



THE  
POWER  
TO KNOW.

---

*Technical Paper*

## Tactics for Pushing SQL to the Relational Databases

*December 2008*

---

---

## Table of Contents

---

<b>Introduction .....</b>	<b>1</b>
Audience for this Paper .....	1
Purpose of the Paper .....	1
Scope of This Paper .....	1
Overview .....	1
Background for This Paper .....	2
Important Concepts .....	3
SAS/ACCESS® Software .....	3
SAS/ACCESS® Interface to ODBC .....	3
SAS/ ACCESS Interface to OLE DB .....	3
Databases, Relational Databases, and Relational Database Management Systems (RDBMS) .....	4
Using SAS to Analyze Relational Database Information .....	4
SAS/ACCESS Connectivity via LIBNAME Statements .....	5
Standard SAS/ACCESS Connectivity versus Java Database Connectivity (JDBC) .....	5
<b>Establishing SAS and Relational Database Connectivity .....</b>	<b>6</b>
Verify Compatibility of the Versions of the Operating System, the Relational Database, and SAS .....	6
Relational Database Server .....	6
Network .....	7
SAS® System .....	7
Connect to the Relational Database with a Simple Relational Database Client Tool .....	7
Oracle Connectivity .....	8
DB2 UDB Connectivity .....	8
Teradata Connectivity .....	9
Connect to the Relational Database with SAS .....	9
SAS to Oracle Connectivity .....	9
SAS to DB2 Connectivity .....	10
SAS to Teradata Connectivity .....	10
Common Problems You Might Encounter .....	10
Run a SAS Verification Program .....	10
Testing Connectivity to a Relational Database Using a LIBNAME Statement .....	11
PROC SQL “connect to” .....	11
Encoding Passwords .....	12
Determining Relational Database Client and Server Versions .....	12

Oracle Client and Server Versions.....	12
DB2 Client and Server Versions .....	13
Teradata Client and Server Versions.....	13
<b>The Basics: Things to Consider .....</b>	<b>15</b>
Introduction.....	15
Minimizing Data Transfer.....	15
Limiting Columns.....	15
Limiting Rows.....	17
Improving Throughput by Using Multi-Row Operations.....	20
Using the READBUFF Option.....	20
Multi-Row Output Techniques.....	21
Using the INSERTBUFF Option.....	21
Using the UPDATEBUFF Option .....	23
Restricting SAS Functions.....	23
Selecting the SAS Function That SAS/ACCESS Can Pass to the Database...	23
Using Expressions Instead of Functions.....	25
Assigning a SAS Function Value to a SAS Macro When Possible.....	26
Constructing a WHERE Clause Correctly .....	28
Sorting in the Relational Database .....	29
Naming Issues.....	30
SAS Names and Support for Database Names.....	30
PROC SQL Reserved Words.....	32
Database Reserved Words.....	32
<b>Differences in Behavior .....</b>	<b>32</b>
NULL Values and MISSING Values .....	32
Matching Differences with NULL or MISSING Values.....	33
NULL and MISSING Sort Order.....	35
Character MISSING Represented as SAS Blank String and Its Ramifications	35
SAS Special MISSING Values.....	36
Repeatable Observation or Row Order.....	36
Repeatable Observation Order and SAS/ACCESS Spooling.....	37
Understanding the Differences between SAS and Database Native Data Types...	38
Differences between Database and SAS Data Types.....	38
Appropriate Choices for Database Columns that Contain Large Numeric Values	39
Handling Large Numeric Key Columns with SAS .....	39
How Numeric Precision Can Affect WHERE clause or Join Evaluation .....	45
SAS Formats and SAS/ACCESS Default Formatting Rules.....	46
Relational Database VARCHAR Types .....	48
Date, Datetime, and Other Temporal Values.....	49

Large Object Data Types .....	52
<b>Basic Tracing.....</b>	<b>53</b>
Trace Information Shows Where Work Performed and Amount of Time Taken.....	53
Basic SASTRACE SQL Trace .....	53
SASTRACE Threaded Read trace.....	57
<b>Implicit Pass-Through .....</b>	<b>58</b>
Introduction .....	58
SAS and SQL Generation .....	58
SQL Generation through SAS/ACCESS.....	59
Using SAS PROC SQL .....	62
Criteria for Implicit Pass-Through .....	64
Select DISTINCT.....	65
Query Contains an SQL Aggregate or COALESCE Function .....	66
Query Containing a GROUP BY Clause.....	66
Query Containing a HAVING Clause.....	66
Query Containing an ORDER BY Clause .....	66
Query Contains an SQL Join .....	66
Query Involves a Set Operation.....	68
Query with a WHERE clause Containing a Subquery .....	68
Inhibitors to Implicit Pass-Through .....	69
Heterogeneous Queries .....	69
Queries that Incorporate Explicit Pass-Through Statements.....	69
Queries that Use SAS Data Set Options .....	70
Queries that Contain the SAS OUTER UNION Operator .....	70
Specification of a SAS Language Function that is not Mapped to a Database Equivalent by the Engine .....	70
Queries That Require REMERGING in PROC SQL.....	70
Specification of ANSIMISS or NOMISS Keywords in the Join Syntax .....	70
Specification of the DIRECT_SQL= LIBNAME Option .....	70
Identifying Other Inhibitors to Implicit Pass-Through.....	71
Using SAS/ACCESS LIBNAME Options That Indicate Member-Level Controls are wanted.....	72
Implicit Pass-Through Back-down Behavior.....	72
Guidelines for Writing SQL for Implicit Pass-Through.....	74
Avoid SAS Specific SQL Language Extensions .....	74
Database-Specific Syntax Limitations or Requirements.....	79
Casual Use of External SQL Views .....	81
<b>Explicit Pass-Through .....</b>	<b>83</b>
Preliminary Considerations.....	83
Managing connection activity .....	83

When to Use Explicit Pass-Through.....	84	
For Database Operations That a SAS/ACCESS Engine Does Not Support ....	84	
For Database-Specific SQL Syntax Extensions .....	85	
For Database Processing That the Implicit Pass-Through Facility Does Not Translate		86
<b>Heterogeneous Joins.....</b>	<b>89</b>	
Uploading and Joining in the Database.....	90	
Using Temporary and Permanent Tables to Upload and Join in the Database	90	
Factors Influencing Choice of Database Temporary Table or Database Permanent Table for		
Upload and Join .....	91	
Optimizing Load of the Upload Table.....	92	
Upload and Join in the Database with Data Integration Studio .....	92	
Examples of a Join on a Single Numeric Column.....	93	
Using Key Subsetting Equijoins.....	96	
Using Database Indexes in Key Subsetting Equijoins .....	97	
MULTI_DATASRC_OPT= Option.....	98	
DBINDEX= Option .....	99	
DBKEY= Option .....	100	
<b>Advanced Topics .....</b>	<b>101</b>	
WHERE Clause Processor versus the Implicit Pass-Through Facility.....	101	
Invoking the WHERE Processor .....	101	
How the WHERE Processor Relates to Implicit Pass-through .....	101	
Determining Whether the WHERE Processor Has Passed a WHERE Clause	102	
For more information, see the SASTRACE Threaded Read trace section.....	102	
Constraints the WHERE Processor Shares in Common with Implicit Pass-Through		102
Differences between the WHERE Processor and Implicit Pass-through.....	104	
Coding Efficient WHERE Clauses For the WHERE Processor .....	106	
Preventing Unneeded SQL Dictionary Queries.....	106	
Bulk Loading the Relational Database .....	108	
Why Bulk Load? .....	109	
Bulk Load Overhead Cost.....	109	
Complexities of Bulk Load .....	109	
Bulk Load Performance.....	113	
Threaded Reads .....	117	
Setting Performance Expectations.....	117	
Automatic Threaded Reads Versus User-Controlled Threaded Reads.....	117	
Understanding When Automatic Threaded Reads Occur.....	118	
Controlling Threaded Reads .....	119	
Determining Whether a Threaded Read Occurred .....	121	
For more information, see the “SASTRACE Threaded Read trace ” section of this paper.		121

Assessing Threaded Read Performance .....	122
Optimizing Threaded Reads .....	123
Understanding the SAS/ACCESS Database Connection Model .....	126
Why Update and Insert Processes on the Database are Segregated.....	127
SAS/ACCESS Utility Database Connection.....	127
Using SAS/ACCESS LIBNAME Options to Manage Database Connections.	128
Options That Affect SAS/ACCESS Utility Connections .....	129
Links for More Information .....	129
<b>Contributing Experts .....</b>	<b>131</b>

---

## Introduction

---

### Audience for this Paper

---

This paper is written for developers, database administrator, and project leads who implement ETL jobs to access data, write data, or create objects in Relational Database Management Systems (RDBMS).

Developers, database administrators, and testers contend with the abundance of operating system and relational database server releases, relational database client software, and the combinations supported by SAS. The **SAS/ACCESS® Validation Application** Web site (<http://support.sas.com/matrix>) provides a database and operating system tool that enables you to navigate the releases and combinations that are supported by SAS.

### Purpose of the Paper

---

This paper is a definitive study for using SAS with third-party relational databases such as Oracle, DB2, and Teradata. While the examples in this paper show these three relational databases, the concepts and techniques that are presented are intended to be relevant to other relational databases unless specified. In general, this paper assumes that you have a powerful relational database and wants SAS software to maximally leverage the relational database by performing data-intensive operations in the relational database. This involves:

- Pushing more processing to the relational database
- Minimizing data transfers
- Optimizing unavoidable data transfers

It also looks at the tradeoffs of performing work primarily by using SAS versus work that is primarily performed by using the relational database.

### Scope of This Paper

---

This paper is applicable to SAS® 9.1.3 and later releases. It is applicable to SAS® Data Integration Studio.

Although not specific to particular operating system and relational database releases, paper examples often use the UNIX operating system, in particular AIX. The paper does not cover the mainframe.

### Overview

---

SAS software should leverage *as appropriate* the relational database.

There are two components to leveraging the relational database investment:

SAS software should be implemented to easily pass work to the relational database. The primary components often involved in such implementations are SAS/ACCESS engines and the SQL procedure.

SAS solutions such as SAS Data Integration Studio should try to take advantage of SAS features that engage relational database-based processing.

Some data transfer is always required between the relational database and SAS. Even a SAS solution that has been written to perform with a relational database- can require large transfers because of the volumes of data that can be stored in the relational database. This paper presents recommendations to minimize data traffic and the fundamentals of optimizing necessary traffic. Optimization is an art as well as a science; some tips might demonstrate performance gain, while others require experimentation.

## Background for This Paper

---

Very large data volumes are a fact of life. ETL, by nature, deals with very large volumes. Ad hoc work regularly deals with very large data volumes. In the context of using SAS with relational database data, operations on relational database data can cause substantial or even catastrophic performance problems. Here are some examples of common classes of activity that can instigate performance problems:

- Large extracts from relational database tables into SAS; transferring billions of rows takes a lot of time.
- Large writes to relational database tables. Creating large relational database tables, appending many rows to existing tables, and large-scale updates and deletes are expensive.

SAS table operations normally operate a row-at-a-time. Updating a relational database table from a SAS table can be a row-at-a-time operation, as contrasted to a consolidated table-to-table update between two tables residing on the relational database.

However, well-crafted SQL can perform extremely well and leverage the relational database capabilities. Large data volumes and the relatively high cost of data transfer between the relational database and SAS mean that SQL tuning and appropriate SAS options can further optimize a well-crafted job.

Writing SAS code that performs well with relational database data requires the use of SQL. PROC SQL queries can be coded either as:

- SAS SQL syntax that uses SAS libref references to relational database tables.
- Explicit relational database SQL syntax using the SQL pass-through facility.

The pass-through syntax hides the relational database SQL from SAS parsing. It is sent "as is" to the relational database. A single SQL syntax for all data sources and the ability to use SAS language extensions are the hallmarks of SAS SQL. SAS SQL queries are not passed "as is" to a relational database. When an SQL query is not passed in entirety to a relational database, some processing occurs in SAS. These operations can involve larger data transfers into SAS and often do not perform as expected. A central point of this paper is to demonstrate how to write SAS SQL syntax that performs well against relational databases.



For more details, see the Explicit Pass-Through

section.

## Important Concepts

---

### **SAS/ACCESS® Software**

The SAS/ACCESS family of interfaces enables transparent access and manipulation of database data. SAS/ACCESS harnesses database functionality and enhances it with the power of the SAS language and functions. Where possible, SAS/ACCESS converts SAS functionality to equivalent database functionality. For optimal throughput and access to all database functionality, many SAS/ACCESS interfaces call database vendor-supplied native libraries: OCI for Oracle, CLI for DB2, CLIV2 for Teradata, and so forth. Not all SAS/ACCESS interfaces call native libraries.

In SAS 9.1 and later, SAS/ACCESS enhances read performance with threaded reads and load performance with native bulk loading. Threaded reads retrieve a result set on multiple connections to the relational database. Separate SAS threads simultaneously read a portion (partition) of the result set from each connection, and thus the name “threaded reads.” Threaded reads are covered extensively in SAS online documentation.

SAS uses a single writer. However, bulk load into the relational database can replace row-at-a-time SQL INSERT. Large groups of rows are bulk loaded as a single unit and the performance enhancement can be an order of magnitude or more, depending on the particular relational database and particular bulk load method that is used.

### **SAS/ACCESS® Interface to ODBC**

ODBC is an industry standard protocol for communication between an ODBC-compliant application and a relational database. The SAS/ACCESS Interface to ODBC:

- Communicates indirectly with a relational database via an ODBC driver that is specific to the database.
- Supports threaded read operations for any driver; supports bulk loading to SQL Server.
- Can be used to access data when there is no native SAS/ACCESS interface engine available.

### **SAS/ ACCESS Interface to OLE DB**

OLE DB from Microsoft is an Application Programming Interface (API) for access to data in database tables, e-mail files, text files, and other file types.

The SAS/ACCESS interface to OLE DB:

- Accesses data from these sources through OLE DB data providers such as Microsoft Access, Microsoft SQL Server, and Oracle.
- Provides a facility for accessing OLE DB for OLAP data by allowing you to pass MDX (Multidimensional Expressions) to access the OLAP data.

- Does not support threaded reads; supports bulk loading to SQL Server.

## Databases, Relational Databases, and Relational Database Management Systems (RDBMS)

Databases store, manage, and retrieve information through the use of tables. Using a common identifier (key) between tables, you “relate” tables to one another to form a Relational Database (RDB).

Applications access, search, and manipulate information in the database by using a special programming language known as structured query language (SQL). SQL can also be used to add, delete, and modify information in tables. The SQL standard is supported by all major databases, although implementation of the full standard is not guaranteed. Each database also extends standard SQL with database-specific extensions that are valid only for that database.

RDBMS add numerous features to ensure that the database handles enterprise-wide requirements:

**Optimized SQL** – SQL is made to work optimally with the structure of the specific RDBMS.

**Stored Procedures** – Common SQL commands are compiled into a store procedure for secure encapsulation and faster execution.

**Security** – A special set of internal tables called “system tables” or “SQL dictionary” contains access permissions for users and databases. This enables a database administrator (DBA) to vary permissions in order to keep data secure.

**Monitoring and management tools** – These help the DBA monitor and understand SQL usage and determine ways to increase efficiency.

**Scalability** – Modern databases manage multiple high-speed processors, clustered servers, high bandwidth connectivity, and fault-tolerant storage technology.

**Back up and recovery** – Enables periodic backup of the entire database, and restoration if there is catastrophic failure.

**Auditing and roll-back recovery** – These methods track changes to tables in the database and enable rolling back changes and recover the previous state if something unexpected occurs during a process such as ETL.

Databases are typically targeted to support Online Transactional Processing (OLTP) or data warehousing. OLTP is characterized by large numbers of simple queries and lightweight data modifications (singleton inserts/updates/deletes). A data warehouse expects smaller numbers of queries and data modifications. However, the queries are often complex (for example, “strategic queries”) and data modifications large (for example, periodic bulk loads of large data volumes). Due to their, OLTP systems, and warehouses are structured differently.

## Using SAS to Analyze Relational Database Information

Many companies want one instance of the “truth” regarding their corporate data. Because the RDBMS is accessible to any application supporting SQL, storing corporate data in an RDBMS makes sense. Because SAS SQL, is suitable for analyze information that is stored in an RDBMS.

The level of data analysis, and whether the database is OLTP or a warehouse, affects how an application best uses the RDBMS. Whether OLTP or a warehouse, basic reporting is usually best expressed in SQL. Complex reporting, analytics, and modeling usually warrant extracting OLTP data to the application, possibly creating a separate data mart, as OLTP data is not well structured for these operations. Complex reporting, analytics, and modeling can be performed in a database warehouse; however, SAS complex reporting, analytics, and modeling might have specific needs where ETL to an optimally structured data mart is preferable.

For basic reporting, the key is to do maximum processing in the RDBMS and transfer minimal data to SAS. ETL for complex reporting, analytics, and modeling should perform as much pre-processing as possible in the database and transfer minimal data to SAS.

In summary:

For more basic information retrieval, sending SQL statements to the RDBMS to be processed internally in the RDBMS (using the power of the RDBMS) might be most efficient.

For analytics and or modeling, you might find that the information is not well structured in the RDBMS. It might be best to move the data into SAS data files and restructure it.

### **SAS/ACCESS Connectivity via LIBNAME Statements**

Most relational databases use a client/server model. In the context of the relational database, the relational database itself is the server. The client software API, whether native, ODBC, or Java, is usually supplied by the relational database vendor and is often free of charge.

The SAS client uses an API to interact with the relational database. The SAS/ACCESS interfaces provide connectivity to the API through a SAS LIBNAME statement. For example, the SAS/ACCESS Interfaces to Oracle, DB2, and Teradata call relational database vendor-supplied native APIs. The SAS/ACCESS Interface to ODBC uses relational database or third party (for example, DataDirect) supplied ODBC drivers. For another example, when you issue a SAS libref to the Oracle database, code in the SAS/ACCESS Interface to Oracle engine calls Oracle OCI.

### **Standard SAS/ACCESS Connectivity versus Java Database Connectivity (JDBC)**

Java code written at SAS interacts with a relational database indirectly through a SAS Workspace or a SAS Stored Process server. In this context, SAS/ACCESS APIs are invoked and native or ODBC drivers must be installed along with SAS/ACCESS. Java code can also directly access a relational database via Java Database Connectivity (JDBC). There are pros and cons to either technique; slow throughput and high overhead for Java object creation and cleanup can be negatives to JDBC access.

