db2
6      (select distinct custcity from
customers);

DB2_7: Prepared:
select distinct custcity from customers

DB2_8: Executed:
select distinct custcity from customers

7   quit;
NOTE: The PROCEDURE SQL used 0.08 CPU
seconds and 16901K.

NOTE: The address space has used a maximum
of 1224K below the line and 19848K above the
line.
```

```
1   options db2dbug nodate
sastrace=',,,d';
TRACE: Entering d2init
TRACE: Exiting d2init
2   title2 'Customer Cities';
3   libname db2lib db2 ssid=db2a;
NOTE: Libref DB2LIB was successfully
assigned as follows:
      Engine:        DB2
      Physical Name: DB2A
4   proc sql noerrorstop;
5      select distinct custcity from
        db2lib.customers;

DB2_1: Prepared:
SELECT * FROM CUSTOMERS FOR FETCH ONLY

DB2_2: Prepared:
select distinct customers."CUSTCITY" from
CUSTOMERS FOR FETCH ONLY

DB2_3: Executed:
select distinct customers."CUSTCITY" from
CUSTOMERS FOR FETCH ONLY

ACCESS ENGINE:  SQL statement was passed
to the DBMS for fetching data.
COMMIT WORK
DB2_4: Executed:
COMMIT WORK
6   quit;
NOTE: The PROCEDURE SQL used 0.07 CPU
seconds and 15721K.

NOTE: The address space has used a maximum
of 1224K below the line and 20344K above
the line.
```

SAS Output

```
The SAS System
Customer Cities
Custcity
---------------
Charleston
Folly Beach
Myrtle Beach
Ocean City
Ocean Isle
Surfside
Virginia Beach
Wilmington
```

```
The SAS System
Customer Cities
Custcity
---------------
Charleston
Folly Beach
Myrtle Beach
Ocean City
Ocean Isle
Surfside
Virginia Beach
Wilmington
```

*This statement is executed to determine whether the table exists.

Table 3 (continued). Side-by-Side SQL Pass-Through Examples

In the preceding examples, look at the SQL that is marked in bold. Notice that, with explicit pass-through, the SQL that is submitted to DB2 is exactly as specified; with implicit pass-through, the SAS/ACCESS Interface to DB2 generates the SQL that is submitted on your behalf. (In this example, the SQL submitted to DB2 is essentially the same.)

Loading Data into the DBMS

DB2 Bulk Loading

By default, the SAS/ACCESS Interface to DB2 loads data into tables by executing a SQL INSERT statement for each row. If you specify BULKLOAD=YES, the DB2 LOAD utility is invoked. This enables you to bulk load the rows of data as a single unit, which can significantly enhance performance. For small tables, the extra overhead of the bulk-load process might slow performance. For large tables, the speed of the bulk-load process outweighs the overhead costs.

Note: When using the bulk loader, always look at the SYSPRINT output for information about the load.

In the SAS/ACCESS Interface to DB2, the bulk loading capability is provided via DSNUTILS, which is an IBM stored procedure that invokes the DB2 LOAD utility. DSNUTILS is included in DB2 Version 6 and later and is available in DB2 Version 5 via a maintenance release. Because the LOAD utility is complex, familiarize yourself with it before you use it through SAS/ACCESS. Ask your database administrator whether this utility is available.

The DB2 LOAD utility does not create tables; it loads data into existing tables. SAS/ACCESS creates a table before loading data into it (whether you use SQL INSERT statements or invoke the LOAD utility). You might want to invoke the LOAD utility for an existing table. In this case, specify BL_DB2TBLXST=YES to tell the engine that the table already exists. If the table does not yet exist and you do not want to create and load it, you can use BL_DB2TBLXST in conjunction with BL_DB2LDEXT=GENONLY. In this case, the control and data files are generated, but the table is not created.

The column types of the data that will be loaded into an existing table must match the column types of that table. SAS does not verify the input data against the table definition. Any incompatibilities are flagged by the LOAD utility.

BL_DB2LDCT1, BL_DB2LDCT2, and BL_DB2LDCT3 Options

The options BL_DB2LDCT1 and BL_DB2LDCT2 enable you to pass parameters to the load process. Using these options, you can collect statistics about the table that is being loaded, compress its content, load data using a different character set, or avoid putting the table in copy-pending status when LOG NO is also specified.

The parameters that are specified by using the option BL_DB2LDCT1 are added to the LOAD command after the keyword LOAD and before the clause INTO TABLE. The parameters that are specified by using the option BL_DB2LDCT2 are added to the LOAD command between the table name and the column list.

The parameters that can be passed depend on which release of DB2 is running. See the DB2 documentation from IBM for the list of options that can be passed by using BL_DB2LDCT1 and BL_DB2LDCT2.

The option BL_DB2LDCT3 enables you to pass extra options that are added at the end of the control file. These options are not related to the load process, but they can be used to call other DB2 utilities, such as the REPAIR utility, to re-set the tablespace status when LOG NO is used and NOCOPYPEND is not specified. See the examples at the end of this section that show how this option can be used.

SMS Support

The SAS/ACCESS Interface to DB2 in SAS®9 enables SMS parameters to be specified for the SYSIN, SYSREC, and SYSPRINT data sets that are allocated by the bulk loader. Data, management and storage classes can be passed by using the options BL_DB2DATACLAS, BL_DB2MGMTCLAS, and BL_DB2STORCLAS.

Multi-Volume Data Set Support

Support for multi-volume data sets is provided by the option BL_DB2UNITCOUNT. This option applies only to the SYSREC data set, which is the file that has to be loaded into the DB2 table.

The value that is specified for the option BL_DB2UNITCOUNT must be an integer between 1 and 59; if the value is greater than 59, this option is ignored. However, the value depends on the unit name that is provided by using the option BL_DB2DEVT_PERM, which, by default, is SYSDA. An association exists at the operating environment level that defines the maximum number of volumes for a unit name. Ask your storage administrator for this number.

If the value that is specified for the option BL_DB2UNITCOUNT exceeds the maximum number of volumes for the unit, an error is returned. For SMS-managed data sets, the data class determines whether they can extend on multiple volumes. When the options BL_DB2DATACLAS and BL_DB2UNITCOUNT are both specified, the latter overrides the unit count values for the data class.

BL_DB2DATACLAS Option

Use the option BL_DB2DATACLAS to specify a data class for a new SMS-managed data set. If you specify this option for a data set that SMS does not support, SMS ignores it. If SMS is not installed or is not active, the operating environment ignores any data class that is passed by using BL_DB2DATACLAS. The storage administrator at your site defines the names of the data classes that can be specified when using BL_DB2DATACLAS.

The option BL_DB2DATACLAS applies to the control file (BL_DB2IN), the input file (BL_DB2REC), and the output file (BL_DB2PRINT) for the bulk loader.

BL_DB2MGMTCLAS Option

Use the option BL_DB2MGMTCLAS to specify a management class for a new SMS-managed data set.. If SMS is not installed or is not active, the operating environment ignores any management class that is passed by using BL_DB2MGMTCLAS. The storage administrator at

your site defines the names of the management classes that can be specified when using BL_DB2MGMTCLAS.

The option BL_DB2MGMTCLAS applies to the control file (BL_DB2IN), the input file (BL_DB2REC), and the output file (BL_DB2PRINT) for the bulk loader.

BL_DB2STORCLAS Option

Use the option BL_DB2STORCLAS to specify a storage class for a new SMS-managed data set. If SMS is not installed or is not active, the operating environment ignores any storage class that is passed by using BL_DB2STORCLAS. A storage class contains the attributes that identify a storage service level that SMS uses for storage of the data set. It replaces any storage attributes that are specified by using the option BL_DB2DEVT_PERM.The storage administrator at your site defines the names of the storage classes that can be specified when using BL_DB2STORCLAS.

The option BL_DB2STORCLAS applies to the control file (BL_DB2IN), the input file (BL_DB2REC), and the output file (BL_DB2PRINT) for the bulk loader.

Re-Starting the Bulk Loader

In case of failure, you can re-start a bulk-load operation. Re-start functionality is controlled by the option BL_DB2RSTRT, which tells the LOAD utility whether the current load is a re-start and, for a re-start, indicates where to begin. When you specify a value other than NO for the option BL_DB2RSTRT, you must also specify BL_DB2TBLXST=YES and BL_DB2LDEXT=USERUN. Valid values for the option BL_DB2RSTRT are:

NO – specifies a new invocation of the LOAD utility, not a re-start. This is the default.

CURRENT – specfies to re-start at the last commit point.

PHASE – specifies to re-start at the beginning of the current phase.

How do these options interact with DB2? To answer this question, consider how the DB2 load utility works. The utility has 10 phases: UTILINIT, RELOAD, ENFORCE, DISCARD, SORT, INDEXVAL, REPORT, BUILD, SORTBLD, and UTILTERM. The ability to re-start a load operation is determined as follows:

- If a failure occurs during the UTILINIT phase, re-start the utility from the beginning.
- If a failure occurs during the RELOAD, ENFORCE, or DISCARD phase, re-start the utility using BL_DB2RSTRT=CURRENT.
- If a failure occurs during the SORT, INDEXVAL, or REPORT phase, re-start the utility using BL_DB2RSTRT=PHASE.
- If a failure occurs during the BUILD phase, re-start the utility using BL_DB2RSTRT=PHASE if the option REPLACE was also used. If the load was appending data, the utility is not re-startable; it must be terminated and the indexes must be rebuilt.
- If the utility fails during the UTILTERM phase, there is usually no need to re-start it.

Note: When the load option SORTKEYS is used and the utility fails during the RELOAD, SORT, or BUILD phase, the value CURRENT or PHASE for the option BL_DB2RSTRT re-starts the utility from the beginning of the RELOAD phase.

For a complete description of all the phases of the DB2 load utility and their ability to be re-started, see the DB2 documentation from IBM.

SQL Inserts vs. Bulk Load

A database table can be loaded either by using SQL INSERT statements or by using the bulk loader. Which method is best? The answer is…"it depends". It depends on:

- the amount of data to be loaded
- whether the table has indexes and how many indexes there are
- whether the table has referential constraints
- whether the table is empty and, if it's not, whether its content must be kept or replaced

Table 4 shows a comparison between SQL inserts and the bulk loader.

SQL Inserts	Bulk Loader
Rows are inserted one at a time.	The load utility inserts formatted pages of data.
All rows are logged.	LOG NO can be used to prevent logging, especially when the volume of data is large.
If the table has triggers defined on it, they will fire.	Triggers are not fired.
All constraints are validated.	Constraints might or might not be validated during the load process, depending on the load input statements.
Parallelism cannot be exploited.	Beginning with DB2 Version 6, the load utility can perform parallel index builds.
The DELETE statement, which causes logging, must be executed before the SQL inserts if the table content is being replaced.	The option REPLACE enables the table content to be deleted without the overhead due to logging.
If indexes exist, their keys are populated one at a time.	During the load, index keys are sorted and used to build the indexes after the table has been loaded.

continued

Table 4. Differences between SQL Inserts and Bulk Loader

SQL Inserts	Bulk Loader
Statistics cannot be collected during the inserts.	Statistics can be collected as part of the load process.
No backup image is required.	With the option LOG NO, the table space of the table that is being loaded is placed in COPY PENDING STATUS when the load process finishes.* To re-set the table space status to normal, either a backup is performed or the DB2 REPAIR utility is invoked. Otherwise, all the tables in the table space are inaccessible for modifications.

* In the DB2 Versions 6 and 7, the load option NOCOPYPEND re-sets the table-space status even if LOG NO is used.

Table 4 (continued). Differences between SQL Inserts and Bulk Loader

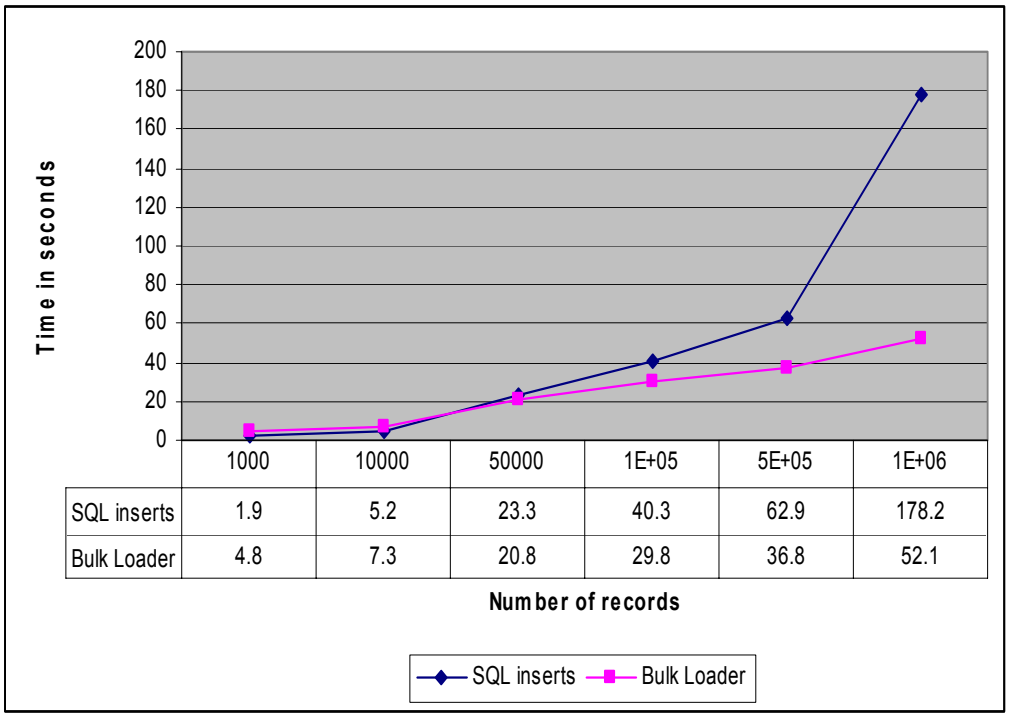
To demonstrate the difference between SQL inserts and the bulk loader, three charts are shown that represent the total time that is necessary to load different amounts of data into a table. In the first chart, the table does not have indexes; in the second and third charts, the table has, respectively, one and two non-unique indexes defined. All tests were run against an empty table.

The following table was used for the test:

```
CREATE TABLE EMPPROJACT (
 EMPNO      VARCHAR (0006) NOT  NULL
,PROJNO     VARCHAR (0006) NOT  NULL
,ACTNO      FLOAT          NOT  NULL
,EMPTIME    FLOAT                    WITH DEFAULT
,EMSTDATE   DATE                     WITH DEFAULT
,EMENDATE   DATE                     WITH DEFAULT)
```

The tests were performed loading 1000, 10000, 50000, 100000, 500000, and 1000000 records. The data wasn't sorted. Results will vary from site-to-site due to different workloads, machines, and software.

Test 1

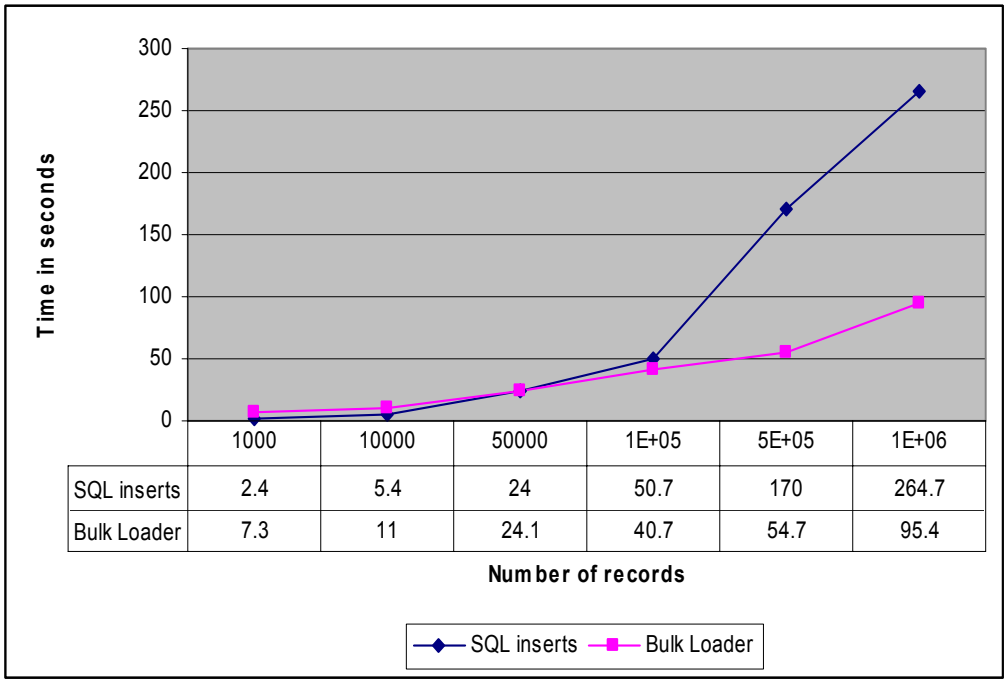


	1000	10000	50000	1E+05	5E+05	1E+06
SQL inserts	1.9	5.2	23.3	40.3	62.9	178.2
Bulk Loader	4.8	7.3	20.8	29.8	36.8	52.1

Figure 2. SQL Inserts vs. Bulk Loader on a Table without Indexes

Test 2 The following index was added to the table:

```
CREATE INDEX XEMP1 ON EMPPROJACT (EMPNO) CLUSTER
```



	1000	10000	50000	1E+05	5E+05	1E+06
SQL inserts	2.4	5.4	24	50.7	170	264.7
Bulk Loader	7.3	11	24.1	40.7	54.7	95.4

Figure 3. SQL Inserts vs. Bulk Loader on a Table That Has One Non-Unique Cluster Index

Test 3 A second index was added to the table:

```
CREATE INDEX XEMP2 ON EMPPROJACT (PROJNO)
```

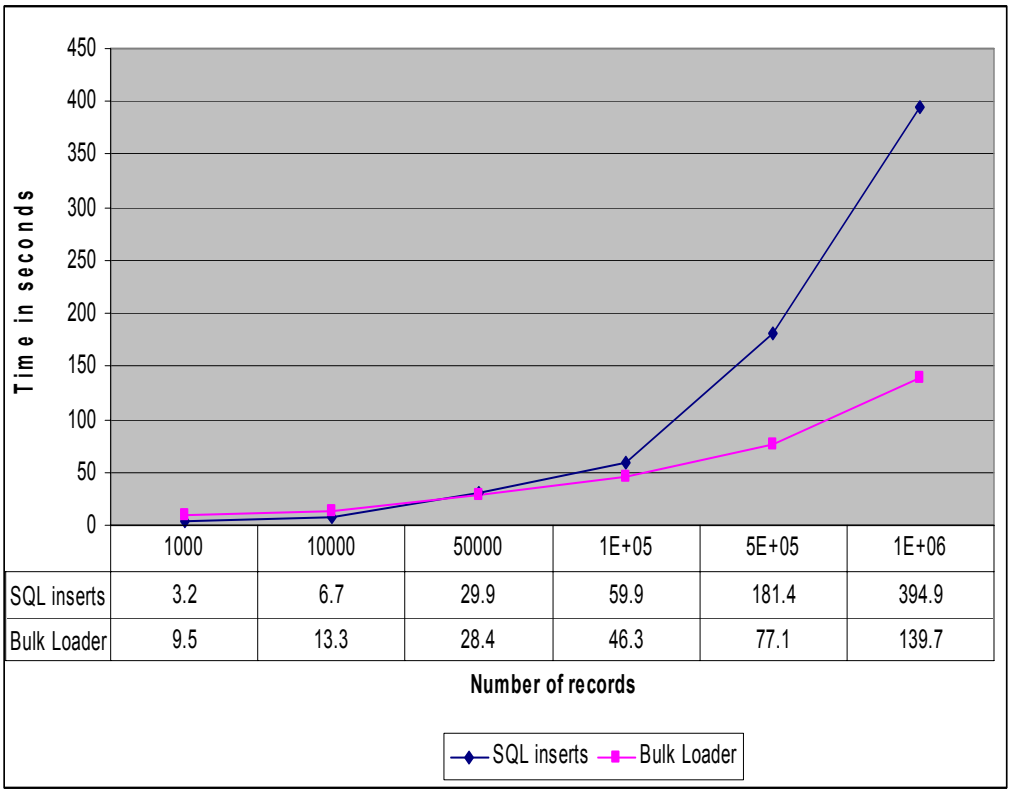


Figure 4. SQL Inserts vs. Bulk Loader on a Table That Has Two Non-Unique Indexes

Example 1 Create and load data into a DB2 table.

```
/* Setup: Create a small SAS sample table */

data work.customers;
     input custname $ 1-10
           custnum
           custcity $ 22-36;

datalines;
Beach Land    16     Ocean City
Coast Shop     3     Myrtle Beach
Coast Shop     5     Myrtle Beach
Coast Shop    12     Virginia Beach
Coast Shop    14     Charleston
Del Mar        3     Folly Beach
Del Mar        8     Charleston
Del Mar       11     Charleston
New Waves      3     Ocean City
New Waves      6     Virginia Beach
Sea Sports     8     Charleston
Sea Sports    20     Virginia Beach
```

```
Surf Mart    101      Charleston
Surf Mart    118      Surfside
Surf Mart    127      Ocean Isle
Surf Mart    133      Charleston
run;

libname db2lib db2 ssid=db2a;

data db2lib.customers (bulkload=yes
  dbtype=(custnum="smallint"));
     set work.customers;
run;
```

Example 2 Load a table, collect statistics on it, and re-set the table-space status to normal after the process completes.

```
libname db2lib db2 ssid=db2a;

data db2lib.customers
    (bulkload=yes
     bl_db2ldct1='REPLACE LOG NO NOCOPYPEND'
               'STATISTICS REPORT YES');
     set work.customers;
run;
```

Note: Using the NO COPYPEND option avoids the need to run the COPY or the REPAIR utility to reset the tablespace status.

Example 3 Append data to a DB2 table

```
/* Setup: Create a small SAS sample table */

data work.customers;
     input custname $ 1-10
           custnum
           custcity $ 22-36;

datalines;
Sea World      4      Cape Hatteras
Sea World      9      Cape Hatteras
Sea World      7      Cape Hatteras
Sea World     24      Cape Hatteras
run;

libname db2lib db2 ssid=db2a;

data db2lib.customers (bulkload=yes
  bl_db2tblxst=yes

  bl_db2ldct1='RESUME YES');
```

```
     set work.customers;
run;
```

Note: Be sure to use uppercase when passing parameters to the LOAD utility with the BL_DB2LDCT1 option.

Example 4 Replace the content of a DB2 table.

```
/* Setup: Create a small SAS sample table */

data work.customers;
     input custname $ 1-10
           custnum
           custcity $ 22-36;
datalines;
Beach Land    16     Ocean City
Coast Shop     3     Myrtle Beach
Coast Shop     5     Myrtle Beach
Coast Shop    12     Virginia Beach
 Coast Shop   14     Charleston
Del Mar        3     Folly Beach
Del Mar        8     Charleston
Del Mar       11     Charleston
New Waves      3     Ocean City
New Waves      6     Virginia Beach
Sea Sports     8     Charleston
Sea Sports    20     Virginia Beach
Surf Mart    101     Charleston
Surf Mart    118     Surfside
Surf Mart    127     Ocean Isle
Surf Mart    133     Charleston
Sea World      4    Cape Hatteras
Sea World      9    Cape Hatteras
Sea World      7    Cape Hatteras
Sea World     24    Cape Hatteras
run;

libname db2lib db2 ssid=db2a;

data db2lib.customers (bulkload=yes
  bl_db2tblxst=yes
  bl_db2ldct1='REPLACE');
     set work.customers;
run;
```

Note: Be sure to use uppercase when passing parameters to the LOAD utility with the BL_DB2LDCT1 option.

Example 5 Generate control and data files, create the table, but do not run the utility to load it.