---

## Establishing SAS and Relational Database Connectivity

---

Before examining the concepts, advices, and practices of pushing SQL to a relational database, verify that SAS and the relational database that you are working with is correctly installed and configured. Perform the following tasks to ensure that the software is properly configured and that SAS is interacting correctly with the relational database:

Verify that you have compatible versions of the operating system, the relational database, and SAS software.

- Connect to the relational database with a simple relational database client tool.
- Connect to the relational database with SAS.
- Run a SAS verification program.

Instructions and resources for completing these steps are given in the following sections.

---

### Verify Compatibility of the Versions of the Operating System, the Relational Database, and SAS

---

On UNIX and Windows, SAS interacts with the relational database in a client/server manner. Usually the relational database server runs on one machine and SAS on another machine. The relational database client software must be installed on the same machine as SAS, and SAS uses the relational database client software to communicate with the relational database. Being on machines, SAS, and the relational database server communicate across a network. The following diagram shows the SAS client machine on the left with a TCP/IP network connection to a relational database server. The required software is listed below each machine in *italics*.

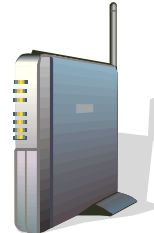
#### SAS System



*operating system*  
*relational database client software*  
*SAS software*

*TCP/IP network*

#### Relational Database Server System



*operating system*  
*relational database server*

### Relational Database Server

You can assume that the correct software is deployed on the relational database server. Whoever installed the relational database should have verified its compatibility with the local operating system. If you have reason to suspect incompatibility, contact your database administrator.

## Network

TCP/IP networks are the standard. Again, you assume correct installation of the network, including the adapters or “cards” that connect both machines to the network. If one machine can successfully “ping” the other, network connectivity is likely correct.

## SAS® System

The relational database client software must be compatible with the operating system. If the relational database client software was properly installed, compatibility with the operating system is likely. Again, if you have reason to suspect incompatibility might be an issue, contact the site database administrator.

The relational database client software must be compatible with the (remote) relational database server edition. Do not assume this compatibility. Compare the client software version to the relational database server version. A match is preferred. If the client is a later (usually higher number) version than the relational database server, the two are likely compatible. However, a client older than the relational database server is a warning flag; if the versions are “close” (major versions the same), the combination might be okay.

Otherwise, upgrade the client to match the server or research to ensure compatibility. For example, Teradata presents client/server compatibility on the **Teradata FAQ** Web site: (<http://www.teradata.com/t/page/118779/index.html#whatis>).

Read the section [Determining Relational Database Client and Server Versions](#) later in this paper.

SAS software must be compatible with the local operating system. SAS System Requirements documentation for the supported operating system versions can be found on the **SAS Install Center** Web site (<http://support.sas.com/documentation/installcenter/>). Select your operating system, select the appropriate SAS release, then select “System Requirements”.

SAS software must be compatible with the relational database client software. This information is found on the same SAS Web site as the SAS and operating system compatibility. Open the **SAS Install Center** Web site (<http://support.sas.com/documentation/installcenter/>), select your operating system, select the appropriate SAS release, select System Requirements.

The **SAS/ACCESS Validation** Web site (<http://support.sas.com/matrix>) enables you to compare the operating system level and SAS version to the relational database product and its version to determine compatibility.

## Connect to the Relational Database with a Simple Relational Database Client Tool

---

SAS does not connect to the relational database if a simple relational database client tool cannot. Connecting with a client tool confirms that the relational database client software is correctly installed and configured to communicate with the relational database.

This section presents examples of testing connections with Oracle, DB2, and Teradata.



**Note:** SAS/ACCESS does not use or call these relational database tools. SAS uses the same lower-level relational database client components that are used by these tools.

## Oracle Connectivity

Use Oracle SQLPlus. In this example, *dbserver* is an Oracle server that is configured in the Oracle *tnsnames.ora* configuration file. *oracleuser/oracle\_passwd* is an example of a valid user name and password pair on the *dbserver* server.

From an operating system command prompt, issue:

```
C:\>sqlplus oracleuser/oracle_passwd@dbserver

SQL*Plus: Release 9.2.0.1.0 - Production on Thu Jan 26 19:39:55 2006

Copyright © 1982, 2002, Oracle Corporation. All rights reserved.
```

```
Connected to:
Oracle9i Enterprise Edition Release 9.2.0.6.0 - 64bit Production
With the Partitioning, OLAP and Oracle Data Mining options
Jserver Release 9.2.0.6.0 - Production
```

If *dbserver* is not configured correctly in *tnsnames.ora*, you usually see the following message:

```
ERROR:
ORA-12154: TNS:could not resolve service name
```

## DB2 UDB Connectivity

Use the DB2 UDB Command Line Processor (CLP). In this example, “sample” is a DB2 server that is configured with the DB2 UDB configuration assistant. *db2* and *db2\_passwd* is a valid user name and password pair on the *sample* server.

At a UNIX command prompt, issue:

```
$ db2
```

or, from a Windows command prompt, issue:

```
C:\> db2cmd db2
```

Next, at the *db2 =>* prompt, issue:

```
db2 => connect to sample user db2 using db2_passwd

Database Connection Information

Database server          = DB2/SUN64 8.2.3
SQL authorization ID    = DB2
Local database alias    = SAMPLE
```

If “sample” is not configured correctly, you usually see the following message:

```
SQL1013N The database alias name or database name "SAMPLE" could not be found.
SQLSTATE=42705
```

## Teradata Connectivity

Use Teradata BTEQ. In this example, *teradata\_server* is a Teradata server configured in the operating system hosts file as *teradata\_server1*. *teradata\_user* / *teradata\_passwd* is a valid user name and password pair on the *teradata\_server* server.

From an operating system command prompt, issue:

```
C:\>bteq
```

```
Teradata BTEQ 08.02.01.00 for WIN32.
Copyright 1984-2003, NCR Corporation. ALL RIGHTS RESERVED.
```

Type your logon or BTEQ command:

```
.logon teradata_server/teradata_user

.logon teradata_server/teradata_user
Password: (type your password here, teradata_passwd in this example)

*** Logon successfully completed.
*** Transaction Semantics are BTET.
*** Character Set Name is 'ASCII'.
```

If *teradata\_server* is not configured correctly in the hosts file, you usually see the following message:

```
*** CLI error: MTDP: EM_NOHOST(224): name not in HOSTS file or names database.

*** Return code from CLI is: 224
*** Error: Logon failed!
```

## Connect to the Relational Database with SAS

---

Connect to the relational database with a correctly formatted LIBNAME statement that uses the same server names, user names, and passwords as shown in the preceding examples. After connecting successfully with the relational database client tool, you need to perform the following tasks in order to connect with SAS:

1. Install Base SAS® and SAS/ACCESS for the appropriate relational database. Set any required operating-system specific environment variables so that SAS/ACCESS modules can find the relational database client libraries that they use. At the SAS Install Center Web site (<http://support.sas.com/documentation/installcenter/>), select your operating system, select the appropriate SAS release, and open the Configuration Guide, and search for your relational database.



**Note:** Contact your database administrator for the appropriate paths.

2. Carefully transcribe server and user and password references from your successful relational database client tool connection into an appropriately formatted SAS LIBNAME statement. Examples follow.

### SAS to Oracle Connectivity

```
libname test oracle path=dbserver user=oracleuser password=oracle_passwd;
NOTE: Libref TEST was successfully assigned as follows:
```

```
Engine:          ORACLE
Physical Name:  dbserver
```

### SAS to DB2 Connectivity

```
libname test db2 datasrc=sample user=db2 password=db2_passwd;
NOTE: Libref TEST was successfully assigned as follows:
Engine:          DB2
Physical Name:  sample
```

### SAS to Teradata Connectivity

```
libname test teradata server=teradata_server user=teradata_user password=
teradata_passwd;
NOTE: Libref TEST was successfully assigned as follows:
Engine:          TERADATA
Physical Name:  teradata_server
```

### Common Problems You Might Encounter

If your SAS install did not include the appropriate SAS/ACCESS product, you might see a message similar to the following Oracle example:

```
ERROR: The SAS/ACCESS Interface to ORACLE cannot be loaded. ERROR: Module
SASORA not found in search paths.
```

If the appropriate operating-system specific environment variable does not point to the relational database client libraries, you might see a message similar to the following Oracle example:

```
ERROR: The SAS/ACCESS Interface to ORACLE cannot be loaded. ERROR: Image
SASORA found but not loadable.
```

If you see these errors, perform the tasks presented in the previous sections: “Verify Compatibility of the Versions of the Operating System, the Relational Database, and SAS ” and “Connect to the Relational Database with a Simple Relational Database Client Tool”, and contact the site database administrator for assistance.

If other errors appear, open the **SAS Technical Support Knowledge Base** Web site (<http://support.sas.com/resources/>) and search for the error in the samples and SAS notes database.

### Run a SAS Verification Program

---

SAS/ACCESS software installation populates a sample relational database verification program in the *dbi* subdirectory of the *samples* directory of the SAS installation. This program performs a variety of basic operations, thus demonstrating the SAS to relational database connection that you have established. Complete the following tasks to test this connection:

1. Create a scratch directory, then ‘cd’ to it so it is your working directory.
2. Copy the following files to the working directory:



```
accauto.sas
```

```
accddata.sas
```

```
accrun.sas
```

3. Modify *accauto.sas* so that the LIBNAME statements execute without error. (The “mydblib” libref connects you to the relational database server.)

4. Create a new SAS program that contains the following code:

```
%include accauto/source2;
%include accdata/source2;
%include accrun/source2;
```

Execute this program, and check the SAS log for errors. A few DEBUG error messages are expected. An otherwise - log means that SAS is communicating and interacting correctly with the relational database. Any errors should be investigated.



**Note:** Successfully running this verification program provides a high level of confidence that all software components are correct. However, it does not guarantee that you will proceed problem-free. Though unusual, it is possible to later encounter problems between SAS and the relational database. If you do encounter problems later after successfully running this verification program, first verify that the correct combination of operating system, relational database, and SAS software is present, then contact SAS Technical Support.



**Caution:** The SAS verification program drops and creates tables in the relational database. Run the verification program against a relational database test account, preferably an account that does not have tables. You need CREATE TABLE access in the account. The tables that SAS creates are extremely small, so a very resource-constrained test account is appropriate.

### Testing Connectivity to a Relational Database Using a LIBNAME Statement

You have already seen examples of valid LIBNAME statements. With appropriate system options or environment variables set, connecting can be made simpler, for example:

```
LIBNAME mylib ORACLE;
```

or, you can specify connection and other options in the LIBNAME statement.

```
libname mydblib oracle user=oracle_user password=oracle_passwd path=myorapath
insertbuff=50;
```

### PROC SQL “connect to”

The explicit pass-through facility of PROC SQL sends statements unmodified to the relational database. An alternative to the LIBNAME statement, the explicit pass-through facility enables you to issue relational database-specific syntax that cannot be generated with the LIBNAME statement. You can issue most SQL supported by your relational database. The CONNECT and DISCONNECT statements establish and terminate connections to the database.

```
PROC SQL;
connect to oracle as mycon (user=oracle_user
    password=oracle_passwd path='myorapath');
```

```

create view samples.hires88 as
  select *
    from connection to mycon
      (select empid, lastname, firstname,
         hiredate,salary
       from employees where
         hiredate>='31-dec-88');

disconnect from mycon;

```

## Encoding Passwords

Password visibility in the LIBNAME or the CONNECT statement can be a security issue. You can use the PWENCODE procedure to encode passwords. Encoded passwords are used in place of plain-text passwords. When the following statement is executed:

```
PROC PWENCODE IN='mypassword' METHOD=dbserver;
an encoded password is generated, for example:
```

```
{dbserver}bXlwYXNz
```

This replaces the visible password:

```
libname mydblib oracle user=oracle_user password='{dbserver}bXlwYXNz' path=myorapath;
```

## Determining Relational Database Client and Server Versions

---

As discussed in the “Verify Compatibility of the Versions of the Operating System, the Relational Database, and SAS” section of this paper, it is important to verify compatibility of the relational database client software and the relational database server. Using previous examples, here is how to determine relational database client and server versions.

### Oracle Client and Server Versions

In the following example, the Oracle client version is bolded and underlined. The server version (Oracle 9.2) is bolded, underlined, and italicized.

```

C :\>sqlplus oracleuser/oracle_passwd@dbserver

SQL*Plus: Release 9.2.0.1.0 - Production on Thu Jan 26 19:39:55 2006

Copyright © 1982, 2002, Oracle Corporation. All rights reserved.

```

```

Connected to:
Oracle9i Enterprise Edition Release 9.2.0.6.0 - 64bit Production
With the Partitioning, OLAP and Oracle Data Mining options
Jserver Release 9.2.0.6.0 - Production

```

## DB2 Client and Server Versions

The DB2 client version is bolded and underlined. The server version (DB2 8.2) is bolded, underlined, and italicized.

```
C:\> db2cmd db2
```

```
(c) Copyright IBM Corporation 1993,2002
Command Line Processor for DB2 SDK 8.1.0
```

You can issue database manager commands and SQL statements from the command prompt. For example:

```
db2 => connect to sample
```

For general help, type: ?.

For command help, type: ? command, where command can be the first few keywords of a database manager command. For example:

```
? CATALOG DATABASE for help on the CATALOG DATABASE command
? CATALOG           for help on all of the CATALOG commands.
```

To exit db2 interactive mode, type QUIT at the command prompt. Outside interactive mode, all commands must be prefixed with 'db2'.

To list the current command option settings, type: LIST COMMAND OPTIONS.

For more detailed help, see the Online Reference Manual.

```
db2 => connect to sample user db2 using db2_passwd
```

### Database Connection Information

```
Database server      = DB2/SUN64 8.2.3
SQL authorization ID = DB2
Local database alias = SAMPLE
```

## Teradata Client and Server Versions

The Teradata client version is bolded and underlined. The server version is bolded, underlined, and italicized. Notice that both the client and server version numbers are a subset of the entire string that is displayed.

In this example, the Teradata client (07.01) is TTU 7.1. The server (05.01) is V2R5.1. Teradata combines **recent** client and server versions under the label "Teradata Warehouse". Teradata Warehouse 7 and Teradata Warehouse 8 are the recent versions. The client shares the number of the Warehouse version. In this example, the TTU 7.1 client is from Teradata Warehouse 7.1. However, the server numbering is two less than the Warehouse number. In this example, the V2R5.1 server is from Teradata Warehouse 7.1. Our client and server versions match in this case.

```
C:\>bteq
```

```
Teradata BTEQ 08.02.01.00 for WIN32.
Copyright 1984-2003, NCR Corporation. ALL RIGHTS RESERVED.
Enter your logon or BTEQ command:
```

```
.logon teradata_server/teradata_user
```

```
.logon teradata_server/teradata_user
```

Password: (type your password here, *teradata\_passwd* in this case)

```
*** Logon successfully completed.
*** Transaction Semantics are BTET.
*** Character Set Name is 'ASCII'.
```

BTEQ -- Type your DBC/SQL request or BTEQ command:

```
.show versions
```

```
BTEQ Version 08.02.01.00 for Win 32 running Windows Sockets
BTQMain   : 08.02.01.02
BTQUtil   : 08.02.01.03
BTQResp   : 08.02.01.06
BTQParse  : 08.02.01.01
BTQNotfy  : H5_03
BTQPiom   : 08.02.00.00
BTQIOUTL  : 08.02.01.04
BTQMEMR   : 08.02.01.02
CapAAUtl  : 08.02.01.00
CapERUtl  : 08.02.00.00
CapCLUtl  : 08.02.01.02
CapIOUtl  : 08.02.01.00
CapLogW   : 08.02.00.00
CapNfy    : H3_03
CapLoadM  : 06.01.00.00
CLIV2     : 04.07.01.33
MTDP      : 04.07.01.24
MOSIoS    : 04.07.01.01
MOSIDEP   : 04.07.01.04
OSENCRYPT  : N/A
OSERR     : 04.07.01.00
PMPROCS   : 02.01.00.01
PMRWFMT   : 02.00.00.02
PMTRCE    : 02.00.00.02
PMMM      : 02.00.00.01
PMHEXDMP  : 02.00.00.01
PMUNXDSK  : 02.00.00.03
```

BTEQ -- Type your DBC/SQL request or BTEQ command:

```
select * from dbc.dbcinfo;
```

```
*** Query completed. 2 rows found. 2 columns returned.
*** Total elapsed time was 1 second.
```

InfoKey	InfoData
RELEASE	V2R. <u>05.01</u> .00.01
VERSION	05.01.00.02

---

## The Basics: Things to Consider

---

### Introduction

---

One of the main points to remember when you are accessing data stored in a relational database is that the data is in an application that runs independently of SAS. When you go to access that data, you need to either move the data into SAS so that SAS can analyze, query, or manipulate it; or you have to send SQL commands into the relational database to query or manipulate the data in the relational database and then have the results returned to SAS. With the size of relational databases continuing to increase, it is important to take time to determine the best approach to get the data within SAS, and to minimize the amount of data that is transferred between the relational database and SAS. This section presents some of these basic concepts:

- Minimize data transfer
- Improve throughput with multi-row operations
- Restrict SAS functions
- Properly construct WHERE clauses
- Sort data in SAS versus in a relational database

### Minimizing Data Transfer

---

When looking for ways to increase performance, minimizing the amount of data that is transferred between the database and SAS is the first place to start. For example, changing your PROC SQL or using the KEEP data set option to select only the columns needed from the database can have a significant impact on performance, especially if the rows are wide and contain many unused columns. Character data might need to be transposed if the data is stored in different code pages on the database client and server. Limiting the number of rows selected in a table can drastically improve response time when testing process flow while developing an application. The following sections discuss several ways to minimize data transfer from the database to SAS by limiting either the selection of columns or rows from a table.

#### Limiting Columns

Usually, when minimizing data transfer, it is a good idea to select only the needed columns in a table to work with. Wide rows that consist of many columns can create much unnecessary data transfer across the network. Even unused numeric columns have to be transformed from the database format to a SAS float value, which is the internal representation of numeric data in SAS.