```
libname db2lib db2 ssid=db2a;

data db2lib.customers (bulkload=yes
     bl_db2ldext=genonly
     bl_db2in='testuser.sysin'
     bl_db2rec='testuser.sysrec'
     bl_db2ldct1='REPLACE');
     set work.customers;
run;
```

Note: Be sure to use uppercase when passing parameters to the LOAD utility with the BL_DB2LDCT1 option.

Example 6 Use the control and data files that were generated by the preceding program to load the table.

```
libname db2lib db2 ssid=db2a;

data db2lib.customers (bulkload=yes
     bl_db2ldext=userun
     bl_db2tblxst=yes
     bl_db2in='testuser.sysin'
     bl_db2rec='testuser.sysrec');
     set work.customers;
run;
```

Example 6 requires that the input file be specified (even though the data that is being loaded is physically in the SYSREC file) because the input file contains the descriptions for the table fields that are used by SAS/ACCESS during the load. For Example 6, it would be more efficient to eliminate the input file by replacing it with one or more fictitious variables. Notice that the number of variables does not have to correspond to the number of fields that are in the table.

```
libname db2lib db2 ssid=db2a;

data db2lib.customers (bulkload=yes
     bl_db2ldext=userun
     bl_db2tblxst=yes
     bl_db2in='testuser.sysin'
     bl_db2rec='testuser.sysrec');
     custname='';
run;
```

Another solution for this problem is to self-reference the table in the SET statement if the table already exists, as shown in the following example:

```
libname db2lib db2 ssid=db2a;

data db2lib.customers (bulkload=yes
     bl_db2ldext=userun
```

```
     bl_db2tblxst=yes
     bl_db2in='testuser.sysin'
     bl_db2rec='testuser.sysrec');
     set db2lib.customers;
run;
```

Example 7 Load a table by using another DB2 table as input.

```
libname db2lib db2 ssid=db2a;

data db2lib.customers (bulkload=yes
     bl_db2tblxst=yes
     bl_db2ldct1='RESUME YES');
     set db2lib.new_customers;
run;
```

Example 8 Generate SMS-managed control and data files; do not create the table, and do not run the utility to load it.

```
libname db2lib db2 ssid=db2a;

data db2lib.customers (bulkload=yes
     bl_db2ldext=genonly
     bl_db2in='testuser.sysin'
     bl_db2rec='testuser.sysrec'
     bl_db2tblxst=yes
     bl_db2ldct1='REPLACE'
     bl_db2dataclas='STD'
     bl_db2mgmtclas='STD'
     bl_db2storclas='STD');
     set work.customers;
run;
```

Example 9 Append data to a table by using another DB2 table as input. This enables the SYSREC file to extend to a maximum of 6 volumes.

```
libname db2lib db2 ssid=db2a;

data db2lib.customers (bulkload=yes
     bl_db2tblxst=yes
     bl_db2ldct1='RESUME YES'
     bl_db2unitcount=6);
     set db2lib.merge_customers;
run;
```

Example 10 Load data into a table and collect statistics.

```
libname db2lib db2 ssid=db2a;

data db2lib.employees (bulkload=yes
     bl_db2tblxst=yes
     bl_db2ldct1='REPLACE'
     bl_db2ldct3= 'RUNSTATS TABLESPACE
```

```
      MYDB.EMP SHRLEVEL REFERENCE');
      set work.emp_list;
    run;
```

In Example 10, the statistics could also have been collected by adding the keyword STATISTICS in the BL_DB2LDCT1 option. Notice that the RUNSTATS utility requires that the database and table-space names be provided, which means that the user must know in advance where the table resides.

Example 11 Load data into a table and invoke the REPAIR utility to re-set the table-space status to READ and WRITE.

```
libname db2lib db2 ssid=db2a;

data db2lib.employees (bulkload=yes
     bl_db2tblxst=yes
     bl_db2ldct1='REPLACE'
     bl_db2ldct3= 'REPAIR SET TABLESPACE
     MYDB.EMP NOCOPYPEND');
     set work.emp_list;
   run;
```

In Example 11, the table-space status could also have been re-set by adding NOCOPYPEND to the option BL_DB2LDCT1. However, the option BL_DB2LDCT1 is available only in DB2 Version 6 and later.

Maximizing Performance with the SAS/ACCESS Interface to DB2

Among the factors that affect DB2 performance are the size of the table that is being accessed and the form of the SQL SELECT statements. As a general rule, if the table is larger than 10,000 rows (or 1,000 pages), you should evaluate all SAS programs that access the table directly. In evaluating, consider the following questions:

- Can any of the selected columns be dropped?
- Do the WHERE-clause criteria retrieve only those rows that are needed for subsequent analysis?
- Is the data used by more than one SAS procedure? If yes, consider extracting the data into a SAS data file for the SAS procedures to use, instead of having each procedure directly accessing the DB2.
- Do the rows need to be in a specific order? If yes, can an indexed column be used to order them? If there is no index column, is DB2 or SAS performing the sort?
- Do the WHERE-clause criteria enable DB2 to use the available indexes efficiently?
- What kind of locks does DB2 need to acquire?
- Are joins being passed to DB2?

- Can your DB2 system use parallel processing to access the data more quickly?
- Are statistics up-to-date on the DB2 tables that are being accessed?

Note: Statistics should be collected on DB2 objects. Failure to do so might result in poor performance for applications (such as SAS) that access the database. Your DBA should collect statistics periodically.

DB2 has a Resource Limit Facility (RLF) to limit the execution time of dynamic SQL statements. If the time limit is exceeded, the dynamic statement is terminated with SQL code -905. The RLF can stop a user from consuming large quantities of CPU time in the following situations:

- A very complex query accesses many tables and data. If the SQL statement is already optimized, consider breaking it into smaller chunks. Each chunk can produce intermediate results that are processed by the next chunk until the result set that you want is generated.
- An extensive search is being executed by the FSEDIT, FSVIEW, or FSBROWSE procedures.
- Extensive extraction of data from DB2. Breaking up the extraction might be a reasonable workaround.
- A large load is being entered into a DB2 table. Lowering the commit frequency or bulk loading might overcome the problem.

Here are some tips and techniques to improve SAS performance with DB2:

- Set the SAS system option SASTRACE=',,,d' (DB2DBUG for SAS 8.2 and earlier). This option logs SQL that is passed to DB2, thereby enabling you to verify WHERE clauses, PROC SQL joins, and ORDER BY clauses that are being passed to DB2.
- Verify that SAS procedures and DATA step code share connections where possible. You can do this by using one SAS libref for all SAS applications that read DB2 data, and by accepting the default value of SHAREDREAD for the CONNECTION option in the LIBNAME statement.
- If your DB2 subsystem supports parallel processing, assign a value to the CURRENT DEGREE special register. Setting this register might enable your SQL query to use parallel operations. To set the special register, use the options DBCONINIT or DBLIBINIT in the LIBNAME statement with the DB2 SET statement as shown in the following example:

```
libname db2lib db2
  dbconinit="SET CURRENT DEGREE='ANY'";
```

- With a SAS application that reads data twice, let SAS/ACCESS spool the DB2 data, which occurs via the default SPOOL=YES. The spool file is read when the application re-reads the data and when the application scrolls forward or backward through the data.

- Use the SQL procedure to pass joins to DB2 instead of using the MATCH MERGE capability (that is, merging with a BY statement) in a DATA step.
- Use the option DBKEY with SAS processing that involves the option KEY. When you use the option DBKEY, the DB2 engine generates a WHERE clause with parameter markers. Values for the key are substituted into the parameter markers in the WHERE clause.

> Without the DBKEY option, a new WHERE clause is generated for each key value. When the key value changes, a SQL query with the new WHERE clause must be optimized by BD2. A SQL query with parameter markers is optimized, or prepared, only once.

- Stored procedures can improve performance in client/server applications by reducing network traffic.

Making the Most of Your Connections

In SAS versions after SAS 6, SAS/ACCESS supports more than one connection to DB2. This is very useful. One advantage is that this segregates tasks that fetch rows by using a cursor from tasks that must issue commits. This eliminates re-synchronizing the cursor, preparing the statement again, and re-fetching rows until you are positioned back to the row that you were on. In general, when tables are opened for fetch, no commits are issued; when tables are open for update, commits can take place.

You control how the DB2 engine uses connections by using the CONNECTION option in the LIBNAME statement. At one extreme, when CONNECTION=UNIQUE, each table access creates and uses its own connection. At the other extreme, when CONNECTION=SHARED, only one connection is made, and input, update, and output accesses all share that connection.

The default CONNECTION=SHAREDREAD causes tables that are opened for input to share one connection, and update and output opens get separate connections. CONNECTION=SHAREDREAD results in the best separation between tasks that fetch from cursors and tasks that must issue commits, and eliminates the re-synchronization of cursors.

The values GLOBAL and GLOBALREAD for the CONNECTION option perform similarly to SHARED and SHAREDREAD. The difference is that you can share the connection across all librefs that you specify as GLOBAL or GLOBALREAD.

Most often, the default CONNECTION=SHAREDREAD is optimal but, if you use multiple librefs, GLOBALREAD enables you to have one connection for all input opens, regardless of which libref you use. In a single-user environment (as opposed to a SAS server session), you might know that you won't have multiple opens at the same time. In this case, you can use CONNECTION= SHARED (or CONNECTION=GLOBAL for multiple librefs). This eliminates the overhead of creating separate connections for input, update, and output transactions and, having only one open at a time, eliminates re-synchronizing input cursors.

The values SHARED and GLOBAL also eliminate - 911 deadlock when opening a table for output while opening another table in the same database for input.

The first connection to DB2 is made from the main SAS task. Subsequent connections are made from SAS subtasks because DB2 allows only one connection per task. Due to subtask overhead, the main SAS task is faster in terms of CPU time. If you are reading or writing large numbers of rows, the values SHAREDREAD or GLOBALREAD enable sharing the first connection.

One other type of connection that the DB2 engine uses, which can now be controlled by using the option UTILCONN_TRANSIENT (see the next section), is the utility connection. The utility connection accesses the system catalog, issues commits to release locks, and is a separate connection. Utility procedures such as DATASETS and CONTENTS can cause the utility connection to be created, although other actions might also necessitate it creation. There is one utility connection per LIBNAME statement, but it is not created until it is needed. If you have critical steps that must use the main SAS task connection for performance reasons, don't use the option DEFER=YES in the LIBNAME statement. It is possible that the utility connection can be established from that task, which causes the connection that you use for your opens to come from a slower subtask.

UTILCONN_TRANSIENT Option

By default, the utility connection persists for the duration of the SAS session. The new option UTILCONN_TRANSIENT is used to determine whether an idle utility connection should be closed. UTILCONN_TRANSIENT=YES, the default setting, causes SAS to close the connection when it is not in use, therefore reducing the total number of concurrent client connections.

Accessing DB2 Catalog Tables

Some SAS procedures access the DB2 system catalog for information. For example, PROC DATASETS and PROC CONTENTS need to query SYSIBM.SYSTABLES.

If the SAS 9.1 Open Metadata Server is installed at your site, it retrieves metadata from DB2 system catalog tables. The SYSIBM.SYSTABLES, SYSIBM.SYSINDEXES, SYSIBM.SYSKEYS, SYSIBM.SYSFOREIGNKEYS, and SYSIBM.SYSRELS tables are accessed.

Note: If the SAS user does not have authorization to read these DB2 system catalog tables, some procedures or applications fail.

Other tables being accessed by SAS include SYSIBM.SYSPARMS for stored procedure support, and SYSIBM.SYSKEYS (along with SYSIBM.SYSINDEXES and SYSIBM.SYSCOLUMNS) for threaded reads support.

Controlling DB2 Locks

SAS/ACCESS provides the following four options to control how locks are acquired on a database resource:

- READ_LOCK_TYPE
- UPDATE_LOCK_TYPE

- READ_ISOLATION_LEVEL
- UPDATE_ISOLATION_LEVEL

The options READ_LOCK_TYPE and UPDATE_LOCK_TYPE refer to the type of locks to be acquired when a table is opened for read or update processing. The only valid value for these options is TABLE, which means that a table-level lock is acquired.

The options READ_ISOLATION_LEVEL and UPDATE_ISOLATION_LEVEL indicate the degree of granularity that is used to process rows in a result set. Depending on the value that is specified, concurrency is impacted. However, read-only applications can benefit by using a value that avoids any potential conflict on database resources.

Table 5 shows all the values that can be used for the READ_ISOLATION_LEVEL and UPDATE_ISOLATION_LEVEL options. For a complete explanation of isolation level values, see the DB2 for OS/390 and z/OS documentation from IBM.

Value	Isolation Level
CS	Cursor stability
UR	Uncommitted read
RR	Repeatable read
“RR KEEP UPDATE LOCKS” *	Repeatable read, keep update locks
RS	Read stability
“RS KEEP UPDATE LOCKS” *	Read stability, keep update locks

*When specifying a value that consists of multiple words, enclose the entire string in quotation marks.

Table 5. Isolation Level for DB2 OS/390 and z/OS

Binding SAS/ACCESS to DB2

Whether you are using CAF or RRSAF, access to DB2 takes place by using an execution plan. Before you can manipulate database data, SAS/ACCESS must be bound to DB2.

A plan tells DB2 how and what an application will do against the database. A plan also indicates to DB2 what to do when the application uses database resources (for example, databases, table spaces, tables, and so on), specifically how the resources are locked, and when the locks are released.

Some important BIND parameters are listed below. Their values can be changed by the DBA if necessary. Let’s look at what the impact of these parameters can be.

CURRENTDATA

determines whether to require data currency for read-only and ambiguous cursors when the isolation level of cursor stability is specified (ISOLATION(CS)). It also determines whether block fetching can be used for distributed, ambiguous cursors. This parameter has a big impact on DRDA access. The SAS default is CURRENTDATA(YES).

A distributed cursor is one that accesses data at remote locations. A cursor is said to be ambiguous when DB2 cannot determine whether or not it is read-only.

(YES) specifies that currency is required for read-only and ambiguous cursors. DB2 acquires page or row locks to ensure data currency. Block fetching for distributed, ambiguous cursors is inhibited.

(NO) specifies that currency is not required for read-only and ambiguous cursors. Block fetching for distributed, ambiguous cursors is allowed.

DEFER/NODEFER

determines whether to defer preparation for dynamic SQL statements that refer to remote objects or to prepare the dynamic statements immediately. If you defer preparation, the dynamic statement prepares when DB2 first encounters an EXECUTE, an OPEN, or a DESCRIBE statement that refers to the dynamic statement. The SAS default is DEFER(PREPARE).

If you choose to defer PREPARE statements, after the EXECUTE or the DESCRIBE statement, you should code your application to handle any SQL error codes or SQLSTATEs that the PREPARE statement might return. You can defer PREPARE statements only if you specify the bind option DEFER(PREPARE).

To improve performance, consider using DEFER(PREPARE), especially when binding for DRDA access. DB2 does not prepare the dynamic SQL statement until that statement executes. This reduces network traffic, which improves the performance of the SQL statement.

KEEPDYNAMIC

determines whether DB2 keeps dynamic SQL statements after commit points. The SAS default is KEEPDYNAMIC(NO).

(*NO*) specifies that DB2 does not keep dynamic SQL statements after commit points.

(*YES*) specifies that DB2 keeps dynamic SQL statements after commit points.

If you specify KEEPDYNAMIC(YES), the application does not need to prepare a SQL statement after every commit point. DB2 keeps the dynamic SQL statement until one of the following occurs:

- The application process ends.
- A rollback operation occurs.
- The application executes an explicit PREPARE statement with the same statement identifier.

If you specify KEEPDYNAMIC(YES) and the prepared statement cache is active, DB2 keeps a copy of the prepared statement in the cache. If the prepared statement cache is not active, DB2 keeps only the SQL statement string past a commit point. Then, DB2 implicitly prepares the SQL statement if the application executes an OPEN, an EXECUTE, or a DESCRIBE operation for that statement.

If you specify KEEPDYNAMIC(YES), do not specify REOPT(VARS). KEEPDYNAMIC(YES) and REOPT(VARS) are mutually exclusive.

With KEEPDYNAMIC(YES), problems can arise when the majority of the SQL executed is dynamic. The EDM pool should be monitored to determine whether more memory is necessary to cache the dynamic SQL that is being run.

Creating and Copying Database Tables

SAS allows database tables to be created explicitly (using explicit pass-through) or implicitly (the result of a DATA step). When a table is created implicitly, SAS uses its default data types—CHAR, FLOAT, DATE, TIME, and DB2 TIMESTAMP—to define each field, and assumes that nulls are allowed. This might not always be advantageous, especially when an exact copy of a DB2 table is required.

The DBTYPE option enables you to replicate DB2 data types. While the DBTYPE option provides a way to specify data column-level types, lengths, and constraints, the parameters at the table definition level can be specified by using DBCREATE_TABLE_OPTS. (For details, see “DBCREATE_TABLE_OPTS Option” later in this section).

DBTYPE Option

Use the DBTYPE option to specify data types, lengths, and constraints for a column. Let’s look at the following example.

```
options sastrace=',,,d ';
libname db2lib db2 ssid=db2a authid=dsn8710;
libname mylib  db2 ssid=db2a;

data mylib.eact;
     set db2lib.eact;
run;
```

The table EACT is created from the existing DSN8710.EACT table. The definition for the source table and the SAS log follow:

```
  CREATE TABLE DSN8710.EACT
  (
   ACTNO         SMALLINT            NOT  NULL
   CHECK (ACTNO > 0)
  ,ACTKWD        CHAR         (0006) NOT  NULL
  ,ACTDESC       VARCHAR      (0020) NOT  NULL
  ,RID           CHAR         (0004) WITH DEFAULT
  ,TSTAMP        TIMESTAMP           WITH DEFAULT
  )
```

SAS® log:

```
1    options sastrace=',,,d ';
2    libname db2lib db2 ssid=db2a authid=dsn8710;
NOTE: Libref DB2LIB was successfully assigned as follows:
      Engine:        DB2
      Physical Name: DB2A
3    libname mylib db2 ssid=db2a;
NOTE: Libref MYLIB was successfully assigned as follows:
      Engine:        DB2
      Physical Name: DB2A
4    data mylib.eact;
5         set db2lib.eact;
SELECT * FROM DSN8710.EACT FOR FETCH ONLY
6    run;
SELECT * FROM EACT FOR FETCH ONLY
DB2 SQL Error, sqlca->sqlcode=-204
COMMIT WORK
NOTE: SAS variable labels, formats, and lengths are not written to DBMS
tables.
CREATE TABLE EACT (ACTNO FLOAT, ACTKWD CHAR(6),
                   ACTDESC CHAR(20), RID CHAR(4),
                   TSTAMP TIMESTAMP)
COMMIT WORK
NOTE: There were 0 observations read from the data set DB2LIB.EACT.
COMMIT WORK
NOTE: The data set MYLIB.EACT has 0 observations and 5 variables.
COMMIT WORK
NOTE: The DATA statement used 0.05 CPU seconds and 12789K.

NOTE: The address space has used a maximum of 1516K below the line and 15192K
above the line.
```

The definition of the new table does not match the original. You must use the DBTYPE option to alter this behavior as shown in the following example:

```
options sastrace=',,,d ';

libname db2lib db2 ssid=db2a authid=dsn8710;
libname mylib  db2 ssid=db2a;

data mylib.eact
    (dbtype=(actno= 'smallint not null
                    check (actno>0) '
             actkwd= 'char(6) not null '
             actdesc= 'varchar(20) not null '
             rid= 'char(4) '));
     set db2lib.eact;
run;
```

All columns except TSTAMP were specified because its data type is one of the default data types that are used by SAS, which means that translation is not necessary. The same is true for the ACTDESC field. However, this field was specified because of the 'NOT NULL' clause.

Using the DBTYPE option, the new table definition is identical to the original.

```
CREATE TABLE EACT
(
 ACTNO         SMALLINT            NOT  NULL
 CHECK (ACTNO > 0)
,ACTKWD        CHAR         (0006) NOT  NULL
,ACTDESC       VARCHAR      (0020) NOT  NULL
,RID           CHAR         (0004) WITH DEFAULT
,TSTAMP        TIMESTAMP           WITH DEFAULT
)
```

The DBTYPE option can generate primary and unique keys too, as long as they are single-column constraints. For example:

```
options sastrace=',,,d ';

libname db2lib db2 ssid=db2a authid=dsn8710;
libname mylib  db2 ssid=db2a;

data mylib.eact
    (dbtype=(actno='smallint not null
                    primary key'
             actkwd='char(6) not null'
             actdesc='varchar(20) not null'
                rid='char(4)'));
     set db2lib.eact (obs=0);
run;
```

Notice that the OBS=0 option was used to allow the new table to be created while also preventing records from being inserted. Otherwise, SAS would try to copy the records, but the copy would fail and the table definition would be marked as "incomplete" by DB2. To "complete" the table, define an index on the primary key column.

In SAS, an index can only be added by using the Pass-Through Facility. Because of this limitation and because primary and unique constraints can only be single column, the DBTYPE option is not useful when constraints are involved. The following code is preferable:

```
options sastrace=',,,d ';

libname db2lib db2 authid=dsn8710;
libname mylib  db2;

proc sql;
     connect to db2 (ssid=db2a);
     execute (CREATE TABLE EACT
             (
              ACTNO   SMALLINT         NOT  NULL
```

```
              ,ACTKWD  CHAR      (0006) NOT  NULL
              ,ACTDESC VARCHAR   (0020) NOT  NULL
              ,RID     CHAR      (0004) WITH DEFAULT
              ,TSTAMP  TIMESTAMP        WITH DEFAULT
              ,
              PRIMARY KEY
              (
              ACTNO
              )
             )) by db2;
     execute (CREATE UNIQUE INDEX XEACT1 ON EACT
                    (ACTNO) CLUSTER) by db2;
     disconnect from db2;
     insert into mylib.eact
            select * from db2lib.eact;
quit;
```

DBCREATE_TABLE_OPTS Option

The option DBCREATE_TABLE_OPTS allows some CREATE TABLE parameters to be generated. Let's look at the following example.

```
options sastrace=',,,d ';

 libname db2lib db2 ssid=db2a authid=dsn8710;
 libname mylib  db2 ssid=db2a;

 data mylib.eact (dbcreate_table_opts=
                  'CCSID UNICODE');
      set db2lib.eact;
 run;
```

Following is the DDL statement that is passed to the database:

```
CREATE TABLE EACT (ACTNO FLOAT,
                   ACTKWD VARCHAR(6),
                   ACTDESC VARCHAR(20),
                   RID VARCHAR(4),
                   TSTAMP TIMESTAMP) CCSID UNICODE
```

The CCSID clause is appended to the table definition after the final parenthesis ("**)**").

PRESERVE_TAB_NAMES and PRESERVE_COL_NAMES Options

By default, when SAS creates a database table or refers to an existing one, it changes table and column names to uppercase, unless PRESERVE_TAB_NAMES=YES and PRESERVE_COL_NAMES=YES are used.

Example 1 Select data from the DB2 table DSN8710.STAFF. If you don't specify PRESERVE_TAB_NAMES=YES, an error is returned.