This section explains how to limit the columns that are selected in a table by using the following methods:

- PROC SQL
- VAR statement
- The DROP and KEEP data set options

### ***Subsetting the Column List in PROC SQL***

Consider extracting only the database columns that are needed to meet the requirements in your job. Instead of coding `select * from tablename`, evaluate the programs that follow the selection process and determine which columns are needed. Re-code your SELECT statement to select those columns. For example, if you need only the SALARY and START-DATE columns from the EMPLOYEE table, change:

```
Select * from employee;
```

to:

```
Select salary, start-date from employee;
```

### ***Using VAR Statements***

Another way to limit the columns in the SQL SELECT statement is to use the VAR statement with a SAS procedure. In the following example, the VAR statement causes the SQL statement to be generated with only the column "column1" in the SELECT statement.

```
proc print data=mydatabase.mytable;
var column1;
run;
```

### ***Using the DROP and KEEP Data Set Options***

The SQL SELECT method works only with PROC SQL. However, you can use the DROP and KEEP data set options to prevent retrieving unneeded columns from your relational database table. Using the DROP or KEEP data set options limits the number of columns in the SELECT statement that are passed to the database.



**Note:** Because minimizing the input data volume automatically minimizes the output data volume, it is recommended that you use the DROP and KEEP data set options on input data sources, rather than output sources. If writing out to a relational database, a data set option is as efficient as a DATA step statement.

In the following example, the KEEP data set option causes the SAS/ACCESS engine to select only the SALARY and DEPT columns when it reads the MYDBLIB.EMPLOYEES table:

```
libname mydblib db2 user=user password=XXXXX database=testdb;

Data work.acc024;
set mydblib.employees(keep=salary dept);
where dept='ACC024';
run;
```

The generated SQL that is processed by the relational database is be similar to the following:

```
SELECT "SALARY", "DEPT" FROM EMPLOYEES
WHERE (DEPT="ACC024")
```

Without the KEEP data set option, all the columns would be selected from the database. Here are additional examples using the KEEP data set option:

Using KEEP to limit the input:

```
data outtable;
set intable(keep= ...

proc sort data=intable(keep = ...) out=outtable;
```

Using KEEP to limit the output:

```
data outtable (keep = ...
set intable

proc sort data=intable out=outtable(keep = ...);
```



**Caution:** Using any data set options (including DROP and KEEP) with PROC SQL disables SQL implicit pass-through.

### Limiting Rows

In addition to limiting the columns that are selected from a given table, you can significantly improve performance by limiting the number of rows that are selected in a table. This section explains how to limit row selection in a table by using:

- A WHERE clause, statement, expression
- The OBS data set option
- The INOBS and OUTOBS PROC SQL options

#### ***Using WHERE Clause to Filter Rows***

If you are selecting all rows from a table, but later in your program you limit the rows that you work with, consider limiting the number of rows you select initially from the database by using a WHERE clause filter. For example, suppose that you need only the columns SALARY and START-DATE from the EMPLOYEE table for the IT department.

Instead of selecting all the rows from the database and then looking for the IT department rows in your program, consider these two examples:

Not recommended:

```
Select salary, start-date from employee;
```

Recommended:

```
Select salary, start-date from employee where department = "IT";
```

Because using a WHERE clause does not disable SQL implicit pass-through, it is recommended that, when possible, you use a WHERE clause instead of using SAS data set options such as OBS.



**Note:** The placement of the WHERE clause in a SAS query can affect how much data is filtered. The WHERE clause might have no effect on minimizing the amount of data, depending on the join order and the relational database, until the very end of the query,

#### ***Using the OBS Data Set Option to Limit Rows***

Developing, testing, or debugging a new application is one situation in which limiting the amount of data being transferred between SAS and a relational database is beneficial. You don't need to read and transfer all the data from and to a relational database, but you do want to transfer enough data to adequately test the process and verify its design. One method of limiting rows is the OBS data set option.



**Caution:** Using any SAS data set options (including OBS) with PROC SQL disables SQL implicit pass-through. To limit data transfer when using PROC SQL statements and still permit SQL Implicit pass-through, use the INOBS and OUTOBS options. For more information, see "Using the PROC SQL Options INOBS and OUTOBS ".later in this documentation.

Use the OBS data set option in SAS to limit the number of rows that SAS accesses. Especially when used against large database tables, setting OBS to an appropriately small value can reduce workload and pool space consumption on the database, and help complete queries much faster.

While the OBS data set option is a standard SAS data set option, the OBS option is optimized on the relational database side of the process by the SAS/ACCESS engines for the following relational databases:

- Teradata
- DB2 UDB
- ODBC
- SQL Server
- MySQL
- OLE DB



The following examples show how to use the OBS option with the listed relational databases.

#### Minimizing row transfer in Teradata

For Teradata, the OBS data set option causes the SAS/ACCESS engine to add a SAMPLE clause to generated SQL. In the following example, 10 rows are printed from the table PRODUCT:

```
libname tra teradata user=xyz pass=***** database=dbc;
proc print data=product (obs=10);
run;
```

The SQL passed to Teradata is:

```
SELECT "product_id","product_code", "product_price" FROM "product" SAMPLE 10
```

#### Minimizing row transfer in DB2 UDB , ODBC, and SQL Server

DB2 UDB, ODBC, and SQL Server support limiting the result set that is created by the database by using the value that OBS is set to as the parameter in the API calls to the database. For DB2 UDB, ODBC, and SQL Server, the API call to limit the result set is:

```
SQLSetStmtAttr(SQL_ATTR_MAX_ROWS)
```

#### Minimizing row transfer in MySQL

MySQL also supports limiting the result set that is created by the database. The SAS/ACCESS engine to MySQL uses the LIMIT clause:

```
[LIMIT {[offset,] row_count | row_count OFFSET offset}]
```

For example:

```
libname x mysql user=xyz pw=***** server=myserver db=test;
proc print data=x.emp(obs=5); run;
```

You will see the SQL as:

```
MYSQL_2: Executed: on connection 2
SELECT * FROM `emp` LIMIT 5
```

#### ***Using the PROC SQL Options INOBS and OUTOBS***

Like OBS, the INOBS and OUTOBS options in PROC SQL can be used to limit the data transfer of rows of a table between SAS and a relational database. The INOBS option restricts the number of rows that PROC SQL retrieves from any single source, but does not restrict the number of rows for output operations. Restricting the rows that are retrieved for data sources by using the INOBS or the OUTOBS option can affect the results returned for SQL join operations.

The OUTOBS option restricts the number of rows output to a data source, but does not restrict the number of rows that are retrieved. The OUTOBS option can be used, for example, with an INSERT or a CREATE statement based on a query. The OUTOBS option is useful when a sampling based on the true result set of a join is needed.



**Notes:** Because using any SAS data set options such as OBS in PROC SQL statements disables SQL implicit pass-through, you can use the INOBS and OUTOBS options in PROC SQL if you do not want to disable implicit SQL pass-through.

However, the OBS data set option is optimized for performance on the relational database side in the SAS/ACCESS engines for the relational databases listed in the earlier section “Using the OBS Data Set Option to Limit Rows”.

Using the INOBS or OUTOBS options generates warnings in the logs. Therefore, it is recommended that, when possible, you restrict their use to testing or debugging.

## Improving Throughput by Using Multi-Row Operations

---

### Using the READBUFF Option

The READBUFF option improves throughput by reading multiple rows from a relational database into SAS.



**Note:** Another useful technique for reading multiple rows is using threaded reads. For more information, see the Threaded Reads section of the Advanced Topics chapter in this paper, and the SAS online documentation:


SAS Institute Inc., 2007, SAS online documentation, “SAS/ACCESS: Relational Databases: Concepts: Threaded Reads” Available <http://support.sas.com/onlinedoc/913/getDoc/en/acreldb.hlp/a002238650.htm>

The READBUFF option specifies the number of rows that can be held in memory for input into SAS from the database. The buffering is done on the database. Buffering data reads can decrease network activities and increase performance. However, because SAS stores the rows in memory, higher values for the READBUFF option uses more memory. If the data in the database is constantly changing and the value set for this option is too high, there is a risk that the rows that are returned to SAS might be out of date.

If READBUFF=1, which is the default for some SAS/ACCESS engines, only one row is retrieved at a time. The higher the value for the READBUFF option, the more rows the SAS/ACCESS engine retrieves in one fetch operation. See *Table 1* for information about support for this option.

SAS/ACCESS Interface to	READBUFF option supported	Default value or caveat
Sybase	Yes	100
Oracle	Yes	250
ODBC	Yes	If option is not set, then single-row fetch is done.
SQL Server	Yes	If option is not set, then single-row fetch is done.
DB2 UDB	Yes	If option is not set, then single-row fetch is done.
OLE DB	Yes	1


Table 1: SAS/ACCESS Interfaces and Support for READBUFF Option

 **Note:** The READBUFF option is not supported as a configurable option in all SAS/ACCESS products. For example, in the SAS/ACCESS Interface to Teradata, the optimum buffer size for input from Teradata is calculated and used automatically.

For more information on how to efficiently use the READBUFF option, see the “Increase the Read Buffer Size with the READBUFF=Option” section of this paper.

### Multi-Row Output Techniques

The INSERTBUFF and UPDATEBUFF options improve throughput by writing multiple rows from SAS into a relational database.

 **Note:** The fastest way to insert data into a relational data when using the SAS/ACCESS engine is to use the bulk loading capabilities of the database. For more information, see the examples presented in the “Bulk Loading the Relational Database” section of this paper.

### Using the INSERTBUFF Option

The INSERTBUFF option specifies the number of rows that can be held in memory for inserting into the database from SAS. The INSERTBUFF option can be specified as a data set option or as an option in the LIBNAME statement. You might need to experiment with different values for this option to determine the optimal value for your application. The optimal performance varies with network type, network traffic, and available memory in SAS and the database.

The SAS notes in the SAS log that indicate the success or failure of the insert operation might be incorrect because these notes only represent information for a single insert, even when multiple inserts are performed. Be aware, if you use this option with the DBCOMMIT option, the values should be the same for both options. If the values are not the same, the DBCOMMIT option overrides the INSERTBUFF option.

If you are inserting rows by using PROC FSVIEW or the VIEWTABLE, setting the value of the INSERTBUFF option to 1 prevents the database from trying to insert more than one row at a time and holding a lock on the table.

The INSERTBUFF option is supported by the SAS/ACCESS engines for the following products:

- DB2 UDB
- ODBC
- OLE DB
- Oracle
- MySQL
- Microsoft SQL Server

The following sections present specific issues when using the INSERTBUFF option with some of the listed SAS/ACCESS interfaces.

#### ***Using the INSERTBUFF option with DB2 UDB***

If one row in the insert buffer fails, all rows in the insert buffer fail.

#### ***Using the INSERTBUFF option with MySQL***

MySQL supports the multi-value INSERT statement through the INSERTBUFF option. When the INSERTBUFF option is set to any value, a multi-value INSERT statement is generated by the SAS/ACCESS engine. For example, instead of generating single INSERT statements as follows:

```
Insert into x values ('a', 1, 1000)
Insert into x values ('b', 2, 1001)
...
Insert into x values ('zzz', 10001, 10000000)
```

one SQL statement is generated by the SAS/ACCESS engine that hold as many rows as possible without overflowing the maximum length specified in the SQL statements. For example:

```
Insert into x values ('a', 1, 1000), ('b', 2, 1001), ... ('zzz', 10001, 10000000)
```

#### ***Using the MULTISTMT Option in the SAS/ACCESS Interface to Teradata***

The SAS/ACCESS engine to Teradata supports a MULTISTMT option that causes the engine to generate a multi-row insert, using the same mechanism as the Teradata TPump utility. Significant performance gains can be obtained when compared to single-row inserts when large volumes of data are inserted.

The SAS/ACCESS engine to Teradata dynamically calculates the actual number of INSERT statements to send to the Teradata relational database based on several factors such as how many SQL INSERT statements can fit in a 64K buffer, how many data rows can fit in the 64K data buffer, and how many inserts the Teradata server chooses to accept.



**Notes:** Be aware of the following caveats when using the MULTISTMT option:

If you use the MULTISTMT option with the DBCOMMIT option, the values should be the same for both options. If the values are not the same, the smaller of the value of the DBCOMMIT option and the number of INSERT statements that fit in a buffer are used as the number of INSERT statements to send in a batch.

The ERRLIMIT option is not supported with the MULTISTMT option at this time.

### Using the UPDATEBUFF Option

This UPDATEBUFF option is supported by the SAS/ACCESS to Oracle engine only. When the UPDATEBUFF option is set to a value greater than 1, the option specifies the number of rows that can be held in memory for an update from SAS to the database. The default value is 1. The value of 1 is useful for PROC FSVIEW and the VIEWTABLE because it prevents the engine from updating multiple rows and holding a lock on the table. When the UPDATEBUFF option is set to 1, a record-level lock is obtained only for the row that is being edited.

## Restricting SAS Functions

---

A great strength of SAS is that it supports hundreds of SAS language functions that can be specified in SAS SQL queries or SAS WHERE statements. Databases also provide similar functions that can be used in SQL queries.

When there are even slight differences in implementation of the function, mapping (translating) the SAS function to the database function can introduce differences in the query results. For this reason, a SAS/ACCESS product automatically maps a function only when implementation of the SAS function matches the database function. SAS functions that cannot be mapped to a database function cannot be passed to the database, so all the rows that would have been filtered must be pulled into SAS.

Functions that don't pass to a database prevent implicit pass-through. Because SAS supports a large number of functions—and relatively few SAS functions have a true database counterpart—we offer the following recommendations for using SAS functions with database data:

Choose the SAS function that SAS/ACCESS can pass to the database.

When possible, use expressions instead of SAS functions.

If the function value is constant across the query, consider using a SAS macro assignment of a SAS function value.



**Note:** SAS functions that can be passed to one database might not be passed to another database. Check the SAS/ACCESS documentation for your database to see which SAS functions the SAS/ACCESS map. (See link to the documentation at end of subsection.)

### Selecting the SAS Function That SAS/ACCESS Can Pass to the Database

Whenever possible, select a SAS function that SAS/ACCESS can pass to the database. For example, the online documentation for the SAS/ACCESS Interface to Oracle indicates that the product can translate the SAS TRIMN function to Oracle RTRIM function. In contrast, the SAS/ACCESS cannot translate the SAS TRIM function because that function behaves differently with database data that contains either

values of all blanks or NULLs.

In the following example, the Oracle table column COMPLETE\_CODE does not contain blank data. In this case, you could use either the SAS TRIMN or TRIM function to obtain the expected results. However, as shown in the following example, you can enhance performance by using the TRIMN function because this SAS function has a database equivalent. (The elapsed time for both query runs, first using the TRIM function then the TRIMN function, is shown in bold in the SASTRACE output.)

The following code example uses the SAS function TRIM that cannot be passed to Oracle, therefore, all the rows of the table are retrieved into SAS. The portion of the query that is passed to Oracle and the process times are bolded.

```
options sastrace=",,,d" sastraceloc=saslog nostsuffix;

/* A query using TRIM, an unmapped SAS function*/
proc sql;
select component_id, lead from dbms.project
where trim(complete_code) = 'True';
quit;
SASTRACE Output

ORACLE_1: Prepared: on connection 0
SELECT * FROM PROJECT

ORACLE_2: Prepared: on connection 0
SELECT "COMPONENT_ID", "LEAD", "COMPLETE_CODE" FROM PROJECT

ORACLE_3: Executed: on connection 0
SELECT statement ORACLE_2

NOTE: The PROCEDURE SQL printed page 1.
NOTE: PROCEDURE SQL used (Total process time):
      real time          4.14 seconds
      cpu time           2.03 seconds
```

The following code example uses the WHERE clause with the SAS function TRIMN that can be passed to Oracle, therefore, only the rows matching the WHERE clause are transferred to SAS. The portion of the query that is passed to Oracle and the process times are bolded.

```
/* A query using TRIMN, a mapped SAS function */
proc sql;
select component_id, lead from dbms.project
where trimn(complete_code) = 'True';
quit;

SASTRACE Output

ORACLE_4: Prepared: on connection 0
SELECT * FROM PROJECT

ORACLE_5: Prepared: on connection 0
SELECT "COMPONENT_ID", "LEAD", "COMPLETE_CODE" FROM PROJECT WHERE
( RTRIM("COMPLETE_CODE") = 'True' )

ORACLE_6: Executed: on connection 0
SELECT statement ORACLE_5
```

NOTE: The PROCEDURE SQL printed page 2.  
 NOTE: PROCEDURE SQL used (Total process time):  
       real time              **0.32 seconds**  
       cpu time               **0.01 seconds**



**Note:** SAS functions that can be passed to one database might not be passed to another database. Check the SAS/ACCESS documentation for your database to see which SAS functions the SAS/ACCESS product can map.

### Using Expressions Instead of Functions

SAS functions provide analytic processing with an economy of program code. This convenience presents a problem because the use of SAS functions in some situations can diminish processing performance. Some situations where SAS functions can reduce performance include:

A repeated call of a SAS function in a DATA step or in a WHERE clause.

Applying a function to a column can prohibit index use.



**Note:** There is significant overhead when you call a function repeatedly in program code. The repetition is easy to recognize in a SAS DATA step, but it can also exist in SAS WHERE clause processing. The function that is specified in your WHERE clause is called to evaluate each row as it is fetched from the database. Because the additional overhead can be whether the function is passed through to the database, the above cautions apply not only to database data access but also to native SAS data access as well.

For optimal performance, carefully consider the use of a SAS function in these situations. Sometimes you can avoid using a function in an SQL query by making slight modifications to the code as shown in the following example.

In both of the following examples, a query is run to obtain a list of projects that were completed in the calendar year 2005. (The elapsed time for both runs, first using a SAS function and then an efficient comparison, is shown in bold in the SASTRACE output.)

The following example uses the SAS YEAR function with a database DATE column in the query. Because the SAS/ACCESS Interface for DB2 cannot map the YEAR function to an equivalent database function, this query cannot be passed to the database. The portion of the query that is passed to DB2 and the process times are bolded.

```
options sastrace=",,,d" ;

proc sql;
select component_id, lead from dbms.project
where year(complete_date) = 2005;
quit;
```

SASTRACE Output

```
DB2_1: Prepared: on connection 0
SELECT * FROM PROJECT FOR READ ONLY
```

```
DB2_2: Prepared: on connection 0
```

```
SELECT "COMPONENT_ID", "LEAD", "COMPLETE_DATE" FROM PROJECT FOR READ ONLY
```

```
DB2_3: Executed: on connection 0
Prepared statement DB2_2
```

```
10                                     ;
NOTE: The PROCEDURE SQL printed page 1.
NOTE: PROCEDURE SQL used (Total process time):
      real time          4.29 seconds
      cpu time           1.96 seconds
```

In the above example, while only one row of the PROJECT table meets the criteria of the WHERE clause, all 500,000 rows of the PROJECT table had to be brought into SAS because the WHERE clause could not be passed to the database. In the following example, the WHERE clause is re-coded with an expression that avoids the function reference. The WHERE clause is passed to the database and only the identified rows that match the WHERE clause are returned to SAS. The re-coded query can also be passed to the database.

```
proc sql;
select component_id, lead from dbms.project
where complete_date between '01jan2005'd and '31dec2005'd;
quit;
```

Here is the SASTRACE output from this example:

```
DB2_4: Prepared: on connection 0
SELECT * FROM PROJECT FOR READ ONLY
```

```
DB2_5: Prepared: on connection 0
SELECT "COMPONENT_ID", "LEAD", "COMPLETE_DATE" FROM PROJECT WHERE ( (
"COMPLETE_DATE" BETWEEN DATE({d '2005-01-01' }) AND DATE({d '2005-12-31' }) ) ) FOR
READ ONLY
```

```
DB2_6: Executed: on connection 0
Prepared statement DB2_5
```

```
16         quit;
NOTE: The PROCEDURE SQL printed page 2.
NOTE: PROCEDURE SQL used (Total process time):
      real time          0.56 seconds
      cpu time           0.03 seconds
```

### Assigning a SAS Function Value to a SAS Macro When Possible

To avoid using a SAS function, you can pre-evaluate an expression in a SAS macro variable definition, and then use the SAS macro variable instead of a SAS function call. This technique can be useful in repetitive call situations or when the SAS function cannot be translated to an equivalent database function. The following example illustrates use of a sample macro assignment replacement.

In both of the examples in this section, the query contains a filter expression that returns account numbers when the Date column value is less than one week from the current date. The elapsed time for both runs, first using an arithmetic expression and then a SAS macro variable, is shown in bold in the SASTRACE output.

The first example uses a SAS function as an arithmetic expression in the WHERE clause. As the SASTRACE output shows, the arithmetic expression or the function cannot be passed to the database.



Therefore, the entire table is returned to SAS, not just the result set specified. In this case, the function TODAY is evaluated for every row in the database table. This unnecessary evaluation consumes processing resources and is negatively impact performance.

```
/* Use of an arithmetic expression in the query filter */
option sastrace=",,,d" sastraceloc=saslog no$stsuffix;

proc sql;
select account, ship_date from dbms.neworders where ship_date < today() + 7 ;
quit;
```

Here is the SASTRACE output from this example:

SASTRACE Output

```
DB2_2: Prepared: on connection 0
SELECT "ACCOUNT", "SHIP_DATE" FROM NEWORDERS FOR READ ONLY

NOTE: The PROCEDURE SQL printed page 1.
NOTE: PROCEDURE SQL used (Total process time):
      real time          0.89 seconds
      cpu time           0.73 seconds
```

The second example uses the macro variable &TARGET\_DATE to make the date comparison. Using the SAS macro variable enables you to directly compare the column value with a date literal value. This re-coded query can be passed to the database, and as the elapsed time shows, is a more efficient comparison.

```
/* Use of a SAS macro variable in the query filter */
%let target_date=%eval(%sysfunc(today()) + 7 );
proc sql;
select account, ship_date from dbms.neworders where ship_date < &target_date ;
quit;
```

Here is the SASTRACE output from this example:

SASTRACE Output

```
DB2_5: Prepared: on connection 0
SELECT "ACCOUNT", "SHIP_DATE" FROM NEWORDERS WHERE ( "SHIP_DATE" < DATE({d '2006-03-16' }) ) FOR READ ONLY

NOTE: The PROCEDURE SQL printed page 2.
NOTE: PROCEDURE SQL used (Total process time):
      real time          0.09 seconds
      cpu time           0.03 seconds
```



**Note:** If this query is a long-running job that begins before midnight but ends after midnight, the macro value will not change but the value of a function changes after midnight.

In some situations, you might need to use a specific SAS function, and you might not have the option of using alternative SAS program code. In these situations, using a SAS function that does not have a database equivalent means that more processing occurs in SAS. Consequently, the performance of your SAS program step might not be optimal.

For more information: , SAS Institute Inc. 2007.SAS online documentation “Optimizing Your SQL Usage Passing Functions to the DBMS Using PROC SQL” Available (<http://support.sas.com/onlinedoc/913/getDoc/en/acreldb.hlp/a001630468.htm>).

## Constructing a WHERE Clause Correctly

Whenever possible, SAS/ACCESS translates a WHERE clause into generated SQL code and passes them to the relational database for more efficient processing. The performance impact can be particularly significant when accessing large database tables.



**Note:** If you need to prevent WHERE clauses from being passed to the relational database, use the LIBNAME option **DIRECT SQL=** NOWHERE.

The following list gives general guidelines for writing an efficient WHERE clause:

Avoid the NOT operator if you can use an equivalent form.

Efficient	Inefficient
where zipcode<=8000	where zipcode not>8000

Avoid the >= and <= operators if you can use the BETWEEN predicate.

Efficient	Inefficient
where ZIPCODE between 70000 and 80000	where ZIPCODE>=70000 and ZIPCODE<=80000

Avoid LIKE predicates that begin with % or \_ .

Efficient	Inefficient
where COUNTRY like 'A%INA'	where COUNTRY like '%INA'

Avoid arithmetic expressions in a predicate.

Efficient	Inefficient
where SALARY>48000.00	where SALARY>12*4000.00

Avoid using SAS functions in a WHERE clause. If the SAS function does not pass down to the database, the entire WHERE clause is handled by SAS and not by the database.

Use the DBKEY, DBINDEX, and MULTI\_DATASRC\_OPT options when appropriate. For details about these options, see the SAS online documentation: “ SAS Institute Inc. 2007.SAS online documentation “SAS/ACCESS Relational Databases: Concepts: Optimizing Your SQL Usage: Using the DBINDEX=, DBKEY=, and MULTI DATASRC OPT= Options”Available (<http://support.sas.com/onlinedoc/913/getDoc/en/acreldb.hlp/a002253008.htm> ).



**Note:** If you have NULL values in a database column that is used in a WHERE clause, be aware that your results might differ depending on whether the WHERE clause is processed in SAS or is passed to the database for processing. This is because databases tend to remove NULL values from consideration in a WHERE clause, while SAS does not. For more information, see the "Difference in Behavior" section later in this document.

## Sorting in the Relational Database

---

Sorting data is a resource-intensive operation and should only be done when you need the data in a specific order. Efficient sorts can maximize the performance of your jobs.

It is usually more efficient to pass the sort to the relational database. If the BY statement in the DATA step or the ORDERBY clause in PROC SQL is specified, SAS/ACCESS automatically generates an ORDERBY clause for that variable. PROC SORT passes the sort to the database using the ORDERBY clause unless the SORTPGM option is set to something other than BEST. The ORDERBY clause causes a sort of the data to occur on the relational database before the data set or PROC step uses the data in SAS.

Sort stability, meaning that the ordering of the observations in the BY statement is exactly the same every time the sort is run, is not guaranteed when you query data stored in a relational database. Because the data in the relational database might not be static data, the same query issued at different times might return the data in different order. If you require sort stability of the data, sort on a unique key, or place your database data into a SAS data set and then sort it.



**Note:** Do not use PROC SORT to sort data from SAS back to the relational database. Doing so has no effect on the order of the data in the database and only impedes performance.

Also, sorting by columns that are included in database indexes can be much faster than sorting by columns that are not sorted. Therefore, if some of the columns to be sorted by are indexed and others are not, sort first by the indexed columns.

## Naming Issues

---

When reading in database data into SAS or outputting data on the database, you must be aware of certain naming issues that involve:

- SAS names and support for database names
- PROC SQL reserved words
- Database reserved words

SAS/ACCESS online documentation gives extensive information about variable naming conventions. To avoid redundancy, this section only identifies naming issues, and then refers you to the online documentation for details and examples. For full documentation on SAS naming conventions, see the SAS/ACCESS online documentation topic:

SAS Institute Inc. 2007. SAS online documentation "SAS Names and Support for DBMS Names SAS Naming Conventions" Available (<http://support.sas.com/onlinedoc/913/getDoc/en/acreldb.hlp/a001392879.htm>)

### SAS Names and Support for Database Names

In SAS, table names, column names, and aliases are limited to 32 characters and can contain mixed case. A SAS name can contain mixed case, but the capitalization of a name is not used as a distinguishing characteristic when compared to other SAS names. Most databases allow case-sensitive names, names with special characters, and names that exceed 32 characters. To work with database data, you must know how SAS/ACCESS handles table and column names that are not allowed in SAS, as well as how to modify the default behavior if desired or required.

#### ***SAS/ACCESS Default Naming Behavior for Input Operations***

By default, as SAS/ACCESS reads data in from the database it will:

- Replace blank spaces or special characters that are not allowed in SAS names (such as @, #, %) with an underscore (\_).
- Truncate any database column names that exceed 32 characters.



**Note:** Some databases support maximum name length of less than 32 characters. See the appropriate database documentation when creating tables in the database.

See: SAS Institute Inc. 2007 "SAS Names and Support for DBMS Names SAS/ACCESS Default Naming Behaviors" Available <http://support.sas.com/onlinedoc/913/getDoc/en/acreldb.hlp/a001716973.htm>

**Options That Modify SAS/ACCESS Default Naming Behavior**

To change the default behavior for case-sensitive or nonstandard database table and column names, you can use one or more of the following options.

SAS Option	Option Type	Setting	Effect
PRESERVE_COL_NAMES=	LIBNAME and dataset option	YES	Preserves spaces, special characters, and mixed case in database column names. This option affects column names only when they are created in the database.
PRESERVE_TAB_NAMES=	LIBNAME	YES	Preserves blank spaces, special characters, and mixed case in database table names.
DQUOTE=	PROC SQL	ANSI	Enables SAS code to refer to database names that contain characters and spaces that are not allowed by SAS naming conventions. When you specify DQUOTE=ANSI, you can preserve special characters in table and column names in your SAS SQL statements by enclosing the names in double quotation marks.



**Note:** The availability of these options and their default settings are database-specific, so consult the SAS/ACCESS documentation for your database to learn how your SAS/ACCESS product processes database names.

For more information about these options:

SAS Institute Inc. 2007. SAS online documentation “SAS/ACCESS for Relational Databases/SAS Names and Support for DBMS Names: Options that Affect naming behavior” Available <http://support.sas.com/onlinedoc/913/getDoc/en/acreldb.hp/a001393325.htm>

**Use of the Options and Examples**

Below are links to tables (from SAS/ACCESS online documentation) that illustrate how SAS/ACCESS processes database names when retrieving data and creating data from a database. This information applies generally to all the interfaces. See the documentation for your SAS/ACCESS interface for details.

For **retrieving** database data:

SAS Institute Inc. 2007 “SAS/ACCESS for Relational Databases/SAS Names and Support for DBMS Names: Naming Behavior When Retrieving DBMS Data” <http://support.sas.com/onlinedoc/913/getDoc/en/acreldb.hp/a001393327.htm>

For **creating** database data:

SAS Institute Inc. 2007 “SAS/ACCESS for Relational Databases/SAS Names and Support for DBMS Names: Naming Behavior When Creating DBMS Data” <http://support.sas.com/onlinedoc/913/getDoc/en/acreldb.hp/a001393329.htm>

For naming examples:

SAS Institute Inc. 2007 "SAS/ACCESS for Relational Databases/SAS Names and Support for DBMS Names: SAS/ACCESS Naming Examples"

<http://support.sas.com/onlinedoc/913/getDoc/en/acreldb.hlp/a001393333.htm>

### **PROC SQL Reserved Words**

PROC SQL's SQL syntax contains a few reserved words: an example is the keyword CASE. If you have a table column that is named CASE, and you want to specify it in a PROC SQL step, you can surround CASE in double quotation marks ("CASE") and set the PROC SQL option DQUOTE=ANSI.

For more information, see the Reserved Words topic in the section:

SAS Institute Inc. 2006 SAS online documentation "SAS/ACCESS for Relational Databases/The SQL Procedure: PROC SQL and the ANSI Standard" Available:

<http://support.sas.com/onlinedoc/913/getDoc/en/proc.hlp/a002473705.htm>

### **Database Reserved Words**

Every database also has reserved words. To avoid inadvertent use of a database reserved word when you create or name database data that results in SQL errors or code failures, use the PRESERVE\_COL\_NAME option presented in this section, or check the documentation for your database.

---

## **Differences in Behavior**

---

It is imperative to understand the differences in how SAS and relational databases store and process data. Behavioral differences between how SAS and databases store and process data are found in these three areas:

- NULL values and MISSING values
- Physical ordering of data, or lack thereof
- Native data types support

### **NULL Values and MISSING Values**

---

Relational databases and SAS provide different mechanisms for indicating the absence of a data value. In the SAS language, data absence is represented using the MISSING value, which is conceptually but not exactly analogous to a relational database NULL value. SAS/ACCESS APIs to relational databases translate a SAS MISSING value into a relational database NULL when inserting or updating a relational database table. A relational database NULL will conversely be translated into a SAS MISSING value when a relational database table is queried from within SAS. The following sections discuss the notable differences in how these two similar elements are handled by these systems.

## Matching Differences with NULL or MISSING Values

In SAS, MISSING values represent an absence of data and are considered as special data values to SAS. Standard SAS MISSING values have the following attributes:

- For numeric variables, the standard SAS MISSING value is the lowest (most negative) possible value and is represented by special floating point numbers.
- For character variables, a MISSING value is represented by all blanks. This permits MISSING values in SAS to be successfully used with standard comparison operators in WHERE or HAVING clauses, or with join conditional expressions.

However, for most relational database systems, a blank can be a valid data value. Most relational databases use a NULL value to represent a true *absence* of data. A column's status as NULL is reflected by a 'Null indicator'. There is no data value associated with the column for standard comparison operations to operate on. Relational database NULLs successfully compare only with special SQL NULL comparison operators, such as '*column IS NULL*' or '*column IS NOT NULL*'. For more information about blanks and relational databases, see "Character MISSING Represented as SAS Blank String and Its Ramifications".

It is important to be aware of these differences when working with relational data stores. NULL or MISSING data in the columns used in such expressions can return unexpected results, as shown in the following example.

Identical sample data is loaded into SAS and DB2 target tables. Suppose you also want to create a separate SAS data set that contains only the observation that contained MISSING values from the tables. Coding a standard comparison filter (Where  $x < 0$ ) works successfully when used to filter the SAS data set, but no rows are found using this filter with the DB2 table. This is due to the failure of the standard comparison filter.

```

6          LIBNAME RDBMS Db2 Db=mysample;
NOTE: Libref RDBMS was successfully assigned as follows:
      Engine:          DB2
      Physical Name:  mysample
7
8
9          /* create sample data tables in SAS and DB2 */
10         data work.source rdbms.source;
11         x=1; output;
12         x=.; output;
13         run;
NOTE: The data set WORK.SOURCE has 2 observations and 1 variables.
NOTE: The data set RDBMS.SOURCE has 2 observations and 1 variables.
NOTE: DATA statement used (Total process time):
      real time          0.31 seconds
      cpu time           0.03 seconds
14
15         /* find SAS observations that contain missing values */
16         data work.target;
17         set work.source; where (x < 0 );
18         run;
NOTE: There were 1 observations read from the data set WORK.SOURCE.
      WHERE x<0;
NOTE: The data set WORK.TARGET has 1 observations and 1 variables.
NOTE: DATA statement used (Total process time):
      real time          0.03 seconds

```

```
cpu time          0.03 seconds
```

```
19
20      /* find DB2 observations that contain missing values */
21      data work.target;
22      set rdbms.source; where (x < 0 );
23      run;
NOTE: There were 0 observations read from the data set RDBMS.SOURCE.
      WHERE x<0;
NOTE: The data set WORK.TARGET has 0 observations and 1 variables.
NOTE: DATA statement used (Total process time):
      real time          0.01 seconds
      cpu time           0.01 seconds
```

Using standard comparators or SAS functions to test for MISSING with relational data stores is not recommended. To avoid filter processing differences when there are NULL and MISSING data variables, it is recommended that you use the 'IS NULL' or 'IS NOT NULL' operators.

In the following example, notice that the target tables are successfully created for both data stores because the filter expression was coded using the 'IS NULL' conditional.

```
24
25      /* find SAS observations that contain missing values */
26      data work.target;
27      set work.source; where (x is null );
28      run;
NOTE: There were 1 observations read from the data set WORK.SOURCE.
      WHERE x is null;
NOTE: The data set WORK.TARGET has 1 observations and 1 variables.
NOTE: DATA statement used (Total process time):
      real time          0.03 seconds
      cpu time           0.03 seconds
```

```
29
30      /* find DB2 observations that contain missing values */
31      data work.target;
32      set rdbms.source; where (x is null );
33      run;
NOTE: There were 1 observations read from the data set RDBMS.SOURCE.
      WHERE x is null;
NOTE: The data set WORK.TARGET has 1 observations and 1 variables.
NOTE: DATA statement used (Total process time):
      real time          0.03 seconds
      cpu time           0.03 seconds
```

It is also possible to notice result differences between SAS and a relational database when a multiple table outer join is executed by using PROC SQL. This can occur even when the data in the tables does not contain NULLS. Because PROC SQL logically processes joins as two table operations, when more than two tables are involved in a join, intermediate result sets are constructed. These intermediate result sets might contain NULLS generated through the execution of the join. For more information, see the following sources:

SAS Institute Inc. 2007 SAS online documentation: "Potential Result Set Differences When Processing Null Data" <http://support.sas.com/onlinedoc/913/getDoc/en/acreldb.hlp/a001924296.htm>



Levine, Fred. 2008. "Potential Result Set Differences between Relational DBMS and the SAS System" Proceedings of the SAS Global Forum 2008 Conference. Cary, NC: SAS Institute Inc. Available at (<http://support.sas.com/resources/papers/resultsets.pdf>)

### **NULL and MISSING Sort Order**

Sorts occur in SAS when an ORDER BY clause is used in a PROC SQL query, or a BY statement is used in a SAS DATA step or in a SAS procedure.