```
options sastrace=',,,d ';

libname db2lib db2 ssid=db2a authid=dsn8710
        preserve_tab_names=YES;

proc sql;
     select * from db2lib.Staff;
quit;
```

Following is the SQL statement that is passed to the database:

```
SELECT * FROM "DSN8710"."Staff" FOR FETCH ONLY
```

Example 2 Create the DB2 table DSN8710.STAFF; preserve the column names as they are entered in the code.

```
options sastrace=',,,d';

libname db2lib db2 ssid=db2a authid=dsn8710
        preserve_col_names=yes;

proc sql;
     create table db2lib.staff (
            id      char(30),
            dept    float,
            salary  float);
quit;
```

Following is the DDL statement that is passed to the database:

```
CREATE TABLE STAFF (
       "id" CHAR(30),
       "dept" FLOAT,
       "salary" FLOAT)
```

Enhancing the Performance of Joins between Database Tables and SAS Tables

A join of a SAS table and a DB2 table can be optimized by using the options MULTI_DATASRC_OPT, DBKEY, and DBINDEX.

MULTI_DATASRC_OPT Option

When no SAS options are specified and heterogeneous data sources are involved in a join, the objects that are being joined are read into SAS tables and the join is performed by SAS.

When MULTI_DATASRC_OPT=IN_CLAUSE is specified in a PROC SQL join operation, the unique values of the join column are retrieved from the SAS table to construct an IN clause. The IN clause is built and passed to the DBMS to retrieve data from the larger DB2 table, which dramatically reduces the time that is needed to process the join. If the join involves only database

tables, MULTI_DATASRC_OPT=IN_CLAUSE is ignored, and the join is performed by the database using an equality predicate (column=column).

MULTI_DATASRC_OPT is used only when SAS is processing a join by using PROC SQL. It is not used in DATA step processing. For some join operations that involve additional sub-setting of the query, PROC SQL might determine that it is more efficient to process the join internally, and it will not use the MULTI_DATASRC_OPT option, even if it's specified.

The option MULTI_DATASRC_OPT might provide performance improvement over the option DBKEY. If both options are specified, DBKEY overrides MULTI_DATASRC_OPT.

Note: If a SAS data set is used in the join, no matter how large, it will be used to generate the IN clause. Too large a SAS data set will slow performance, or will cause DB2 to reject the statement if its length exceeds the maximum length for a SQL statement, which is 32K for DB2, Version 7.

In the following example, the option MULTI_DATASRC_OPT is used to improve the performance of a SQL join statement. MULTI_DATASRC_OPT instructs PROC SQL to subset DB2 rows by using an IN clause built from the SAS table. The IN clause is built from the unique values of the SAS table's column DeptNo; as a result, only rows that match the WHERE clause are retrieved from the DBMS. Without the option MULTI_DATASRC_OPT, PROC SQL retrieves all the rows from the EMP table and applies the WHERE processing in SAS. For a large DB2 EMP table, this would be CPU- and I/O-intensive.

```
libname db2lib db2 ssid=db2a authid=dsn8710
        multi_datasrc_opt=in_clause;

data smallds;
     deptno='C01';
     output;
     deptno='B01';
     output;
     deptno='A01';
     output;
run;

proc sql;
     select bigtab.empno, bigtab.salary
       from db2lib.emp bigtab,
            work.smallds smallds
      where bigtab.workdept=smallds.deptno;
quit;
```

Following is the SQL statement that is created by SAS/ACCESS and passed to DB2:

```
SELECT "EMPNO", "SALARY", "WORKDEPT"
 FROM DSN8710.EMP
 WHERE (("WORKDEPT" IN ('A01','B01','C01')))
   FOR FETCH ONLY
```

DBKEY Option

The option DBKEY increases performance when the SAS data set is small. DBKEY causes each value in the SAS transaction data set to generate a new result set (or open cursor) from the DBMS table. For example, if your SAS table has 100 observations with unique key values, you request 100 result sets from the DBMS. This might be expensive. You should determine if using DBKEY is appropriate, or whether you would achieve better performance by reading the entire DBMS table (or by creating a subset of the table).

DBKEY does not require database indexes to be defined. It instructs SAS to use the specified DBMS column name or names in the WHERE clause that is passed to the DBMS in the join.

The DBKEY option can also be used in a DATA step with the KEY option in the SET statement to improve the performance of joins. To do this, you specify KEY=DBKEY. The following DATA step creates a new data file by joining the SAS data file SMALLDS with the DBMS table EMP. The variable WORKDEPT is used with the option DBKEY to cause a WHERE clause to be issued by SAS/ACCESS.

```
libname db2lib db2 ssid=db2a authid=dsn8710;

data smallds;
     empno='000010';
     output;
     empno='000020';
     output;
     empno='000030';
     output;
run;

data join_table;
     set smallds;
     set db2lib.emp (dbkey=empno)
         key=dbkey;
run;
```

Following is the SQL statement that is created by SAS/ACCESS and passed to DB2:

```
SELECT "EMPNO", "FIRSTNME", "MIDINIT",
       "LASTNAME", "WORKDEPT", "PHONENO",
    "HIREDATE", "JOB", "EDLEVEL", "SEX",
    "BIRTHDATE", "SALARY", "BONUS", "COMM"
  FROM DSN8710.EMP
 WHERE ("EMPNO"=?)
   FOR FETCH ONLY
```

Note: When you use DBKEY with MODIFY statements in the DATA step, there is no implied ordering of the data that is returned from the database. If the master DBMS table contains records that have duplicate key values, using DBKEY can cause unexpected behavior. Because SAS re-generates result sets (open cursors) during transaction processing, the changes that you make during processing impact the results of subsequent queries. Therefore, before you use DBKEY in this context, determine whether your master DBMS file has duplicate values for keys. (Remember that the REPLACE, OUTPUT, and REMOVE statements in SAS can cause duplicate values to

appear in the master table). The DBKEY option does not require or check for the existence of indexes that are created on the DBMS table.

If you perform a join and use PROC SQL, you must ensure that the columns that are specified by using the DBKEY option match the columns that are specified in the SAS data set.

Updating and Deleting Data

Currently, DB2 updates and deletes are executed by using positional cursors, unless explicit pass-through is used, in which case the statement is passed to the database as is. The problem with a positional cursor is that the records that will be updated or deleted must first be selected, one at a time, to determine their position in the table. With large amounts of data, the total processing time can be significant and lead to poor performance. The following examples show how to solve this problem by implementing different techniques.

Example 1 Update a table from a SAS data set with explicit pass-through to avoid using a positional cursor.

```
%macro modify;
%do i=1 %to &sqlobs;
    proc sql;
      %if &i = 1 %then
          %do;
             connect to db2 (ssid=db2a);
          %end;
       execute (update josmith.target
                   set b = &&b_var&i
                  where a = &&a_var&i) by db2;
    quit;
%end;
%mend;

options sastrace=',,,d';

libname db2lib db2 ssid=db2a authid=josmith;

data db2lib.target;
     a=1;b=1;output;
     a=2;b=1;output;
     a=3;b=1;output;
run;

data work.source;
     a=1;b=10;output;
     a=3;b=20;output;
run;

proc sql;
```

```
       select a, b
       into :a_var1 - :a_var2,
            :b_var1 - :b_var2
       from work.source;
quit;

%modify;

proc print data=db2lib.target;
run;
```

SAS log:

```
.
.
DB2_12: Executed:
update josmith.target set b = 10 where a = 1
.
.
DB2_13: Executed:
update josmith.target set b = 20 where a = 3
.
.
```

```
    The SAS System

          a         b
  ------------------
          1        10
          2         1
          3        20
```

The advantage of this technique is represented by the fact that the source table is read entirely only once into macro variables that are then used to drive the updates through explicit pass-through. Each update can target one or more multiple records, depending on its WHERE clause. This is another advantage over positional updates, where only the current record is modified, even if more than one satisfies the WHERE clause. Note how the connection to the database is executed only once to further improve performance.

Example 2 Update a DB2 table from a SAS data set. However, instead of reading the input table and performing the update for each record read, the SAS data set is loaded into a DB2 table, and a single update statement is executed at the end of the load.

```
options sastrace=',,,d';

libname db2lib db2 ssid=db2a authid=josmith;

data db2lib.target;
```

```
     a=1;b=1;output;
     a=2;b=1;output;
     a=3;b=1;output;
run;

data work.source;
     a=1;b=10;output;
     a=3;b=20;output;
run;

data db2lib.source
    (bulkload=yes
     bl_db2ldct1='REPLACE LOG NO
                  NOCOPYPEND');
     set work.source;

run;



proc sql;
     connect to db2 (ssid=db2a);
     execute (update josmith.target a
                 set b = (select b
                 from josmith.source b
                where a = a.a)
                where a in (select a
                from josmith.source)) by db2;
quit;

proc print data=db2lib.target;
run;
```

SAS log:

```
.
.

DSNU000I    DSNUGUTC - OUTPUT START FOR UTILITY, UTILID =
JOSMITHT162000
DSNU050I    DSNUGUTC -  LOAD REPLACE LOG NO NOCOPYPEND
DSNU650I  < DSNURWI -  INTO TABLE JOSMITH.SOURCE
DSNU650I  < DSNURWI -   (A POSITION(1) DOUBLE PRECISION
NULLIF A=X'80',
DSNU650I  < DSNURWI -    B POSITION(9) DOUBLE PRECISION
NULLIF B=X'80')
DSNU350I  < DSNURRST - EXISTING RECORDS DELETED FROM
TABLESPACE
DSNU304I  < DSNURWT - (RE)LOAD PHASE STATISTICS - NUMBER
OF RECORDS=2 FOR TABLE JOSMITH.SOURCE
DSNU302I    DSNURILD - (RE)LOAD PHASE STATISTICS - NUMBER
OF INPUT RECORDS PROCESSED=2
```

```
DSNU300I    DSNURILD - (RE)LOAD PHASE COMPLETE, ELAPSED
TIME=00:00:02
DSNU010I    DSNUGBAC - UTILITY EXECUTION COMPLETE, HIGHEST
RETURN CODE=0

.
.

DB2_22: Executed:
update josmith.target a set b = (select b from
josmith.source b where a = a.a) where a in (select a from
josmith.source)

.
.

    The SAS System

          a          b
  ------------------
          1         10
          2          1
          3         20
```

The benefit of this solution is that the input table is loaded into the database and only a single update statement is necessary to change the data. Because the data is already in the database, any error that might occur during the update would result in only the update statement being executed again, not the load.

Example 3 Re-write the code from Example 2 to use a temporary table, instead of a regular table, to store the input data that is used to drive the deletes.

```
options sastrace=',,,d';

libname db2lib db2 ssid=db2a authid=josmith
        connection=global;
libname db2tmp db2 ssid=db2a dbmstemp=yes
        connection=global;

data db2lib.target;
     a=1;b=1;output;
     a=2;b=1;output;
     a=3;b=1;output;
run;

data db2tmp.source;
     a=1;b=10;output;
     a=3;b=20;output;
run;
```

```
proc sql;
     connect to db2 (connection=global
                     ssid=db2a);
     execute (delete from josmith.target a
               where exists (select 1
               from session.source
               where a=a.a)) by db2;
quit;

proc print data=db2lib.target;
run;

SAS log:

.
.

DB2_20: Executed:
delete from josmith.target a where a in (select a from
session.source where a = a.a)

.
.