The SAS/ACCESS engines incorporate ORDER BY clauses in the queries they build if an ORDER BY clause is present in the SAS query. For reasons of improved performance, it is generally recommended that sorting be done by the relational database. However, if a sort is done by the relational database, then it is possible that the data that is returned is in a different order than the SAS process expects.

NULLs, whether character or numeric, can be handled differently when sorted by the relational database than when MISSING values are sorted in SAS. When a sort is performed by a relational database, NULL data values are returned last or low in the sort order. When a sort is performed in SAS, MISSING value data items are returned first or high in the sort order.

If the sorted data is used with the BY statement in a data set that requires the data to be in the sorted order supported by SAS, the execution might result in an error message indicating that the data isn't in sorted order.



**Note:** If such an error message appears, consider using the SAS option SORTEDBY to avoid the check of the sort order and the resulting error.

### **Character MISSING Represented as SAS Blank String and Its Ramifications**

The SAS MISSING value for a SAS character variable is represented as a blank. Because relational database character columns can contain blanks, a blank value in a relational database column would be considered a data value and not a NULL or MISSING value. When extracting character column data from a relational database table, SAS/ACCESS engines translate relational database NULLs to the blank SAS MISSING value format. When writing character column data to a relational database table, by default, SAS/ACCESS engines, write the SAS MISSING value as a relational database NULL.

However, because a blank can be a valid data value in a relational database column, then the following sequence of events:

- Reading in a valid data value of blanks from the relational database to SAS.
- SAS converting that data value of blanks to a SAS MISSING value.
- Updating the relational database by writing that MISSING value to the relational database as a NULL, can contribute to dilution of these real data values with extraneous NULL/MISSING values. Just as with NULL numeric relational database data, it is important to maintain an awareness of the differences in behavior when working with relational data stores.

To overcome and influence the default Insert and Update behavior of SAS/ACCESS engines, use the NULLCHAR= or the NULLCHARVAL= option. For more information:

SAS Institute Inc. 2007. SAS Institute online documentation “Data Set Options for Relational Databases NULLCHAR Option” Available  
<http://support.sas.com/onlinedoc/913/getDoc/en/acreldb.hlp/a001371592.htm>)

SAS Institute Inc. 2007. SAS Institute online documentation Data Set Options for Relational Databases NULLCHARVAL:Option” Available  
<http://support.sas.com/onlinedoc/913/getDoc/en/acreldb.hlp/a001371594.htm>)

### **SAS Special MISSING Values**

Another difference between relational database NULL values and SAS MISSING values is the variety of special SAS MISSING values that are supported. Currently, there are 28 possible values for a SAS MISSING value. They are represented by the characters: . (dot), \_ . (underscore-dot), and .A through .Z (dot-a through dot-z). These various MISSING values must still be distinguishable from each other. Because the variety of SAS MISSING values supported are all considered special data values that represent the absence of data, they can be evaluated with standard comparison operators. This is generally not possible with relational database NULLs. The distinctions among SAS special MISSING values cannot be preserved in a relational database table as a relational database NULL. For SAS applications that make a distinction among the multiple possible values of MISSING in SAS, these values would need to be maintained as real data in a relational database table.

### **Repeatable Observation or Row Order**

---

Some SAS analytics require that data set observations be retrieved in the same order when a data set is read multiple times. An analysis delivers incorrect results if observation order varies. An example is the SAS ETS time series analysis. A typical first step in time series is sorting source data on a date column. This creates a new SAS data set with old dates first and new dates last. Ensuing analyses can read and re-read the sorted data set with no BY clause because SAS retrieves data set observations in the order that they were originally written.

Database tables do not retrieve rows in the order in which they were originally written. Without ORDER BY asserted in the database SQL, rows are returned in random order each time a table is read. Therefore, time series analysis of a database table requires a BY clause on a unique key each time a table is read. The BY clause generates ORDER BY as part of the SQL, and the database sorts the rows before transferring them to SAS. Because the key is unique, there are no rows with identical keys whose retrieval order could vary for each read.

Thorough understanding of database ordering avoids error-prone SAS programming practices against databases. Another example is the data set ‘MODIFY’ statement. For a non-unique BY key, the MODIFY statement updates the first row retrieved with a matching key. However, you cannot predict the first row when a database retrieves multiple rows with the same key. Therefore, you probably generate erroneous results by applying the MODIFY to a database table with a non-unique BY key.

For a simple example of unpredictable ordering with a non-unique BY key and adding another BY variable to ensure predictability:

SAS Institute Inc. 2003 SAS online documentation "Using a BY Clause to Order Query Results in the SAS/ACCESS for Teradata" Available:

(<http://support.sas.com/91doc/getDoc/acreldb.hlp/a001399962.htm#a001399973>)

### Repeatable Observation Order and SAS/ACCESS Spooling

A single PROC step in SAS with one input data set reference might cause two or more reads of the data set, and the procedure might require the observation order to be consistent from one read to the next. To mimic SAS data set behavior and return data in identical order on multiple passes when accessing a relational database table, SAS/ACCESS engines include a SPOOL option that is ON by default. The SAS/ACCESS engine executes the SQL and retrieves the result set only once (on the first pass) then spools the output into a SAS work area. On a second or later pass, the data is read in from the spool and not from the relational database. With spooling, SAS/ACCESS engines mimic SAS data set behavior, returning data in identical order each pass.

Because the data is copied from the relational database into a temporary SAS data set, a large result set means a large spool. If you apply the best practices that are recommended in this paper to minimize the amount of data that is requested from the database, you can minimize the spool size and SAS memory usage.



**Note:** If spooling is not needed (for example, if the data is not changed between reads), setting SPOOL to OFF can improve SAS performance by eliminating the WRITE of the temporary table.

### Repeatable Observation Order and Copying Data to a SAS Data Set

Instead of repeatedly accessing a database table with ORDER BY, you might copy the required database table rows and columns to a SAS data set. This simplifies the following aspects of your design:

- Activity to the database is limited to one SQL request.
- Data transfer cost from the database to SAS is incurred only once.
- Other users potentially can modify the database table between your accesses to it; whereas the SAS data set is a private and unchanging snapshot of the data.
- Improved performance after the data is stored in a native SAS data set due to not involving the SAS/ACCESS engine.
- However, there are also the following potential negatives to copying to a SAS data set:
- You risk running out of disk space if the database data is extremely large.
- The static SAS copy might become out-of-date versus the database master version.

Any join of the SAS copy to another database table causes a heterogeneous join, when instead the join could be made to occur entirely on the database using the database version of the table. For more information, see the [Heterogeneous Joins](#)

\_chapter.

There might be other pros and cons to creating a SAS copy of database data, including previously mentioned repeatable observation order issues. Careful consideration of pros and cons is crucial in designing a program or application.

## Understanding the Differences between SAS and Database Native Data Types

---

SAS and relational databases support different native data types. When executing a query to a database, a SAS/ACCESS engine is responsible for mapping and converting between the data types.

The following sections provide details about a variety of issues that relate to the mapping and data conversion of variables that you need to understand in order to write efficient SQL that behaves as expected.

### Differences between Database and SAS Data Types

SAS data stores support two native data types:

- Character: fixed-length character data, 1 to 32767 characters long
- Numeric: double-precision floating-point number

Other data types are represented using SAS formats to impart the type. For example, SAS date values are stored internally as a floating-point number. The underlying data is numeric and can be manipulated as a number but is displayed in DATE format.

Relational database numeric and temporal values are stored in SAS as a double-precision floating-point number. While float numbers support large magnitude (numbers such as 2 to the 30th power) and precision (many digits to the right of the decimal place), they also have an inherent precision constraint. A limit of 15 digits is accurately represented by SAS floating-point numbers. Database columns that support more than 15 digits of precision might require special handling to preserve precision and operate correctly in WHERE clauses and joins.

In contrast to SAS, databases systems support more native data types:

- Numeric types: Decimal, Integer, Floating point, and so on.
- Character types: Fixed and varying length character, national character.
- Temporal types: Date Time, Timestamp, and Interval.
- Binary.
- Large object: Binary and Character large objects.

Databases can also support native Boolean or currency types. Some databases support user-defined or abstract types. These unusual database data types might not effectively map to a native SAS data type. Database temporal and numeric types are converted to SAS floating-point values when the data is brought into SAS. When possible, the SAS/ACCESS engine directs the database to perform the conversion. If the database cannot provide the conversion, the SAS/ACCESS engine performs it.



**Note:** Temporal types are always converted by the SAS/ACCESS engine. This conversion might add some processing overhead.

### Appropriate Choices for Database Columns that Contain Large Numeric Values

The differences in the data types between databases and SAS when converting large numeric columns need to be taken into consideration. Extracting integral values of 15 digits or less from relational databases into SAS are completely reliable. Whether converted by SAS or the database, each integer value translates to a unique and discrete SAS float representation.

For more information about SAS floating-point precision:

SAS Institute Inc. 2003 SAS online documentation "SAS Variables Numeric Precision in SAS Software"  
<http://support.sas.com/onlinedoc/913/getDoc/en/lrcon.hlp/a000695157.htm>

SAS Institute Inc., 2007, SAS Note 654, "Numeric Precision 101,"  
<http://support.sas.com/techsup/technote/ts654.pdf>

### Handling Large Numeric Key Columns with SAS

This conversion of large numeric values can be especially important when working with database keys. Because the maximum reliable precision of a SAS float is 15 digits, this paper defines large database numeric columns as columns with 16 or more digits of precision. These large database numeric columns are capable of precision greater than the maximum reliable precision of a SAS float. Whenever possible, constrain the definition of database key columns to the following database types to ensure that the translation into a SAS float matches the key on the database.

- Character
- Small integer (16-bit)
- Integer (32-bit)
- DECIMAL(1,0), DECIMAL(2,0), DECIMAL(14,0), DECIMAL(15,0)
- FLOAT (constrained to integral values)



**Note:** While this section uses key columns as an example that might be encountered in a real-world situation, this discussion also applies to non-key large numeric columns and non-integer large numeric columns.

While key columns are typically integer values, database key columns can also, but rarely, be large numeric columns or non-integer large numeric columns. Database types for large numeric key columns include:

- Big Integer (64-bit integer)
- DECIMAL(16,0), DECIMAL(17,0), DECIMAL(18,0), and so on

Special considerations apply for the following tasks:

- Reading database large numeric key columns into SAS.
- Writing large numeric key columns back to the database.
- Asserting WHERE clauses against database large numeric key columns.
- Performing joins involving database large numeric key columns.

#### **Reading Database Large Numeric Key Columns into SAS**

One way to preserve precision when reading the large numeric values into SAS is to convert the database numeric column to a SAS character column. With many SAS/ACCESS interfaces, this is accomplished with the DBSASTYPE option.



**Note:** The DBSASTYPE option is not supported with the SAS/ACCESS Interface to Teradata. Explicit SQL pass-through is required to preserve precision when reading a large numeric. Check the SAS/ACCESS documentation to confirm support of the DBSASTYPE option with a specific SAS/ACCESS product.

The following example shows how precision is lost with a standard SAS read, and how to retain precision with the DBSASTYPE option. Given the following DB2 table named TEST:

ACCOUNT_NO	EXPENDITURES
925512340233844230	127.20
925512340233844231	3449.99

This first example query:

```
PROC SQL;
select account_no from db2.test;
quit;
```

returns these results showing the loss of precision:

```
ACCOUNT_NO
-----
925512340233844224
925512340233844224
```

This second example query, however, preserves the precision of the values by defining the values as character variables:

```
PROC SQL;
select account_no from db2.test(dbsastype=(account_no="char(18)"));
quit;
```

by returning the following results:

```
ACCOUNT_NO
-----
925512340233844230
925512340233844231
```

### ***Writing Large Numeric Key Columns Back to the Database***

SAS character columns that contain large numeric data also require special handling to be written back to a database large numeric column. The DBSASTYPE option allows SAS steps to output a SAS character column to a database numeric column. The following example shows how to use the DBSASTYPE option to convert a SAS character column to an Oracle large numeric data column appends the data into the Oracle table by using PROC APPEND.

```
data local;
account_no = '12345678901234567890';
run;

PROC SQL;
connect to oracle(user=oracle_user pass=oracle_passwd path=oracle_path );
execute (create table test(account_no decimal(20))) by oracle;
execute (commit) by oracle;
quit;

proc append base=x.test(dbsastype=(account_no="char(20)")) data=local; run;

proc print data=x.test(dbsastype=(account_no="char(20)")); run;
```

```
Obs    ACCOUNT_NO
  1    12345678901234567890
```

### ***Asserting a WHERE Clause against Database Large Numeric Key Columns***

The imprecision of SAS floating-point numbers can affect WHERE clause processing. This is especially true if you use numeric literals in the WHERE clause expression. The following example attempts to query an Oracle DECIMAL(20,0) column that contains the 20-digit value 12345678901234567890. Due to SAS floating-point imprecision beyond 15 digits, this 20-digit value coded in the WHERE clause is not replicated in the generated SQL.

```
data _null_;
set x.test2;
where (i=12345678901234567890);
run;
ORACLE_23: Prepared:
SELECT  "I" FROM TEST2  WHERE  ("I" = 12345678901234567168)
```

```
ORACLE_24: Executed:
SELECT statement  ORACLE_23
```

NOTE: There were 0 observations read from the data set X.TEST2.

When you use a numeric literal in a standard SAS WHERE clause, SAS internally converts the number to floating point. A number bigger than 15 digits loses precision and the WHERE clause that is generated is not identical to the WHERE clause that was entered. SAS/ACCESS engines provide the DBCONDITION option to work around this. The DBCONDITION option passes the WHERE clause to the database exactly as you typed it without converting the numeric literal. The following revised Oracle example works as expected:

```
data _null_;
set x.test(dbcondition="where ACCOUNT_NO=12345678901234567890");
run;
ORACLE_26: Prepared:
SELECT  " ACCOUNT_NO" FROM TEST  WHERE (ACCOUNT_NO=12345678901234567890)
```

```
ORACLE_27: Executed:
SELECT statement  ORACLE_26
```

NOTE: There were 1 observations read from the data set X.TEST.

Some SAS/ACCESS interfaces also pass extended precision through the WHERE clause by using the DBSASTYPE option. The following example shows the DBSASTYPE option used with the SAS/ACCESS Interface to Oracle:

```
data _null_;
set x.test(dbsastype=(account_no="char(20)"));
where account_no='12345678901234567890';
run;
```

```
ORACLE_2: Prepared:
SELECT  "ACCOUNT_NO" FROM TEST  WHERE  ("ACCOUNT_NO" =12345678901234567890 )
```

```
ORACLE_3: Executed:
SELECT statement  ORACLE_2
```

NOTE: There were 1 observations read from the data set X.TEST.

\_ Note: The DBCONDITION data set option can be specified anywhere a SAS table reference is made, either in a SAS DATA step or in a SAS procedure.

### ***Performing Joins Involving Database Large Numeric Key Columns***

Joining on database keys is a very common practice. If the keys are indexed, the join can be optimal. However, imprecision on large a numeric can cause unexpected results under these conditions:

If a join key is a large numeric and the join is pushed to the database.

If the numeric is extracted into standard SAS float for the join, the way to ensure that correct results is to transfer the tables into SAS. First converting large decimal columns to a SAS character data type, and then doing the join in SAS on the character version.



Following this list, three examples are shown.

**First example** -- The join is pushed to the database, but transfer of the join result set into SAS causes imprecision in the large integer column.

**Second example** -- The LIBNAME statement option `DIRECT_SQL=NO` forces the join to occur in SAS, but the results are unexpected due to imprecision transferring the large decimal into SAS float.

**Third example:** -- The `DBSASTYPE` option retains precision by transferring the large numeric columns into SAS character columns, which produces the expected result.

For the next example queries, use the following two tables:

Table: TEST

ACCOUNT_NO	EXPENDITURES
925512340233844230	127.20
925512340233844231	3449.99

Table: TEST2

ACCOUNT_NO	CARDHOLDER
925512340233844230	some r. person

**First example** -- Pushes the join to DB2. Imprecision occurs when the large numeric in the join result set transfers to SAS.

```
libname db2 db2 USER=db2 PASSWORD= db2_passwd datasrc=sample in=USERSPACE1;
PROC SQL;
select test2.account_no, cardholder, expenditures
from db2.test,db2.test2
where (test.account_no = test2.account_no);
quit;
```

```
DB2_3: Prepared:
  select test2."ACCOUNT_NO", test2."CARDHOLDER", test."EXPENDITURES" from TEST, TEST2
where
TEST."ACCOUNT_NO" = TEST2."ACCOUNT_NO" FOR READ ONLY
```

```
DB2_4: Executed:
Prepared statement DB2_3
```

```
ACCOUNT_NO          CARDHOLDER          EXPENDITURES
-----
925512340233844224  some r. person          127.20
```

In the second example, the use of `DIRECT_SQL=NO` forces the join to occur in SAS. However, imprecision is introduced when the database large numeric transfers to SAS float. An errant result set is produced in SAS.

```
libname db2 db2 USER=db2 PASSWORD= db2_passwd datasrc=sample in=USERSPACE1
direct_sql=no;
PROC SQL;
select test2.account_no, cardholder, expenditures
from db2.test,db2.test2
where (test.account_no = test2.account_no);
quit;
```

```
DB2_2: Prepared:
SELECT * FROM TEST2 FOR READ ONLY
```

ERROR: The value of the DIRECT\_SQL LIBNAME option is set to NO, NONE, or NOGENSQL. This SQL statement will not be passed to the DBMS for processing.

```
DB2_3: Executed:
Prepared statement DB2_2
```

```
DB2_4: Prepared:
SELECT "EXPENDITURES", "ACCOUNT_NO" FROM TEST FOR READ ONLY
```

```
DB2_5: Executed:
Prepared statement DB2_4
```

ACCOUNT_NO	CARDHOLDER	EXPENDITURES
925512340233844224	some r. person	127.20
925512340233844224	some r. person	3449.99

Notice that the data that's returned doesn't match the data in the example tables.