    The SAS System

          a         b
  ------------------
          2         1
```

Using a temporary table is good when the input data set is small. Because a temporary table can only be populated through INSERT statements, a large amount of data would cause performance to suffer. Another disadvantage might be restartability: if the update fails, the temporary table is automatically dropped by DB2, meaning it must be reloaded before the delete executes again.

Debugging SAS Code

Sometimes, there can be a need to see how a SQL statement is being passed to the database, especially when SAS is generating the SQL statement on our behalf. Or, we might want to implement our code so that some parts of it are executed instead of other parts, based on the SQLCODE that is returned by the previous SQL statements. Also, to better tune our code, there might be instances when we need to look at the amount of time that is spent performing database operations. The following options and macro variable can solve all the preceding problems:

- SASTRACE=',,,d' option (in SAS 9.1 and later; replaces DB2DBUG)

Note: Although DB2DBUG is still supported for backward compatibility, it is recommended that SASTRACE be used instead.

- SYSDBRC macro variable
- SASTRACE=',,,s' option
- FULLSTIMER options

When a statement that accesses DB2 data is submitted, SASTRACE displays any DB2 SQL queries (generated by SAS) that are processed by DB2. The queries are written to the SAS log.

For example, if you submit a DATA step statement that creates a DB2 table by referencing another table, the DB2 "CREATE TABLE" statement is displayed in the SAS log.

```
options sastrace=',,,d ';

libname db2lib db2 ssid=db2a authid=dsn8710;
libname mylib  db2 ssid=db2a;

data mylib.emp;
     set db2lib.emp;
run;
```

SAS log:

```
1    options sastrace=',,,d';
2    libname db2lib db2 ssid=db2a authid=dsn8710;
NOTE: Libref DB2LIB was successfully assigned as follows:
      Engine:        DB2
      Physical Name: DB2A
3    libname mylib  db2 ssid=db2a;
NOTE: Libref MYLIB was successfully assigned as follows:
      Engine:        DB2
      Physical Name: DB2A
4
DB2_1: Prepared:
SELECT * FROM DSN8710.EMP FOR FETCH ONLY

5    data mylib.emp;
6         set db2lib.emp;
7    run;

DB2_2: Prepared:
SELECT * FROM EMP FOR FETCH ONLY

DB2_3: Executed:
COMMIT WORK

NOTE: SAS variable labels, formats, and lengths are not written to DBMS
tables.

DB2_4: Executed:
CREATE TABLE EMP (EMPNO CHAR(6), FIRSTNME CHAR(12),
                  MIDINIT CHAR(1), LASTNAME CHAR(15),
                 WORKDEPT CHAR(3), PHONENO CHAR(4),
```

```
                    HIREDATE DATE, JOB CHAR(8), EDLEVEL FLOAT,
                    SEX CHAR(1), BIRTHDATE DATE, SALARY FLOAT,
                    BONUS FLOAT, COMM FLOAT)

DB2_5: Executed:
COMMIT WORK

DB2_6: Executed:
SELECT * FROM DSN8710.EMP FOR FETCH ONLY

DB2_7: Prepared:
INSERT INTO EMP (EMPNO, FIRSTNME, MIDINIT, LASTNAME, WORKDEPT,
                 PHONENO, HIREDATE, JOB, EDLEVEL, SEX,
                 BIRTHDATE, SALARY, BONUS, COMM) VALUES
                (?,?,?,?,?,?,?,?,?,?,?,?,?,?)

NOTE: There were 42 observations read from the data set DB2LIB.EMP.

DB2_8: Executed:
COMMIT WORK

DB2_9: Executed:
COMMIT WORK

NOTE: The data set MYLIB.EMP has 42 observations and 14 variables.


DB2_10: Executed:
COMMIT WORK

NOTE: The DATA statement used 0.07 CPU seconds and 7605K.

NOTE: The address space has used a maximum of 1272K below the line and
11192K above the line.
```

The SYSDBRC macro variable is useful too. The following example shows how it is used to conditionally execute code that is dependent on the value of the SQLCODE returned by DB2.

```
%macro testdbrc;

 options nonotes;

 libname db2lib db2 ssid=db2a authid=dsn8710;
 libname mylib  db2 ssid=db2a;

 proc print data=mylib.emp;
 run;

 %if &sysdbrc EQ -204   /* Table not found */
     %then %do;
              %put 'The DB2 Return Code is: '
                   "&sysdbrc";
```

```
            %put 'Now I create the table...';
            data mylib.emp;
                 set db2lib.emp;
            run;
            proc print data=mylib.emp;
            run;
          %end;

 %mend testdbrc;

 %testdbrc;
```

SAS log:

```
ERROR: File MYLIB.EMP.DATA does not exist.
'The DB2 Return Code is: ' "-204"
'Now I create the table...'
```

Finally, it can be useful to know how much time is spent in the database as opposed to the total amount of time that is necessary to run code. The SASTRACE=',,,s' and the FULLSTIMER options can help, as shown in the following example.

```
options sastrace=',,,ds' fullstimer;
libname db2lib db2 ssid=db2a authid=dsn8710;

proc print data=db2lib.emp (obs=10);
run;
```

SAS log:

```
1    options sastrace=',,,ds' fullstimer;
2    libname db2lib db2 ssid=db2a authid=dsn8710;
NOTE: Libref DB2LIB was successfully assigned as follows:
      Engine:        DB2
      Physical Name: DB2A
3

DB2_1: Prepared:
SELECT * FROM DSN8710.EMP FETCH FIRST 10 ROWS ONLY FOR FETCH ONLY

4    proc print data=db2lib.emp (obs=10);
5    run;

DB2_2: Executed:
SELECT * FROM DSN8710.EMP FETCH FIRST 10 ROWS ONLY FOR FETCH ONLY

NOTE: There were 10 observations read from the data set DB2LIB.EMP.

DB2_3: Executed:
COMMIT WORK
```

DBMS statistics (sastrace=',,,s')

Code statistics (fullstimer)

```
Summary Statistics for DB2 are:
Total row fetch seconds were:                    0.000932
Total SQL prepare seconds were:                  0.000479
Total seconds used by the DB2 ACCESS engine were 1.687351

NOTE: The PROCEDURE PRINT used the following resources:
      CPU     time -         00:00:00.01
      Elapsed time -         00:00:01.68
      EXCP count   - 2
      Task  memory - 4635K (101K data, 4534K program)
      Total memory - 15113K (3104K data, 12009K program)
NOTE: The address space has used a maximum of 1220K below the line and 17316K
above the line.
```

Threaded Reads

SAS 9.1 implements threaded reads for DB2. With threaded reads, the data in a large table can be read in parallel by dividing the statement into multiple threads that are processed concurrently, which reduces the amount of time necessary to retrieve the result set. SAS creates the multiple threads and then a read connection is established between the DBMS and each thread. The result set is partitioned across the connections, and rows are passed to SAS in parallel across the connections, which improves throughput time.

The concept of parallelism is not new to DB2 users; however, DB2 limits parallelism to partitioned objects. SAS threaded reads exploit parallelism even for non-partitioned tables.

Threaded reads enable SAS applications to complete faster. However, their use can increase CPU consumption.

Threaded reads are enabled by default for several SAS procedures.

Note: Threaded reads apply only to queries executed through implicit pass-through.

How Threaded Reads Work

Threaded reads are implemented by using the MOD function, which returns the reminder of a division. This function applies only to numeric columns. For a query to be eligible for threaded reads, the table it accesses must have at least one column whose data type is SMALLINT, INTEGER, or DECIMAL. Eligible DECIMAL columns must be confined to INTEGER range, which means that the precision cannot be greater than 9; for example, DECIMAL(9,2) is acceptable, but DECIMAL(12,0) is not acceptable.

If multiple columns are eligible for partitioning, SAS accesses the DB2 catalog to choose the column according to the following priorities:

- Is the eligible column an identity column?
- Is the eligible column part of a unique index?
- Is the eligible column part of a non-unique index?

- Is there at least one non-nullable column among the eligible columns? If yes, the column is chosen based on its data type in the following order: SMALLINT, INTEGER, and DECIMAL.
- A nullable, eligible column is chosen based on its data type in the following order: SMALLINT, INTEGER, and DECIMAL.

> **Note**: If it is not possible to select a column for partitioning, SAS reverts to single-threaded access.

After the partitioning column is chosen, SAS generates as many SQL statements as specified with the DBSLICEPARM option (the default is 2). SAS appends a WHERE clause to statements that use the MOD function to create a subset of the result set. Combined, these statements return exactly the same result set as would have been returned from the original single SQL statement. The following example shows how the DBSLICEPARM option works:

```
options sastrace=',,,d';

libname db2lib db2 ssid=db2a authid=dsn8710;

proc print data=db2lib.emp
    (dbsliceparm=(ALL,2));
run;
```

SAS log:

```
.
.
.
DB2_7: Executed:
SELECT "EMPNO", "FIRSTNME", "MIDINIT", "LASTNAME",
       "WORKDEPT", "PHONENO", "HIREDATE", "JOB",
       "EDLEVEL", "SEX", "BIRTHDATE",
       "SALARY", "BONUS", "COMM"
  FROM DSN8710.EMP
 WHERE ABS(MOD("EDLEVEL",2))=0
   FOR FETCH ONLY
.
.
.
DB2_8: Executed:
SELECT "EMPNO", "FIRSTNME", "MIDINIT", "LASTNAME",
       "WORKDEPT", "PHONENO", "HIREDATE", "JOB",
       "EDLEVEL", "SEX", "BIRTHDATE",
       "SALARY", "BONUS", "COMM"
  FROM DSN8710.EMP
 WHERE (ABS(MOD("EDLEVEL",2))=1 OR
       "EDLEVEL" IS NULL)
   FOR FETCH ONLY
.
.
.
```

Note: The DBSLICEPARM= option is both a libnames and a data set option. When specified in a libnames statement, it is applied whenever possible to DATA steps and PROCS that refer to the libref.

The DBSLICE option should be used when you do not want SAS to choose the partitioning column and build the WHERE clause.

The following example shows how the DBSLICE option works:

```
options sastrace=',,,d';

libname db2lib db2 ssid=db2a authid=dsn8710;

data db2lib.toppaid;
     set db2lib.emp
        (dbslice=("SALARY>=40000"
                  "SALARY<=50000"));
     where job = "MANAGER";
run;

SAS log:

.
.
.
DB2_7: Executed:
SELECT "EMPNO", "FIRSTNME", "MIDINIT", "LASTNAME",
       "WORKDEPT", "PHONENO", "HIREDATE", "JOB",
       "EDLEVEL", "SEX", "BIRTHDATE",
       "SALARY", "BONUS", "COMM"
  FROM DSN8710.EMP
      WHERE ("JOB" ='MANAGER' ) AND  SALARY>=40000
   FOR FETCH ONLY
.
.
.
DB2_8: Executed:
SELECT "EMPNO", "FIRSTNME", "MIDINIT", "LASTNAME",
       "WORKDEPT", "PHONENO", "HIREDATE", "JOB",
       "EDLEVEL", "SEX", "BIRTHDATE",
       "SALARY", "BONUS", "COMM"
  FROM DSN8710.EMP
 WHERE ("JOB" ='MANAGER' ) AND  SALARY<=50000
   FOR FETCH ONLY
.
.
```

However, you should ensure that the value that is specified for this option does not result in duplicates or omissions in the result set. In the following example, DBSLICE generates an incorrect result set by duplicating SALES that have the value 0.

```
dbslice=("SALES<=0 or SALES=NULL" "SALES>=0")
```

The following example generates a correct result set.

```
dbslice=("SALES<=0 or SALES=NULL" "SALES>0")
```

The following example might return a wrong result set if the column EMPNO allows for nulls:

```
dbslice=("EMPNO<20" "EMPNO>=20")
```

The following example would return a correct result set.

```
dbslice=("EMPNO<20 or EMPNO IS NULL" "EMPNO>=20")
```

The SASTRACE=',,t,' option shows debugging information in the SAS log that is related to the number of threads that are created to execute the query and the amount of rows that each thread retrieved. For example:

```
options sastrace=',,t,d';

libname db2lib db2 ssid=db2a authid=dsn8710;

proc print data=db2lib.emp
    (dbsliceparm=(ALL,2));
run;
```

SAS log:

```
.
.
.
DB2_7: Executed:
SELECT "EMPNO", "FIRSTNME", "MIDINIT", "LASTNAME",
       "WORKDEPT", "PHONENO", "HIREDATE", "JOB",
       "EDLEVEL", "SEX", "BIRTHDATE",
       "SALARY", "BONUS", "COMM"
  FROM DSN8710.EMP
 WHERE ABS(MOD("EDLEVEL",2))=0
   FOR FETCH ONLY
.
.
.
DB2_8: Executed:
SELECT "EMPNO", "FIRSTNME", "MIDINIT", "LASTNAME",
       "WORKDEPT", "PHONENO", "HIREDATE", "JOB",
       "EDLEVEL", "SEX", "BIRTHDATE",
       "SALARY", "BONUS", "COMM"
  FROM DSN8710.EMP
 WHERE (ABS(MOD("EDLEVEL",2))=1 OR
       "EDLEVEL" IS NULL)
   FOR FETCH ONLY
.
.
```

```
.
DB2:  Thread 1 contains 32 obs.
DB2:  Thread 2 contains 10 obs.
DB2:  Threaded read enabled. Number of threads created: 2
NOTE: There were 42 observations read from the data set
      DB2LIB.EMP.
```

Stored Procedure Support

SAS 9.1 provides full support for stored procedures. DB2 Version 6.1 or later is required. Input parameters can be passed as literals or by using SAS macro variables. A stored procedure result set can be stored in a SAS table for later processing. When using SAS, only one result set can be returned by the stored procedure. Stored procedures can also be invoked at a remote location by using a three-part name in the CALL statement, in which the first qualifier represents the name of the remote location as defined in the SYSIBM.LOCATIONS table.

The syntax of the call is case sensitive; if lowercase or mixed case is used to define the schema for the stored procedure, double quotes are used in the SAS code to avoid uppercase translation. For example:

The call is passed to the database as "**CALL SAS.STORPROC** …"

```
proc sql;
      connect to db2 (ssid=db2a);
      select * from connection to db2
              (call sas.storproc(000010));
 quit;
```

The call is passed to the database as "**CALL sas.STORPROC** …"

```
proc sql;
     connect to db2 (ssid=db2a);
     select * from connection to db2
             (call ||sas=.storproc(000010));
quit;
```

The following examples show how to invoke a stored procedure named SAS.STORPROC that produces a result set that is based on the following SQL statement.

```
SELECT FIRSTNME,
       LASTNAME,
       WORKDEPT,
       JOB,
       SALARY,
       BONUS
 FROM DSN8710.EMP
 WHERE EMPNO = EMPLOYEE
    OR EMPNO LIKE EMPLOYEE
```

Example 1 Invoke a stored procedure passing an input parameter.

```
%let par = 0000%;
proc sql;
     connect to db2 (ssid=db2a);
     select * from connection to db2
             (call sas.storproc(:par));
     %put &sysdbrc;
quit;
```

Following is the output that is produced by the stored procedure.

```
FIRSTNME    LASTNAME       WORKDEPT  JOB          SALARY      BONUS
-----------------------------------------------------------------------
CHRISTINE   HAAS           A00       PRES         52750.00  1000.00
MICHAEL     THOMPSON       B01       MANAGER      41250.00   800.00
SALLY       KWAN           C01       MANAGER      38250.00   800.00
JOHN        GEYER          E01       MANAGER      40175.00   800.00
IRVING      STERN          D11       MANAGER      32250.00   600.00
EVA         PULASKI        D21       MANAGER      36170.00   700.00
EILEEN      HENDERSON      E11       MANAGER      29750.00   600.00
```

The same result can be achieved by passing the parameter directly in the call:

```
proc sql;
     connect to db2 (ssid=db2a);
     select * from connection to db2
             (call sas.storproc('0000%'));
quit;
```

Example 2 Store the result set of a stored procedure in a SAS data set.

```
%let p = 000010;
proc sql;
     connect to db2 (ssid=db2a);
     create table result_set as
     select * from connection to db2
             (call sas.storproc(:p));
quit;

proc print data=result_set;
run;
```

Following is the output that is produced by the stored procedure.

```
FIRSTNME    LASTNAME     WORKDEPT  JOB       SALARY        BONUS
-----------------------------------------------------------------
CHRISTINE   HAAS         A00       PRES      52750.00    1000.00
```

Note: When a result set is returned by a stored procedure, the column name for each field is also returned if the DB2 ZPARM configuration parameter DESCSTAT is set to YES, and the SQL statement associated with the cursor that generates the result set is static. If DESCSTAT is set to NO, each column will be named as "EXPRSSNx," where "x" is a progressive number starting with 0. The first field will not have any number at the end of the name. If DESCSTAT is set to YES and the SQL statement associated with the cursor is dynamic, column names are returned only if a DESCRIBE CURSOR statement is used in the stored procedure. Keep this in mind when a SAS data set is used to capture output of a stored procedure.

Example 3 Store the output of a stored procedure in a SAS macro variable.

The following example stores an output variable (SALARY) that is returned by the stored procedure in a SAS macro variable (X):

```
%let p = 000010;
%let x = 0;

proc sql;
     connect to db2 (ssid=db2a);
     execute (call
     sas.storproc('000010',:x)) by db2;
quit;
%put The income for the employee &p is $&x;
```

Following is the output that is produced:

```
The income for the employee 000010 is $52750
```

Note: The format of the call to the stored procedure changes, depending on whether it returns a result set. Use "execute" if you don't want to capture a result set; use "select * from connection to db2" to store the result set.

Example 4 Call a stored procedure at a remote location, and store the result set in a SAS data set.

```
%let p = 000010;

proc sql;
     connect to db2 (ssid=db2a);
     create table result_set as
     select * from connection to db2
             (call db2b.sas.storproc(:p));
quit;

proc print data=result_set;
run;
```

Following is the output that is produced by the stored procedure.

```
FIRSTNME     LASTNAME        WORKDEPT  JOB        SALARY     BONUS
-------------------------------------------------------------------
CHRISTINE    HAAS            A00       PRES     52750.00   1000.00
```

Temporary Table Support

SAS 9.1 provides support for the two types of DB2 temporary tables: GLOBAL TEMPORARY and DECLARE GLOBAL TEMPORARY.

A GLOBAL TEMPORARY table has its definition stored in the catalog. This definition can be shared among users but the content cannot be shared; all users have their own copy of it.

GLOBAL TEMPORARY tables hold their content until one of the following occurs:

- When a commit operation terminates a unit of work and no program has an open WITH HOLD cursor on the table, the commit causes a DELETE FROM <*table*>, deleting its content.
- When a rollback operation terminates a unit of work, the rollback also forces DELETE FROM <*table*>
- When the connection to the database server under which an instance of the table was created terminates, the instance is destroyed.

DECLARE GLOBAL TEMPORARY tables do not have their definition stored in the catalog and cannot be shared among users. Like a GLOBAL TEMPORARY table, this type of table is session specific. It has the following characteristics:

- If the table is defined with the ON COMMIT DELETE ROWS clause, a commit operation deletes the table content if there is not an open cursor declared by using the WITH HOLD clause.
- A rollback operation undoes the rows of the table up to the last commit or specified external savepoint, but leaves all rows that existed up to that point.
- When the application process that created the table terminates, the table is dropped.
- When a DECLARE GLOBAL TEMPORARY table is created, its *creator id* can only be SESSION.

Note: A so-called TEMP database is required to be able to create DECLARE GLOBAL TEMPORARY tables. Ask your DBA whether this database is defined at your site.

SAS 9.1 handles temporary tables by using the LIBNAME engine and the option CONNECTION=GLOBAL. The LIBNAME engine establishes the connection to DB2 that serves as an anchor to other LIBNAME statements and/or SAS procedures that want to share one or more temporary tables. The CONNECTION=GLOBAL option ensures that the same database connection is shared by all SQL statements, which is essential because temporary tables are

session-specific. A LIBNAME statement must always be specified in order to share temporary tables, even though the libref is not referenced anywhere in the code. A temporary table can either be created by using Explicit Pass-Through in PROC SQL, or by using the LIBNAME option DBMSTEMP=YES, which is new in SAS 9.1.2. Because this option is specified at the LIBNAME level, every table that is created by using that libref is created as a DECLARE GLOBAL TEMPORARY table. When DBMSTEMP=YES is used, the CONNECTION option can specify either GLOBAL or SHARED.

Example 1

```
libname db2lib db2 connection=global
        schema=SESSION ssid=db2a;

proc sql;

     connect to db2 (ssid=db2a
                     connection=global);

     execute (DECLARE GLOBAL TEMPORARY TABLE GT
              LIKE SYSIBM.SYSTABLES
              ON COMMIT PRESERVE ROWS) by db2;
     execute (INSERT INTO SESSION.GT
              SELECT * from SYSIBM.SYSTABLES
              WHERE CREATOR = 'DSN8710') by db2;
quit;

proc sql;
     select * from db2lib.gt;

quit;
```

Notice the CONNECTION=GLOBAL option is specified both in the LIBNAME statement and in the PROC SQL. Otherwise, SAS/ACCESS opens different connections for the LIBNAME engine and for PROC SQL, making it impossible to share the table.

Example 2 Reference a temporary table that is created by using PROC SQL, through Explicit Passt-hrough. Notice that the LIBNAME statement with the CONNECTION=GLOBAL option is specified to make sure that a single database connection is used for the two PROC SQL statements.

```
libname anchor db2 ssid=db2a connection=global;

proc sql;
     connect to db2 (ssid=db2a
                     connection=global);
     execute (DECLARE GLOBAL TEMPORARY TABLE GT
              LIKE SYSIBM.SYSTABLES
              ON COMMIT PRESERVE ROWS) by db2;
     execute (INSERT INTO SESSION.GT
              SELECT * from SYSIBM.SYSTABLES
```

```
              WHERE CREATOR = 'DSN8710') by db2;
     disconnect from db2;
quit;

proc sql;
     connect to db2 (ssid=db2a
                     connection=global);
     select * from connection to db2
             (select name from session.gt);
     disconnect from db2;
quit;
```

Example 3 Create, populate, and print a temporary table by using implicit pass-through.

```
options sastrace=',,,d';

libname db2lib db2 connection=shared
        DBMSTEMP=YES;

data db2lib.temptable;
     a=1;
     b=2;
run;

proc print data=db2lib.temptable;
run;
```

SAS log:

```
.
.
.
DB2_4: Executed:
DECLARE GLOBAL TEMPORARY TABLE
SESSION.TEMPTABLE (a FLOAT, b FLOAT)
ON COMMIT PRESERVE ROWS
DB2_5: Executed:
COMMIT WORK


DB2_7: Executed:
INSERT INTO SESSION.TEMPTABLE (a, b) VALUES (?,?)


DB2_8: Executed:
COMMIT WORK
.
.
DB2_10: Executed:
SELECT * FROM SESSION.TEMPTABLE FOR FETCH ONLY
.
.
```

```
                The SAS System

                   Obs    A    B

                     1    1    2
```

Why Use Temporary Tables?

The main reason for using temporary tables when working with databases is to leverage their power. A temporary table can allow a data process to be moved completely onto the database side, which avoids costly trips to pull the data out of the database in order to have SAS execute the process. The following example can help you to better understand the benefits that temporary tables give to SAS users.

This example uses a SAS data set (transaction table) to drive updates and deletes against a database table (master table). Following the example is the code that created the transaction table and the master table.