In this third example, the DIRECT\_SQL=NO option forces the join to occur in SAS as in the second example. However, imprecision is avoided by transferring the database large numeric column to a SAS character column and the correct results are returned.

```
libname db2 db2 USER=db2 PASSWORD= db2_passwd datasrc=sample in=USERSPACE1
direct_sql=no;
PROC SQL;
select test2.account_no, cardholder, expenditures
from
db2.test(dbsastype=account_no="char(18)"),db2.test2(dbsastype=(account_no="char(18)"))
)
where (test.account_no = test2.account_no);
quit;
```

```
DB2_7: Prepared:
SELECT * FROM TEST2 FOR READ ONLY
```

ERROR: The value of the DIRECT\_SQL LIBNAME option is set to NO, NONE, or NOGENSQL. This SQL statement will not be passed to the DBMS for processing.

```
DB2_8: Executed:
Prepared statement DB2_7
```

```
DB2_9: Prepared:
SELECT "EXPENDITURES", "ACCOUNT_NO" FROM TEST FOR READ ONLY
```

```
DB2_10: Executed:
Prepared statement DB2_9
```

ACCOUNT_NO	CARDHOLDER	EXPENDITURES
------------	------------	--------------

-----  
 925512340233844230 some r. person

127.20



**Note:** Both the DIRECT\_SQL=NO option and the data set option disable implicit pass-through and force the join to occur in SAS.

### How Numeric Precision Can Affect WHERE clause or Join Evaluation

If a WHERE clause or join is not passed to the database, it is evaluated in SAS. The values for key columns are extracted from the database, converted to SAS floating point, and the comparison is performed. This opens the possibility of minor precision differences caused by the extraction and conversion. The following example shows data that is loaded to an Oracle decimal column, and contrasts a WHERE clause that is evaluated by the database with a WHERE clause that is evaluated in SAS after data conversion. Unexpected results are returned when the WHERE clause is evaluated in SAS.



**Note:** This numeric precision issue is not seen with DB2 and Teradata whose conversions match SAS values exactly. The rest of this chapter still applies to these relational databases.

For this section's code examples, the following table has been created.

Table: SALES\_LINE\_ITEM

PRODUCT_CODE	SALES_PRICE
31231244	0.95

**First example** --The WHERE clause is passed to and evaluated in the Oracle relational database.

```

/*
 * WHERE clause evaluated in Oracle
 */
PROC SQL;
select product_code from x.sales_line_item where sales_price is null or sales_price
=.95;
quit;
ORACLE_12: Prepared:
SELECT "PRODUCT_CODE", "SALES_PRICE" FROM SALES_LINE_ITEM WHERE (( "SALES_PRICE" IS
NULL ) OR ( "SALES_PRICE" = 0.9500000000000000 ) )
ORACLE_13: Executed:
SELECT statement ORACLE_12
                                PRODUCT_CODE
                                -----
                                31231244

```

The line of test data is successfully retrieved.

In the second example, the use of a non-ANSI syntax prevents the WHERE clause from being passed to the Oracle relational database, and the WHERE clause is evaluated in SAS.

```

/*
 * WHERE clause evaluated in SAS (due to the non-ANSI SQLSALES_PRICE=. construct
 * that prevents passing the WHERE to Oracle).

```

```

*/
PROC SQL;
select product_code from x.sales_line_item where sales_price = . or sales_price =.95;
quit;
ORACLE_15: Prepared:
SELECT "PRODUCT_CODE", "SALES_PRICE" FROM SALES_LINE_ITEM
ORACLE_16: Executed:
SELECT statement ORACLE_15
NOTE: No rows were selected.

```

As seen in this example, evaluating the WHERE clause in SAS rather than the Oracle relational database causes the query to not find the line of test data that the first example successfully retrieved.

### SAS Formats and SAS/ACCESS Default Formatting Rules

SAS augments its two native data types with the use of SAS formats. A SAS format can be associated with a SAS variable to describe how the data should be represented when viewed.

SAS/ACCESS engines automatically associate a default SAS format for the column data read from a database table. These engines use the underlying relational database native data type of a column as the basis for determining the default format value to associate with the variable surfaced to SAS. The default format associations can be overridden by specifying an alternate format for the SAS variable. For example, consider a DB2 table that is created from the following SQL statement:

```

CREATE TABLE ORDERS
( ProductID      INTEGER,
  Description    VARCHAR(40),
  CurrDate       DATE,
  TOD            TIME,
  OrderTime      TIMESTAMP,
  OrderCount     SMALLINT,
  UnitCost       NUMERIC(6,2),
  OrderCost      NUMERIC(8,2),
  CostAdjust     NUMERIC(6,4)
);

```

This table contains a variety of numeric and temporal column types that are all converted to SAS numeric types when the data is brought into SAS. The default SAS formatting can be seen in the following PROC CONTENTS output.

Alphabetic List of Variables and Attributes

#	Variable	Type	Len	Format	Informat	Label
1	ProductID	Num	8	11.	11.	ProductID
2	Description	Char	40	\$40.	\$40.	Description
3	CurrDate	Num	8	DATE9.	DATE9.	CurrDate
4	TOD	Num	8	TIME8.	TIME8.	TOD
5	OrderTime	Num	8	DATETIME25.6	DATETIME25.6	OrderTime
6	OrdCount	Num	8	6.	6.	OrdCount
7	UnitCost	Num	8	8.2	8.2	UnitCost
8	TotCost	Num	8	10.2	10.2	TotCost
9	CostAdjust	Num	8	8.4	8.4	CostAdjust

Here we see the temporal column types: CurrDate, TOD, and OrderTime have default SAS DATE, TIME, or DATETIME format associations. The SAS numeric formats that are associated with the numeric columns reflect the width and scale of the database column definition.

When creating a table, the SAS/ACCESS engines use the SAS formats that are in a SAS data set to determine default database column types that are used for tables it creates in the database. Unlike a relational database numeric data type, SAS format widths must accommodate a potential sign for the number in the data value display. When creating a relational database table based on a SAS data set, this can contribute to an unexpected widening of the data type. Variables that originated as database integer columns might also be widened when rewritten to the database in a new table. A short integer brought into a SAS data set might become a larger integer type when a new database table is created from the SAS data. An integer column might become a larger numeric (decimal) column.

This can be readily seen in the output from the following SAS sample code. Here we create the new database table NEWORDERS from the previously defined ORDERS table, and compare the PROC CONTENTS output with the original table.

```

9          libname dbms db2 db=mysample;
NOTE: Libref DBMS was successfully assigned as follows:
      Engine:          DB2
      Physical Name:  mysample
10
11          options sastrace=",,,d" sastraceloc=saslog no$stsuffix;
12
13          data dbms.NewOrders;
14          set dbms.Orders;
15          run;

```

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

```

DB2_3: Executed: on connection 2
CREATE TABLE NEWORDERS (ProductID NUMERIC(11,0),
                        Description VARCHAR(40),
                        CurrDate DATE,
                        TOD TIME,
                        OrderTime TIMESTAMP,
                        OrdCount INTEGER,
                        UnitCost NUMERIC(7,2),
                        TotCost NUMERIC(9,2),
                        CostAdjust NUMERIC(7,4))

```



**Note:** Compared to the original columns that were created by the initial SQL statement, the column definitions for the numeric types have all widened by a value of 1. For example, the UnitCost column has changed from Numeric (6,2) to a Numeric(7,2) on the new table, and the OrdCount column has changed from a Small Integer to Integer. The changes are shown in bold in the preceding code example.

DB2: COMMIT performed on connection 2.

```

DB2_5: Prepared: on connection 2
INSERT INTO NEWORDERS (ProductID, Description, CurrDate, TOD, OrderTime,
  OrdCount, UnitCost, TotCost, CostAdjust)
VALUES ( ? , ? , ? , ? , ? , ? , ? , ? , ? )

```

The increased column widths and column types are shown in the PROC CONTENTS output.

#	Variable	Type	Len	Format	Informat	Label
1	PRODUCTID	Num	8	12.	12.	PRODUCTID
2	DESCRIPTION	Char	40	\$40.	\$40.	DESCRIPTION
3	CURRDATE	Num	8	DATE9.	DATE9.	CURRDATE
4	TOD	Num	8	TIME8.	TIME8.	TOD
5	ORDERTIME	Num	8	DATETIME25.6	DATETIME25.6	ORDERTIME
6	ORDCOUNT	Num	8	11.	11.	ORDCOUNT
7	UNITCOST	Num	8	9.2	9.2	UNITCOST
8	TOTCOST	Num	8	11.2	11.2	TOTCOST
9	COSTADJUST	Num	8	9.4	9.4	COSTADJUST

To avoid unintentionally widening numeric columns in the output, try one of the following:

- The SAS/ACCESS DBTYPE= option
- Explicitly assert SAS format definitions to override the SAS/ACCESS default column typing

For more information, see the SAS/ACCESS documentation for the DBTYPE option at:

SAS Institute Inc. 2007 SAS online documentation “Data Set Options for Relational Databases DBTYPE= Data Set Option” [Available](http://support.sas.com/onlinedoc/913/getDoc/en/acreldb.hlp/a001371576.htm)  
[http://support.sas.com/onlinedoc/913/getDoc/en/acreldb.hlp/a001371576.htm.](http://support.sas.com/onlinedoc/913/getDoc/en/acreldb.hlp/a001371576.htm)



**Caution:** SAS/ACCESS software default formatting associations are defined only for specific standard SAS formats. Support for user-defined SAS formats is not provided.

### Relational Database VARCHAR Types

Relational database character columns are commonly defined as a VARCHAR, or character varying, data type because they are space-efficient. Many SAS/ACCESS engines use VARCHAR as the default character column type when creating relational database tables.

Because SAS character variables are always fixed length, the following events occur when importing a relational database table that contains VARCHAR columns into SAS:

- SAS reads in the relational database VARCHAR column data from the relational database.
- SAS sets the VARCHAR width to the maximum length for the relational database column.
- SAS pads the empty width of the VARCHAR column values with blanks.

These events can cause the SAS version of the table to occupy more disk space than the original version occupied in the relational database.



**Note:** Because of the default column width and padding involved, when creating a SAS data set from a relational database table that contains VARCHAR columns, you should consider compressing the SAS data set after creation for more efficient memory usage. Trailing blanks might not be stripped out from VARCHARs of an uncompressed SAS data set when writing out from SAS to a relational database, depending on the SAS/ACCESS engine involved. See the SAS/ACCESS documentation for your specific product for more information.

## Date, Datetime, and Other Temporal Values

The internal representation of temporal values in SAS differs significantly from the representation in most database systems. These different representations lead to variations in the rules for their use. It is important to maintain an awareness of these differences, particularly when using these types in filtering expressions. Otherwise, it is easy to write SAS WHERE clauses that cannot be successfully evaluated by a database system.

SAS DATE, TIME, and DATETIME values are stored internally as SAS numeric floats. Because these values are actually numeric, they can readily be used in numeric operations, just as any other numeric variable. They can be compared to or assigned numeric values in the SAS language.

This is generally not true for temporal types on relational database systems. Database columns that are defined as DATE, TIME, or TIMESTAMP types are usually considered distinct non-numeric data types. They often cannot be treated as numeric types in SQL expressions on the database. Referencing these columns in arithmetic expressions or as arguments to arithmetic functions often results in syntax errors from a database.

For example, the following sample SAS code selects the average value of a variable that contains DATE data, both in SAS and in Oracle. Because the SAS date values are backed by an underlying numeric, the average for the variable can be readily obtained using the AVG() aggregate function. The SHIPDATE column in the Oracle table is defined as a DATE type. We can see in SASTRACE output that when the query is passed to Oracle, the database rejects the syntax. The use of the AVG() function is restricted to Oracle numeric data types.



**Note:** These syntax errors can inhibit SQL pass-through to the relational database.

```

23
24          /* average value for column "shipdate" in SAS */
25          PROC SQL;
26          select avg(shipdate) from work.newdates;
27          quit;
NOTE: The PROCEDURE SQL printed page 1.

28
29          /* average value for column "shipdate" in Oracle */
30          PROC SQL;
31          select avg(shipdate) from Dbms.newdates;

ORACLE_1: Prepared: on connection 0
SELECT * FROM NEWDATES

ORACLE_2: Prepared: on connection 0
select AVG(newdates."SHIPDATE") from NEWDATES

ERROR: ORACLE prepare error: ORA-00932: inconsistent datatypes: expected NUMBER got
DATE. SQL statement:  select AVG(newdates."SHIPDATE") from NEWDATES.
ACCESS ENGINE:  SQL statement was not passed to the DBMS, SAS will do the processing.

ORACLE_3: Executed: on connection 0
SELECT statement  ORACLE_1

32          quit;
```

NOTE: The PROCEDURE SQL printed page 2.

The query that incorporates the aggregate function can't be successfully processed by the database because it violates database rules. A simpler query form must be used to retrieve the data. In the preceding example, all the values for the Oracle SHIPDATE column are brought into SAS, converted to SAS numeric form, and these results are used to determine the average. The correct results are returned for the query in this example, but optimal performance cannot be achieved.

### ***Use Consistent Temporal Types in Comparison Expressions***

Some databases systems provide distinct data types for DATE, TIME, or DATETIME column definitions. Other database systems (like some versions of Oracle) might offer only a DATETIME column type to support all three types of data. Please be aware of the underlying column type and use appropriately formatted literals in comparisons. Because the internal representation of any temporal type in SAS is numeric, it does not violate SAS syntax to code a WHERE clause with a formatted literal value that is not appropriate for the variable. However, this generally results in an error from the database.

SAS/ACCESS accommodates comparisons of temporal types directly with a numeric literal. This is a common practice in SAS language code. The literal is converted from a number to database form based on the data type of the column that is referenced in the comparison.

For example, the internal numeric representation of the SAS date literal '10JAN2006'd is the number 16811. This is the number of days from 01Jan1960. A query that uses the numeric representation in a filter might not generate the expected query. The context for the numeric literal is inherited from the associated column type. In the following example, the DB2 column ORDERTIME is a timestamp, so the numeric literal is interpreted as a timestamp.

```
47          PROC SQL;
48          select productid, description from dbms.NewOrders where ordertime = 16811;
```

```
DB2_9: Prepared: on connection 0
SELECT "PRODUCTID", "DESCRIPTION", "ORDERTIME" FROM NEWORDERS WHERE
( "ORDERTIME" = TIMESTAMP({ts '1960-01-01 04:40:11.000000' }) ) FOR READ ONLY
```

This practice of using numeric temporal literals can contribute to ambiguities in more complex expressions. This can inhibit the passing of filters to the database.

Consider the following equivalent SAS WHERE clauses:

```
WHERE DATEVAR = '01JAN1990'd + 30;
```

```
WHERE DATEVAR = 10970 + 30;
```

In the SAS language these are equivalent expressions. Because the date variable is stored in a numeric format, the variable and literals coded in these expressions are all consistent in SAS. This is not true for database systems that support distinct temporal types. In the second clause, the SAS/ACCESS engine would not be able to differentiate between a literal intended as a date value and one intended as an interval value. For both readability and to avoid issues of ambiguity, it is generally a better practice to use the appropriately formatted literals as shown in the first of the preceding two examples in comparisons.



**Avoid Use of Temporal in Arithmetic Expressions or as Arguments to Arithmetic SAS Functions**

SAS SQL statements that include arithmetic expressions with dates often cannot be completely passed to a database and require processing in SAS. SAS/ACCESS engines currently do not include WHERE clause expressions that incorporate temporal type values in arithmetic operations; those are evaluated in SAS. Sometimes this limitation can be avoided by pre-evaluating part of the expression in a SAS macro variable definition. The SAS macro variable can then be used in a direct comparison in SAS WHERE clauses, or assignments in a SAS DATA STEP.

Here is an example of the preceding information. The following two queries contain a filter expression to return account numbers where a DATE column value is less than one week from the current date. The first query uses an arithmetic expression in the WHERE clause, and the second query uses the macro variable &TARGET\_DATE to allow a direct comparison. The filter that contains the arithmetic expression is not present in the query that is passed to the database, so the entire table is retrieved.

```
20      option sastrace=",,,d" sastraceloc=saslog no$stsuffix;
21
22      PROC SQL;
23      select account, ship_date from dbms.neworders where ship_date < today() +
7;
```

```
DB2_2: Prepared: on connection 0
SELECT  "ACCOUNT", "SHIP_DATE" FROM NEWORDERS FOR READ ONLY
```

```
24      quit;
NOTE: The PROCEDURE SQL printed page 1.
NOTE: PROCEDURE SQL used (Total process time):
      real time          0.89 seconds
      cpu time           0.73 seconds
```

The time shown represents the time to import the entire relational database table into SAS and apply the WHERE clause from within SAS.

```
25
26      /* macro variable for filter */
27      %let target_date=%eval(%sysfunc(today()) + 7 );
28      PROC SQL;
29      select account, ship_date from dbms.neworders where ship_date
<&target_date;
```

```
DB2_5: Prepared: on connection 0
SELECT  "ACCOUNT", "SHIP_DATE" FROM NEWORDERS WHERE ( "SHIP_DATE" < DATE({d '2006-
03-16' }) ) FOR READ ONLY
```

```
30      quit;
NOTE: The PROCEDURE SQL printed page 2.
NOTE: PROCEDURE SQL used (Total process time):
      real time          0.09 seconds
      cpu time           0.03 seconds
```

Here, the time shown represents the time to pass the WHERE clause to the relational database, allow the relational database to perform the WHERE clause filter, and then import into SAS only the results of the WHERE clause.

## Large Object Data Types

Database large object (LOB) column types can be used to handle very large (several gigabytes), variably sized data. The content is primarily textual or binary in nature. Character large objects (CLOBs) columns are used for storing textual content such as XML. Binary large object (BLOB) columns are used for storing binary information such as image or video.

These large database string types do not directly map to SAS native data types. SAS/ACCESS engines might retrieve LOB data as a SAS character variable. By default, this column data would be truncated at 1024 characters in length. The maximum length of a SAS character variable (32767 bytes) can be retrieved by specifying the `DB_MAX_TEXT=` option.

---

## Basic Tracing

---



---

### Trace Information Shows Where Work Performed and Amount of Time Taken

---

When working with an underlying relational database source using a SAS/ACCESS engine, it is critical for performance to push as much work as possible to the database. For time-intensive extract processes, verification that the expected query and filtering is being passed to the database should be performed. This section explains how to turn on tracing options so you can see what SQL was generated, and what SQL was pushed down to the database.

SAS/ACCESS engines provide a wealth of tracing information through the use of the SAS option `SASTRACE=`. In this section we highlight some of the `SASTRACE=` settings that are possible to use. For further details on the variety of tracing information provided by SAS/ACCESS engines, see the appropriate SAS/ACCESS documentation.

#### Basic SASTRACE SQL Trace

The primary mechanism in SAS for showing what SQL is passed to an underlying database by a SAS/ACCESS engine is the SAS option `SASTRACE=,,,d`. Using this option setting causes all the SQL or internal API call information that a SAS/ACCESS engine passes to the underlying database to be displayed in the SASLOG output. Any database return codes or messages that are returned during the processing is also displayed in the SASLOG output.

To enable basic SQL tracing, specify the following options in your SAS program code:

```
Option SASTRACE=",,,d" no$stsuffix SASTRACELOC=saslog ;
```

The `NO$STSUFFIX` system option formats the log to be more easily read. The `SASTRACELOC` option sends the trace to the SASLOG output.



**Note:** Use of the `NO$STSUFFIX` system option is recommended for easier reading.

To disable the tracing, clear the `SASTRACE` specification as follows:

```
Option SASTRACE=",,, " ;
```

SAS/ACCESS engines send SQL for *Prepare* and *Execute* operations to the relational database. *Prepare-only* SQL statements are generally inexpensive to process and can be routinely performed during the information gathering processes that are required by the SAS/ACCESS engines. An example is the SQL *Prepare* performed on a “SELECT \* from tablename” statement to determine whether a table exists and to retrieve column metadata. *Executed* SQL statements might contain WHERE filters, joins, or other complex queries that are passed to the database. The *Executed* SQL statement is used to resolve the requested operation.

Errors or warnings from internal processes that are not normally surfaced to the user might be surfaced in the SASTRACE output. These are not a cause for concern. They might merely be indicative of a decision point made in an internal process.

To illustrate the aspects of SAS/ACCESS SQL tracing, consider the following example code. A SAS SQL query is coded that contains the EXCEPT ALL set operator that uses Oracle database tables.

```
libname dbms oracle path=alien user=scott pw=tiger;

options sastrace=",,,d" sastraceloc=saslog no$stsuffix;

PROC SQL ;
select distinct i from dbms.table1 where i < 100
except all
select distinct i from dbms.table2 where i > 2 ;
quit;
```

In the Oracle SQL syntax, the standard EXCEPT operator is supported as MINUS. However, Oracle doesn't support a MINUS ALL operator. When the SQL statement is submitted for *Prepare* on the database, Oracle returns a prepare error, and the SQL statement is not executed. Subsequent SAS processing of the query involves breaking it into smaller component parts. Individual queries for the tables are successfully submitted to Oracle, and the results for the set operation are performed by PROC SQL.

The SASTRACE output shows the *Prepare-only* and *Executed* SQL that is submitted to Oracle. The SQL that is *Executed* by Oracle is marked in bold in the following SASLOG output.

```
18
19      libname dbms oracle path=alien user=scott pw=XXXXXX;
NOTE: Libref DBMS was successfully assigned as follows:
      Engine:          ORACLE
      Physical Name:  alien
20
21      options sastrace=",,,d" sastraceloc=saslog no$stsuffix;
22
23      PROC SQL ;
24
25      select distinct i from dbms.table1 where i < 100
26      except all
27      select distinct i from dbms.table2 where i > 2 ;

ORACLE_1: Prepared: on connection 0
SELECT * FROM TABLE1

ORACLE_2: Prepared: on connection 0
SELECT * FROM TABLE2

ORACLE_3: Prepared: on connection 0
select distinct table1."I" from TABLE1 where TABLE1."I" < 100
minus all
select distinct table2."I" from TABLE2 where TABLE2."I" > 2

ERROR: ORACLE prepare error: ORA-00928: missing SELECT keyword. SQL statement:  select
distinct table1."I" from TABLE1 where TABLE1."I" < 100 minus all select distinct
table2."I" from TABLE2 where TABLE2."I" > 2.
ACCESS ENGINE:  SQL statement was not passed to the DBMS, SAS will do the processing.
```

```

ORACLE_4: Prepared: on connection 0
select distinct TABLE1."I" from TABLE1 where TABLE1."I" < 100

ORACLE_5: Prepared: on connection 0
select distinct TABLE2."I" from TABLE2 where TABLE2."I" > 2

ORACLE_6: Executed: on connection 0
SELECT statement ORACLE_4

ACCESS ENGINE: SQL statement was passed to the DBMS for fetching data.

ORACLE_7: Executed: on connection 0
SELECT statement ORACLE_5

ACCESS ENGINE: SQL statement was passed to the DBMS for fetching data.
28
29          quit;
NOTE: The PROCEDURE SQL printed page 1.
NOTE: PROCEDURE SQL used (Total process time):
      real time          0.53 seconds
      cpu time           0.39 second

```

For other examples of the `SASTRACE=` option for SQL tracing, see the [“Error! Not a valid bookmark self-reference.”](#) section earlier in this document.

### ***SASTRACE Timers: How to Account for Time Spent in the Database***

The `SASTRACE` facility also contains a “timer” option. When enabled, it causes SAS/ACCESS engines to capture and display the amount of time that is spent performing various database activities. If you are concerned with the performance of a SAS step that interacts with the database, `SASTRACE` timers provide a first step in your investigation.

It is useful to understand how timers are implemented. For example, consider the “SQL execute” timer that reports the amount of time that the database consumes when executing SQL submitted from SAS. SAS/ACCESS starts the execute timer just before it calls the database API to execute an SQL statement. Immediately upon return from the call, it captures the time elapsed.

Timers capture the following interactions with the database:

- SQL prepare
- SQL describe
- SQL execute
- Row fetch
- SQL row insert
- SQL row update
- SQL row delete

For each of the items in the preceding list, the execute timer includes only the following items:

- Time spent in the database client library

- Network time sending SQL text to the database server
- Execute statement processing time on the database
- Time spent returning the status of the SQL operation to the database client library.

Turn timers on by using the following line of code:

```
option SASTRACE=",,,s" no$stsuffix SASTRACELOC=saslog ;
```

Turn timers off by using the following line of code:

```
option SASTRACE=",,,";
```

Combine timers and basic SQL trace by using the following line of code:

```
option SASTRACE=",,,sd" no$stsuffix SASTRACELOC=saslog ;
```

A typical SAS step submits SQL to the database and retrieves resulting rows. For this example, the SQL execute and row fetch timers are likely the largest values. If so, then tuning the entire SAS step for better performance might involve tuning SQL execute and row fetch to perform better. For example:

```
libname dbms db2 database=mysample schema=schema;
NOTE: Libref DBMS was successfully assigned as follows:
      Engine:          DB2
      Physical Name:   mysample

options sastrace=",,,ds" sastraceloc=saslog no$stsuffix fullstimer;

PROC SQL noprint;
select Component_Id, expected_completion, completion_date from dbms.project2
where Complete_code = 'Pending' or lead = 'Valance';

DB2_1: Prepared: on connection 0
SELECT * FROM schema.PROJECT2 FOR READ ONLY

DB2_2: Prepared: on connection 0
SELECT  "COMPONENT_ID", "EXPECTED_COMPLETION", "COMPLETION_DATE", "COMPLETE_CODE",
"LEAD" FROM schema.PROJECT2 WHERE ( ( "COMPLETE_CODE" = 'Pending' ) OR ( "LEAD" =
'Valance' ) ) FOR READ ONLY

DB2_3: Executed: on connection 0
Prepared statement DB2_2

Summary Statistics for DB2 are:
Total row fetch seconds were:          0.037036
Total SQL execution seconds were:      5.053354
Total SQL prepare seconds were:        0.020981
Total SQL describe seconds were:       1.186427
Total seconds used by the DB2 ACCESS engine were 6.694703

quit;
NOTE: PROCEDURE SQL used (Total process time):
      real time          7.66 seconds
      user cpu time      0.03 seconds
      system cpu time    0.12 seconds
      Memory              393k
      OS Memory           3804k
```

Timestamp

3/27/06 10:14:45 AM



**Note:** The SAS log is trimmed and rearranged slightly for clarity.

Here is a basic analysis of this step. The standard STIMER output at the bottom tells us that the step used 7.66 seconds. The SASTRACE timer output shows .037 seconds were used fetching rows from Db2, 1.18 seconds were used in database describe processing, and 5.05 seconds were used in executing the query.

Improving step performance begins by comparing the SASTRACE timer output to the time used for the total step, which is represented by "real time" in the SAS STIMER output. If a substantial proportion of time is used in database operations, tuning should begin with SAS/ACCESS and the database. If comparatively little time is used in database operations, tuning should begin with operations that occur in SAS.

The SASTRACE timer output also provides the total time that the SAS/ACCESS engine used for the process. In the above example, this value was 6.694703 seconds. This number shows the amount of time the SAS procedure or DATA step used the engine. It includes the amount of time that was used during calls to the database, and the amount time that was used processing in SAS. Often, this time value might be close to the STIMER value for the step.

For more information about the SASTRACE option, see the SAS/ACCESS documentation.

### SASTRACE Threaded Read trace

When using SAS/ACCESS Threaded Read techniques to improve throughput for data extracts, it is important to evaluate the process by using the Threaded Read tracing that is provided by the SAS/ACCESS engine. Threaded Read trace information is needed in order to get the following information:

- The operation was performed using threaded reads.
- The number of threads that were used for the operation.
- The number of observations that each thread returned.
- The exit code of the thread, if a failure occurs.

To enable basic tracing for threaded reads, specify the following options in your SAS program code:

```
Option SASTRACE=",,t," no$stsuffix SASTRACELOC=saslog ;
```



**Note:** Use of the NO\$ST\_SUFFIX system option is recommended for easier reading.

To obtain Threaded Read trace information with basic SQL tracing, specify the following options in your SAS program code:

```
Option SASTRACE=",,t,d" no$stsuffix SASTRACELOC=saslog ;
```

To disable the tracing, clear the SASTRACE option by specifying the following line of code:

```
Option SASTRACE=",,, " ;
```

The following is an example SAS program that shows SAS/ACCESS Threaded Read tracing. A SAS data set is created from a database table using threaded reads of the database data. The SAS/ACCESS data set option DBSLICEPARM= specifies that threaded reads on two separate threads are requested. The trace output shows that two threads were used to extract the data. In this example, 276 rows are returned on the first thread and 262 rows are returned on the second thread.

```
5
6      libname dbms oracle user=scott pw=XXXXX path=alien;
NOTE: Libref DBMS was successfully assigned as follows:
      Engine:          ORACLE
      Physical Name:  alien
7
8      option sastrace=",,t,";
9
10     data work.group_bg;
11     set dbms.bgtable (dbsliceparm=(all, 2));
12     run;
```

```
ORACLE: DBSLICEPARM option set and 2 threads were requested
ORACLE: No application input on number of threads.
ORACLE: Thread 1 contains 276 obs.
ORACLE: Thread 2 contains 262 obs.
ORACLE: Threaded read enabled. Number of threads created: 2
```

```
NOTE: There were 538 observations read from the data set DBMS.BGTABLE.
NOTE: The data set WORK.GROUP_BG has 538 observations and 4 variables.
NOTE: DATA statement used (Total process time):
      real time          0.25 seconds
      cpu time           0.01 seconds
```

For other examples of SASTRACE= Threaded Read tracing, see the [Threaded Reads](#) section.

---

## Implicit Pass-Through

---

### Introduction

---

Within SAS, SQL syntax submitted to a database might be generated either by the SQL Implicit pass-through Facility or by a SAS/ACCESS engine. This section discusses both of these components to aid in understanding which component is involved in generating SQL in a given situation.

In addition to exploring how SQL is generated and passed to a database using SQL implicit pass-through, this section provides guidelines to increase the potential for success using SQL implicit pass-through.

### SAS and SQL Generation

---

When a SAS procedure or the SAS DATA step uses a SAS/ACCESS engine to access database data, the engine is called in the same way as the Base SAS engine. SAS engines provide a well-defined API to be used by SAS procedures and the SAS DATA step. A SAS procedure has minimal awareness of the



underlying data, or the mechanisms used to access the data. They are insulated from this by the SAS engine architecture. When a SAS/ACCESS engine is used, the engine generates database-specific SQL to honor the requests made by the SAS procedure or the DATA step.

The SAS engine architecture is designed to provide this support using single data source access patterns, and is not designed to provide support for complex multi-table SQL queries in one step. In order to read data, the SAS procedure calls through to the SAS engine API to:

- OPEN a data set
- READ each row
- CLOSE the data set

A join of two SAS data sets using the SAS engine API requires independent open and retrieval of both data sets into SAS. This join of two different data sets in SAS is often referred to as a Heterogeneous Join.

Like other SAS procedures, the SAS SQL procedure primarily uses the SAS engine API to access data. However, because PROC SQL operates on SQL, it can readily provide more efficient mechanisms to pass SQL to a database. In addition to using the SAS engine API directly, SAS SQL also currently supports two other methods that allow passing more complex SQL queries through a SAS engine to an underlying database.

These methods are:

- The SQL pass-through facility (explicit pass-through)
- SQL implicit pass-through (implicit pass-through)

These two methods provide a means of passing more complex SQL queries directly to a database than is possible through the SAS/ACCESS engine API. A join of multiple tables on a single database instance is considered a *Homogeneous Join*. Passing a homogeneous join query to a database through a SAS/ACCESS engine requires the use of one of these two methods.

Explicit pass-through offers the user complete control of the SQL syntax sent to a database, but requires a query to be written in the SQL syntax of the database.

Implicit pass-through can allow much of the performance gain possible through explicit pass-through, but allow users to write queries in the single SAS SQL syntax. However, implicit pass-through cannot support passing the unlimited variety of SQL that is possible with explicit pass-through, or provide user control over the exact syntax of the query sent to the database.

The following help identify the situations where the SAS/ACCESS engine or the SQL implicit pass-through Facility generates the SQL syntax that is submitted to the database.

### **SQL Generation through SAS/ACCESS**

SAS/ACCESS engines internally generate SQL syntax in order to implement the SAS engine APIs used by SAS procedures. The SQL generation can vary from one engine to another, but it usually reflects the functional characteristics for the SAS engine APIs.

There are many benefits to this design, allowing:

- SAS procedures to use a consistent API for accessing data.
- Developers to write SAS program code that operates on many data sources without requiring a detailed understanding of the underlying data.
- Users to easily perform operations that combine heterogeneous (divergent) data sources.
- SAS analytics to be applied directly to database data as easily as native SAS data stores.

The SAS engine API was designed to support native SAS data stores and not the wide range of queries possible in relational databases. Data transfer time and the single-threaded nature of some SAS operations can significantly under perform when compared to performing data manipulation in-place on a powerfully parallel database system.

SAS/ACCESS engines have incorporated many features to improve performance, including buffering of the data read from the database, parallel threading of database queries, exploiting a database's bulk loading facilities, methods for subsetting the data to minimize transfer, and techniques for performing 'indexed' type query operations.

#### ***Normal SAS/ACCESS Use with SAS Procedures or SAS DATA Steps***

We can examine the SQL generated by a SAS/ACCESS engine on behalf of the standard SAS engine APIs by using the SASTRACE= system option. This also illustrates the functional characteristics of the calls made through the standard SAS engine API.

In the following SAS DATA step example, we want to create a table in Oracle that contains a subset of the information kept in an Employees table in the same Oracle instance. Only the employees from a specific department are needed in the newly created table, so we provide a WHERE clause to restrict the query to return those rows.

```
libname oracle oracle path=alien user=scott pw=tiger;

option sastrace=",,,d" sastraceloc=saslog no$stsuffix;

data oracle.ship_dept;
keep EMPID FIRSTNAME LASTNAME SALARY;
set oracle.employees;
where dept = 'SHP013';

run;
```

Executing this SAS DATA step code causes several processes in the SAS/ACCESS to Oracle engine to be called. The engine generates a series of SQL statements in order to complete the requests.

1. The primary and target tables are queried to determine existence and collect column metadata.
2. A CREATE TABLE statement is generated to create the unpopulated target table.
3. A filtering query is generated to obtain the data in the Employees table. This query is used to fetch each qualifying row from the Employees table.

- An INSERT statement using host variables is built to populate the target table. As each row is read from the Employees table, the data is moved into the host variable storage and the insert is repeatedly executed.

These four sets of SQL statements are marked in the following SAS Log output from execution of the SAS DATA step code:

```

5
6      libname oracle oracle path=alien user=scott pw=XXXXX;
NOTE: Libref ORACLE was successfully assigned as follows:
      Engine:          ORACLE
      Physical Name:  alien
7
8      option sastrace=",,,d" sastraceloc=saslog no$stsuffix;
9
/* Begin Set 1, Part A          */
ORACLE_1: Prepared:
SELECT * FROM EMPLOYEES
/* End Set 1, Part B          */

10     data oracle.ship_dept;
11     keep DEPT EMPID FIRSTNAME LASTNAME SALARY;
12     set oracle.employees;
13     where dept = 'SHP013';
14
15     run;

/* Begin Set 1, Part B          */
ORACLE_2: Prepared:
SELECT * FROM SHIP_DEPT
/* End Set 1, Part B          */

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

/* Begin Set 2          */
ORACLE_3: Executed:
CREATE TABLE SHIP_DEPT(EMPID NUMBER (16),SALARY NUMBER ,DEPT VARCHAR2 (6),LASTNAME
VARCHAR2 (18), FIRSTNAME VARCHAR2 (15))
/* End Set 2          */

/* Begin Set 3          */
ORACLE_4: Prepared:
SELECT "EMPID", "HIREDATE", "SALARY", "DEPT", "JOBCODE", "SEX", "BIRTHDATE",
"LASTNAME","FIRSTNAME", "MIDDLENAME", "PHONE" FROM EMPLOYEES WHERE ("DEPT" =
'SHP013' )
/* End Set 3          */

ORACLE_5: Executed:
SELECT statement  ORACLE_4

/* Begin Set 4          */
ORACLE_6: Prepared:
INSERT INTO SHIP_DEPT (EMPID,SALARY,DEPT, LASTNAME,FIRSTNAME) VALUES
(:EMPID,:SALARY,:DEPT,:LASTNAME,:FIRSTNAME)
/* End Set 4          */

NOTE: There were 3 observations read from the data set ORACLE.EMPLOYEES.
      WHERE dept='SHP013';

ORACLE_7: Executed:
INSERT statement  ORACLE_6

```

NOTE: The data set ORACLE.SHIP\_DEPT has 3 observations and 5 variables.  
 NOTE: DATA statement used (Total process time):  
     real time                  0.78 seconds  
     cpu time                    0.07 seconds

## Using SAS PROC SQL

In addition to using the SAS/ ACCESS engine API directly, SAS SQL also currently supports two other methods that allow passing more complex SQL queries through a SAS engine to an underlying database. These methods are:

- The SQL pass-through facility (explicit pass-through)
- SQL implicit pass-through (implicit pass-through)

Explicit pass-through can be used to issue SQL queries or executable SQL statements directly to a database from PROC SQL. No SQL generation is performed internally in SAS for explicit pass-through. When using this facility the user supplies the exact SQL syntax passed to the database.