```
libname mylib db2;

/* create the master data set */
proc sql noerrorstop;
     connect to db2;
     execute (drop table emp) by db2;
     execute (commit) by db2;
     execute (create table emp
              (name        char(20),
               salary      int)) by db2;
     execute (insert into emp
              values ('Mark',   45000)) by db2;
     execute (insert into emp
              values ('Laura',  42000)) by db2;
     execute (insert into emp
              values ('Steve',  61300)) by db2;
     execute (insert into emp
              values ('Ted',    34500)) by db2;
     execute (insert into emp
              values ('George', 29750)) by db2;
     execute (insert into emp
             values ('Louise', 54500)) by db2;
quit;

/* create the transaction data set */
data upd_emp;
     name = 'Laura ';flag = 'U';output;
     name = 'Ted   ';flag = 'U';output;
     name = 'George';flag = 'D';output;
run;
```

Two different solutions are compared side-by-side in Table 6. The first solution uses a DATA step with the MODIFY statement to alter the content of the database table.

The second solution requires the addition of CONNECTION=GLOBAL to the LIBNAME statement that was specified in the preceding example, and pushes the content of the SAS transaction data set to a database temporary table. The PROC SQL step is executed to alter records in the master table.

DATA step with MODIFY Statement	PROC SQL with Temporary Table

`data mylib.emp;`
` modify mylib.emp upd_emp;`
` by name;`
` select(_iorc_);`
`  when (%sysrc(_sok)) do;`
`   if flag = 'U' then do;`
`   salary = salary+(salary*0.1);`
`   replace;`
`  end;`
`  if flag = 'D' then remove;`
 	
`  end;`
` end;`
`run;`
 	
`proc sql;`
` delete from mylib.emp`
`  where name in (select name`
`   from upd_emp`
`   where flag = 'D');`
`  select * from mylib.emp;`
`quit;`
	`proc sql;`
	` connect to db2 (connection=global);`
	` execute (update emp`
	`  set salary = salary+(salary*0.1)`
	`  where name in (`
	`   select name`
	`   from SESSION.upd_emp`
	`   where flag = 'U'`
	`   )) by db2;`
	` execute (delete from emp`
	
	`  where name in (`
	`   select name`
	`    from SESSION.upd_emp`
	`   where flag = 'D'`
	`   )) by db2;`
	` select * from connection to db2`
	` (select * from emp);`
	`quit;`

SAS Log	

`DB2_2:,Prepared:,`
`SELECT "NAME", "SALARY" FROM EMP`
`WHERE  ("NAME" ='Laura   ' ) FOR`
`UPDATE OF "NAME", "SALARY"`
 	
`DB2_3: Executed:`
`SELECT "NAME", "SALARY" FROM EMP`
`WHERE  ("NAME" ='Laura   ' ) FOR`
`UPDATE OF "NAME", "SALARY"`
 	
`DB2_4: Prepared:`
`UPDATE EMP SET "NAME" = ?, "SALARY"`
`= ? WHERE CURRENT OF DB2CUR01`
 	
`DB2_5: Prepared:`
`SELECT "NAME", "SALARY" FROM EMP`
`WHERE  ("NAME" ='Ted`
`' ) FOR UPDATE OF "NAME", "SALARY"`
	`DB2_4:,Executed:`
	`DECLARE GLOBAL TEMPORARY TABLE`
	`SESSION.UPD_EMP (name CHAR(6), flag`
	`CHAR(1)) ON COMMIT PRESERVE ROWS`
	
	`DB2_5: Executed:`
	`COMMIT WORK`
	
	`DB2_6: Prepared:`
	`INSERT INTO SESSION.UPD_EMP (name, flag)`
	`VALUES (?,?)`
	
	`DB2_7: Executed:`
	`COMMIT WORK`
	
	`NOTE: The data set MYLIB.UPD_EMP has 3`
	`observations and 2 variables.`

continued

Table 6. Comparison of Methods - Using a DATA Step with the MODIFY Statement vs. PROC SQL with a Temporary Table

DATA step with MODIFY Statement	PROC SQL with Temporary Table
DB2_6: Executed: SELECT "NAME", "SALARY" FROM EMP WHERE ("NAME" ='Ted ') FOR UPDATE OF "NAME", "SALARY" DB2_7: Prepared: SELECT "NAME", "SALARY" FROM EMP WHERE ("NAME" ='George ') FOR UPDATE OF "NAME", "SALARY" DB2_8: Executed: SELECT "NAME", "SALARY" FROM EMP WHERE ("NAME" ='George ') FOR UPDATE OF "NAME", "SALARY" DB2_9: Prepared: DELETE FROM EMP WHERE CURRENT OF DB2CUR01 NOTE: The data set MYLIB.EMP has been updated. There were 2 observations rewritten, 1 observations deleted. DB2_10: Executed: COMMIT WORK	9 10 proc sql; 11 connect to db2 (connection=global); 12 execute (update emp 13 set salary = salary+(salary*0.1) 14 where name in (15 select name 16 from SESSION.upd_emp 17 where flag = 'U' 18)) by db2; DB2_9: Executed: Update emp set salary = salary+(salary*0.1) where name in (select name from SESSION.upd_emp where flag = 'U' 19 execute (delete from emp 20 where name in (21 select name 22 from SESSION.upd_emp 23 where flag = 'D' 24)) by db2; DB2_10: Executed: Delete from emp where name in (select name from SESSION.upd_emp where flag= 'D')
SAS Output	
The SAS System NAME SALARY --- Mark 45000 Laura 46200 Steve 61300 Ted 37950 Louise 54500 **Elapsed time: 10 secs**	The SAS System NAME SALARY --- Mark 45000 Laura 46200 Steve 61300 Ted 37950 Louise 54500 **Elapsed time: 3 secs**

Table 6 (continued). Comparison of Methods - Using a DATA Step with the MODIFY Statement vs. PROC SQL with a Temporary Table

The larger the size of the master and transaction data sets, the more significant the gain in terms of elapsed time.

Frequently Asked Questions

Q: Can I determine whether DB2 or SAS is performing the query?

A: Yes. The SASTRACE option, used with either the LIBNAME statement or Pass-Through Facility, shows the SQL that's generated. When used with PROC SQL, the option also tells you which side, SAS or DB2, performed the work.

```
option sastrace=',,,d'
```

Another PROC SQL option, _METHOD, generates output to the SAS log that shows whether the query is passed to DB2.

```
libname db2lib db2 ssid=db2 authid=dsn8710;

proc sql _method;
     select *
       from db2lib.emp;
quit;
.
.
NOTE: SQL execution methods chosen are:

      sqxslct
          sqxsrc( DB2LIB.EMP ) ← not passed to DB2
```

Re-write the statement to use explicit pass-through. Following is an excerpt from the output.

```
proc sql _method;
     connect to db2 (ssid=db2a);
     select * from connection to db2;
     (select *
        from db2lib.emp);
quit;
```

This time the output shows how the query is passed to the database for execution.

```
.
.
NOTE: SQL execution methods chosen are:

      sqxslct
          sqxextr(connection to db2
          /* dbms=db2, connect options=(ssid=db2) */
          ( select * from dsn8710.emp ) )

.
.
```

Q: Can I create a partitioned table using implicit pass-through? If not, how do I create it?

A: The definition of a partitioned table is the same as the definition of a non-partitioned table. What makes a table partitioned is the table space where it is defined and the partitioning index, which must be created by using specific DDL statements and explicit pass-through. After the partitioned table space is defined, the table can be created by using either implicit or explicit pass-through. Then, its index must be created by using explicit pass-through.

```
options db2dbug;

libname db2lib db2 ssid=db2a;

proc sql;
     connect to db2 (ssid=db2a);
     execute (CREATE TABLESPACE EMP IN DSNDB04
                     USING STOGROUP DSNPBLIC
                     PRIQTY 720
                     SECQTY 720
                     ERASE NO
                     NUMPARTS 3
                     (PART 1 COMPRESS YES,
                      PART 2 COMPRESS YES,
                      PART 3 COMPRESS YES)
                     LOCKSIZE PAGE
                     CLOSE NO
             ) by db2;
     execute (CREATE TABLE EMPLOYEE
             (
              EMPNO      INTEGER       NOT NULL,
              ID         SMALLINT      NOT NULL,
              NAME       CHAR(30)      NOT NULL,
              SALARY     DECIMAL(5,2)  NOT NULL,
              DEPTNO     SMALLINT      NOT NULL
             )
              IN DSNDB04.EMP
             ) by db2;
     execute (CREATE INDEX XEMP1 ON EMPLOYEE
                    (EMPNO ASC)
                     USING STOGROUP DSNPBLIC
                     PRIQTY 720
                     SECQTY 720
                     ERASE NO
                     CLUSTER
                     (PART 1 VALUES(500),
                      PART 2 VALUES(1000),
                      PART 3 VALUES(1500))
                     CLOSE YES
             ) by db2;
     disconnect from db2;
quit;
```

Q: How can I be sure that parallelism is used to browse a partitioned table?

A: A request sent to the database can be executed in parallel by specifying the DEGREE option, as shown in the following example.

```
libname db2lib db2 ssid=db2a
        authid=dsn8710 degree=ANY;

proc print data=db2lib.emp (keep=firstnme
     lastname job salary);
     where salary > 25000;
run;
```

Following is the output of the DB2 '-DISPLAY THREADS' commands, which shows how the above statement is executed in parallel.

```
DSNV401I  < DISPLAY THREAD REPORT FOLLOWS -,
DSNV402I  < ACTIVE THREADS -,
NAME     ST A   REQ ID           AUTHID   PLAN     ASID TOKEN
DB2CALL  T     3400 DBITEST      DBITEST  SAS91    0025  6896
         PT *     0 DBITEST      DBITEST  SAS91    0086  6901
         PT *     0 DBITEST      DBITEST  SAS91    0086  6900
         PT *     0 DBITEST      DBITEST  SAS91    0086  6899
         PT *     0 DBITEST      DBITEST  SAS91    0086  6898
         PT *     0 DBITEST      DBITEST  SAS91    0086  6897
```

Conversely, if **degree=1** is used, the output of the DB2 '-DISPLAY THREADS' commands clearly shows how parallel processing has been disabled for the statement.

```
DSNV401I  < DISPLAY THREAD REPORT FOLLOWS -,
DSNV402I  < ACTIVE THREADS -,
NAME     ST A   REQ ID           AUTHID   PLAN     ASID TOKEN
DB2CALL  T     5617 DBITEST      DBITEST  SAS91    0025  6925
```

The DEGREE=ANY option cannot generate parallel access if parallelism is disabled at your site. Parallelism for dynamic SQL statements is controlled through the DB2 configuration file (ZPARM), the RLF (Resource Limit Facility), and the buffer pool settings. Ask your DBA whether you can exploit this feature at your site.

Q: Even though stored procedure support is being provided starting with SAS®9, is there any way that I can call a stored procedure and collect its result set using SAS 8.2?

A: Yes. You can create a UDF (User-Defined Function) in the database that can either return a value or a table. The UDF must be invoked in a SELECT statement that references a dummy table using explicit pass-through, as in the following example.

```
libname db2lib db2 ssid=db2a;

proc sql;
```

```
     connect to db2 (ssid=db2a);
     create table results as
     select * from connection to db2
     (select sas_date(current date)
        from sysibm.sysdummy1);
quit;

proc print data=results;
run;

  The SAS System

Obs        EXPRSSN

 1           16317
```

World Headquarters
and SAS Americas
SAS Campus Drive
Cary, NC 27513 USA
Tel: (919) 677 8000
Fax: (919) 677 4444
U.S. & Canada sales:
(800) 727 0025

SAS International
PO Box 10 53 40
Neuenheimer Landstr. 28-30
D-69043 Heidelberg, Germany
Tel: (49) 6221 4160
Fax: (49) 6221 474850

www.sas.com

 317360US.705