Implicit pass-through is an optimization technique in PROC SQL for queries using a single SAS/ACCESS engine. SQL implicit pass-through will attempt to reconstruct the textual representation of a query in database syntax, from PROC SQL's internal SQL tree representation. If successfully reconstructed, the textualized query will then be passed to the database through the SQL explicit pass-through facility.

### ***Explicit SQL on Homogenous Database Data***

The SQL pass-through facility (commonly referred to as 'explicit pass-through') can be used to issue SQL queries or executable statements directly to a database from PROC SQL. When using this facility the user supplies the exact SQL syntax passed to the database. This SQL is not modified by SAS in any way.

Explicit pass-through can permit access to certain database operations that are not supported by the SAS engine API. Examples of this might be creating database indexes or views. It is also useful for situations where the internally generated SQL is not optimal or when a particular query is performance-critical. Explicit pass-through can also be used to submit SQL syntax that is not supported in SAS, allowing the use of SQL syntax extensions supported by a database.

This is generally the most efficient mechanism available in SAS for issuing complex queries to, or manipulating database data. For more information about SQL explicit pass-through, see the "[Explicit Pass-Through](#)

" section later in this paper.

### ***SAS SQL on Homogenous Database Data***

PROC SQL implicit pass-through is an optimization technique in PROC SQL for queries that involve a single SAS/ACCESS engine. The SQL statement is converted into an internal SQL query tree representation. This tree form greatly facilitates the ability to apply optimizations to the query for normal processing.



**Note:** This tree form generally will be equivalent but not identical to the query that was coded.

PROC SQL will attempt to reconstruct a textual representation of the query in database-specific syntax, from its internal SQL query tree representation. If successfully reconstructed, the SQL explicit pass-through facility then passes the textualized query to the database. If this succeeds, the SQL query executes on the database and only the results of the query are returned to SAS. When successful, SQL implicit pass-through can greatly improve the performance of a query's execution when compared with normal processing in SAS.

Although there is no guarantee that a given SQL query can be passed to a database, the implicit SQL optimization will try to do everything it can to generate SQL that will pass to the database. Without tracing, implicit pass-through success or failure is not surfaced to the user.

To determine the success or failure of implicit pass-through for a query, you must examine the SQL that the engine submits to the database. For more information about how to use the SASTRACE mechanism, see the section "[Trace Information Shows Where Work Performed and Amount of Time Taken](#)"

There are a variety of reasons why the implicit pass-through optimization technique might not succeed. Some of these reasons include:

- Variations in SQL syntax between SAS and the database, such as reserved words or other implementation differences.
- Use of SAS specific constructs or SQL extensions in a query. Examples of this would be use of a SAS function that has no database equivalent or SAS dataset options on a table reference.

Even optimizations to internal representations of the query can affect the ability to generate database-specific queries.

Whenever the SQL implicit pass-through optimization is not successful, PROC SQL will continue to process the query internally using the standard SAS Engine API directly. The query is executed, but the performance of the query might not be optimal.

#### ***SAS SQL on Heterogeneous Database Data***

The ability to easily perform join operations that involve heterogeneous data sources is a powerful feature of SAS SQL. However when a PROC SQL query references different sources, such as with a join between SAS and database tables, entire PROC SQL query cannot be passed to the database by the SQL implicit pass-through facility. One or more of the joins are performed in SAS by PROC SQL.

This standard heterogeneous join can perform poorly if very large database data is read into SAS where the join is then performed.

For performance-critical code steps, or cases where a join might be executed more than once, there are three choices that might increase performance:

- Copy database data into SAS data sets, and join SAS data sets.
- Copy SAS data to a database table and perform the join in the database
- Enhance join performance in SAS by sorting on the join key.

Copying database data to SAS data sets is similar to performing standard heterogeneous join – a large volume of database data might be transferred to SAS. In most cases this approach is worthwhile only when multiple joins reference the database data. By copying to a SAS data set, you incur the data transfer cost only once. You change your joins to reference SAS copies of the tables, and your joins become SAS to SAS joins. This choice receives no further treatment in this paper.

For more details, see the “Heterogeneous Joins

” section under “Advanced Topics” later in this document.

#### ***Heterogeneous Joins Using Data Integration Studio SQL Join Transformations***

In the release of SAS Data Integration Studio 3.4, SQL join transformation is new and improved. By setting a few properties, SQL join transform will automatically create the code to upload one or more sources to the database when generating the explicit pass-through SQL. The table options available for uploading the table are as follows:

- Update library before SQL – a defined library is selected that points to where you want to upload the table.
- Enable case sensitivity on upload table – turns quoting on the columns and tables to handle case sensitivity.
- Enable special characters on upload table – turns quoting in the columns and tables to handle special characters.
- Pre-upload action – if the table exists before upload do you want to delete the table or truncate the table.
- Use bulk load for uploading – use the BULKLOAD= option to upload the table.
- Bulk load options – any additional bulk load options,
- Additional upload options – any additional upload options to then be added,

### **Criteria for Implicit Pass-Through**

---

Not all PROC SQL queries are candidates for implicit pass-through. Certain criteria must be met before implicit pass-through would be attempted for a query.

In general, only homogeneous queries are considered candidates for implicit pass-through. These are

queries for which the referenced tables all use the same SAS/ACCESS engine and are from the same database instance. Single or multiple table homogeneous queries that contain one of the following constructs will cause an implicit pass-through processing attempt.

- Select DISTINCT keyword
- Query contains an SQL Aggregate function.
- Query contains a GROUP BY clause
- Query contains a HAVING clause
- Query contains an ORDER BY clause
- Query is an SQL join
- Query involves a Set operation, other than an OUTER UNION
- Query with a WHERE clause containing a Subquery



**Note:** If a heterogeneous query contains a mixture of homogenous database table references and references to Base SAS tables, some portions of the query might be evaluated for implicit pass-through processing.

### Select DISTINCT

The DISTINCT keyword is used to restrict result sets to unique values. In many cases, these result sets will be much smaller than the total number of rows in the table. By passing this processing to a database, only the unique observations would be retrieved into SAS. This can greatly improve the overall time to process the query. Consider the following example:

```
select distinct state from Oracle.Credit_Account;
```

In the above example, the query is used to determine how many states are represented by a group of credit card customers. The requested result set could have a maximum of 50 rows. If the Credit\_Account table contains 5 million rows, you can quickly see the advantage of passing this query to the database for processing. The advantage of transferring a maximum of 50 rows across a network is that it is much faster than transferring all 5 million rows into SAS and processing those results for distinct values.

### Query Contains an SQL Aggregate or COALESCE Function

Including any of the standard SQL aggregate functions (MIN, MAX, SUM, COUNT, FREQ, AVG, MEAN) or the COALESCE function, will cause implicit pass-through processing for that query. Processing of these functions can be more efficiently performed by a database. In the case of a query such as the following:

```
Select COUNT(*) from DB2.Customers;
```

This passes the COUNT function in the query submitted to the database has the obvious advantage of requiring a single row to be transferred from the database into SAS, of every row in the table.

### Query Containing a GROUP BY Clause

Queries containing an SQL GROUP BY clause will also cause implicit pass-through attempts to allow the database to perform the BY groupings. However, applying SAS formatting to the GROUP BY variable can inhibit the ability to pass the query. If a GROUP BY clause is specified on a query without an associated aggregate function reference, the GROUP BY clause might be into an ORDER BY clause.

### Query Containing a HAVING Clause

Queries containing an SQL HAVING clause will also cause implicit pass-through attempts.

### Query Containing an ORDER BY Clause

Queries containing an SQL ORDER BY clause will also cause implicit pass-through attempts. If the implicit pass-through attempt for some reason is unsuccessful, the SAS/ACCESS engine can be used to pass an ORDER BY clause of single table extracts.

### Query Contains an SQL Join

Any SQL join query involves multiple tables, even a join of a table to itself. Often the results of a join operation will be smaller than the combined input tables. Even for the case of a FULL OUTER join query, processing the multiple tables in a single step is preferred.

The performance cost of transmitting the rows from all the joined tables into SAS for processing can be formidable. If the join can be passed to the relational database for processing, only the results of the joined tables need to be transmitted back to SAS. Implicit pass-through can pass both inner and outer joins to the database for processing.

Implicit pass-through supports all SAS SQL inner join queries in ANSI 1992 INNER join syntax, with the inner join conditional contained in the WHERE clause. This syntax is supported by all SAS/ACCESS target database SQL syntaxes. For example:

```
select * from dbms.CUSTOMER, dbms.SALES, dbms.ORDERS
```



```
Where customer.custid = orders.account and
      sales.salesrep = 'SMITH';
```

Implicit pass-through does not generate SQL inner join queries using either the INNER join keyword, or use of an ON clause for the join conditionals. In the following two equivalent inner join queries, the generated SQL syntax that is sent to the database is identical. Neither of the generated queries contains either the INNER join keyword or an ON clause.

```
86          PROC SQL;
87
88          select customer.custname, customer.location,
89                 orders.invoice, orders.salesid
90          from dbms.CUSTOMER, dbms.ORDERS
91          where customer.custid = orders.account and
92                 orders.salesid in (315,215);
```

```
DB2_3: Prepared: on connection 0
      select CUSTOMER."CUSTNAME", CUSTOMER."LOCATION", ORDERS."INVOICE",
      ORDERS."SALESID" from CUSTOMER, ORDERS where (CUSTOMER."CUSTID" =
      ORDERS."ACCOUNT") and ( ORDERS."SALESID" in (215, 315) ) FOR READ ONLY

ACCESS ENGINE:  SQL statement was passed to the DBMS for fetching data.
```

```
93
94          Select customer.custname, customer.location,
95                 orders.invoice, orders.salesid
96          from dbms.CUSTOMER inner join dbms.ORDERS
97          on customer.custid = orders.account and
98                 orders.salesid in (315,215);
```

```
DB2_7: Prepared: on connection 0
      select CUSTOMER."CUSTNAME", CUSTOMER."LOCATION", ORDERS."INVOICE",
      ORDERS."SALESID" from CUSTOMER, ORDERS where (CUSTOMER."CUSTID" =
      ORDERS."ACCOUNT") and ( ORDERS."SALESID" in (215, 315) ) FOR READ ONLY

ACCESS ENGINE:  SQL statement was passed to the DBMS for fetching data.
```

```
99          quit;
```

This form of inner join is widely supported across most databases, but might inhibit passing some queries that combine inner and outer Joins. See the “Guidelines for Implicit SQL” section of this paper for more details.

Some releases of relational databases (for example, Oracle 8) do not support ANSI 1992 standard outer join syntax. Implicit pass-through has the ability to produce outer join queries in the non-standard database-specific SQL join syntax. While implicit pass-through can produce non-standard outer join syntax for these engines, there are some restrictions:

SAS SQL outer join queries using the SAS/ACCESS Interface to Sybase that contain more than two table references with additional WHERE-clause filters are not passed to the database. This restriction exists because Sybase can return different results than SAS when a WHERE clause is applied to an outer join with more than two tables.

For outer join queries with the SAS/ACCESS Interface to Informix, pass-through of outer joins that reference only two tables with no additional WHERE clauses will be attempted. Informix uses a WHERE clause instead of an ON clause for join conditionals. This makes integration of ON clauses and WHERE clauses specified in a SAS SQL query difficult to convert into an Informix-specific query.

For queries that use the SAS/ACCESS Interface to ODBC, passing of outer joins that contain more than two table references will be attempted as long as there are no inner joins specified in the query. This restriction exists due to limitations in ODBC outer join syntax.

Passing down FULL outer join queries are not attempted for Oracle, Sybase, and Informix. Their non-standard outer join syntaxes do not support FULL outer joins.

### **Query Involves a Set Operation**

Using a Set operator (such as UNION, INTERSECT, or EXCEPT) in a query will cause implicit pass-through optimization attempt. These queries are considered multiple table queries.

### **Query with a WHERE clause Containing a Subquery**

With SAS 9.1.3 Service Pack 4, queries that contain a subquery reference in a WHERE clause will also cause an implicit pass-through optimization attempt.

Single table queries that do not contain one of the following can generally be processed as efficiently using the SAS/ACCESS engine API directly:

- Select DISTINCT keyword
- Query contains an SQL Aggregate function.
- Query contains a GROUP BY clause
- Query contains a HAVING clause
- Query contains an ORDER BY clause

SAS/ACCESS options can be used to further enhance performance of these queries. These types of queries will not cause implicit pass-through optimization.

If a table name is referenced more than once in a query, the query is technically a multiple table query.

For example, the following query references the SALESREP table twice in a self join operation:

```
Select emp.NAME
from DBMS.SALESREP emp, DBMS.SALESREP mgr
WHERE emp.EMPLID = mgr.MANAGER
```

This query would be considered a multiple table query even though it only references the SALESREP table. Normal processing of this query in SAS would require joining the results from two separate queries of the SALESREP table. However, because this query causes the implicit pass-through optimization, it can be successfully processed in a single step on the database.

## Inhibitors to Implicit Pass-Through

---

There are certain SQL constructs that will automatically disqualify a query from implicit pass-through processing. The following constructs will inhibit implicit pass-through processing regardless of the engine used. When these are encountered, implicit pass-through processing for the statement will be terminated before performing a database prepare on the textualized query. For performance-critical SQL steps, use of these constructs should be avoided if possible.

- Heterogeneous queries
- Queries that incorporate explicit pass-through statements
- Queries that use SAS data set options
- Queries that contain the SAS OUTER UNION operator
- Specification of a SAS Language function that isn't mapped to a database equivalent by the engine
- Queries that require REMERGING in PROC SQL
- Specification of ANSIMISS or NOMISS keywords in the SQL join syntax
- The SAS/ACCESS DIRECT\_SQL= LIBNAME statement option
- Using SAS/ACCESS LIBNAME statement options that indicate member level controls are desired

### Heterogeneous Queries

Implicit pass-through will automatically be disqualified for queries that involve different SAS/ACCESS engines. Implicit pass-through will also not be attempted for queries that involve a single engine with multiple librefs that do not share connection properties, such as using LIBNAME statements that reference different database instances.

### Queries that Incorporate Explicit Pass-Through Statements

PROC SQL will allow joining SAS table references with explicit pass-through queries. For example, in the following query an explicit pass-through query is used as a table expression for an inner join:

```
libname mylib db2 db=mysample schema=ACNTS;

PROC SQL;

connect to Db2 (db=mysample);

select t1.TRANS_AMT,
       t1.ORDER_ID,
       t1.MNG_REP_N,
       t0.SALES_ORIG

from mylib.ORDERS inner join

      (SELECT * FROM CONNECTION TO DB2 (
        select SALESREP_KEY, SALES_ORIG
        from SALES_KEY_TABLE
        where SALES_ORIG = 'CHICAGO')
```

```

) t0
  on t0.SALESREP_KEY = t1.MNG_REP_KEY
;
quit;

```

Implicit pass-through cannot integrate the explicit pass-through query with SAS SQL internal query tree representation, so the join is performed in SAS.

### Queries that Use SAS Data Set Options

Specification of SAS data set options on a Table in a query will disable the implicit pass-through attempt. There are architectural reasons for this restriction. SAS data set options cannot be honored in a pass-through context. There is also no capability in pass-through to combine multiple data set options.

### Queries that Contain the SAS OUTER UNION Operator

The OUTER UNION set operator is a non-ANSI SAS SQL extension. It can increase the size of the result set beyond the input data. PROC SQL will retrieve the input data and perform the Outer Union processing.

### Specification of a SAS Language Function that is not Mapped to a Database Equivalent by the Engine

SAS function specification will inhibit implicit pass-through processing. However, if the SQL contains a triggering construct, any SAS function references in the query will be checked for mapping to a database equivalent by the engine. If the SAS function is not mapped to a database function equivalent implicit pass-through textualization for the statement is terminated.

### Queries That Require REMERGING in PROC SQL

Remerging requires that PROC SQL must make more than one pass-through the data. The following query shows an example of remerging:

```
SELECT AGE, (AGE - AVG(AGE)) from DBMS.TABLE1;
```

One pass might be needed to accumulate summary statistics and a second pass to integrate the aggregate with the original data in the results. PROC SQL extracts data into SAS to perform the remerge processing.

### Specification of ANSIMISS or NOMISS Keywords in the Join Syntax

Use of the ANSIMISS and NOMISS keywords involve SAS specific that would not be implemented on a database system. Specification of these keywords in an SQL join statement disqualifies it from implicit pass-through processing.

### Specification of the DIRECT\_SQL= LIBNAME Option

SQL implicit pass-through will be attempted for qualifying queries, by default. The SAS/ACCESS DIRECT\_SQL= LIBNAME statement option can be specified to disable implicit pass-through processing if desired. See the SAS/ACCESS product documentation for a description of the DIRECT\_SQL= option.

## Identifying Other Inhibitors to Implicit Pass-Through

---

Implicit pass-through processing might terminate for reasons other than the incomplete list of constructs listed in the [“Inhibitors to implicit pass-through”](#) section of this chapter. For example, a database prepare error might be returned in the SASTRACE SQL trace output.

It is also possible that the implicit pass-through textualizer might detect some condition in PROC SQL's internal tree representation of the query that it realizes cannot be accommodated. When this happens, there is no attempt to pass the syntax to the database. The SQL trace output will not show details for these situations.

Because the implicit pass-through process was designed as a "silent" optimization technique, when the process is interrupted it can be difficult to determine the reasons. One tool that can be useful is the SAS system option `DEBUG=DBMS_SELECT`.

Using this option will show the SQL submitted to a database, as with the `SASTRACE=` option. Additionally, specification of option `debug=DBMS-SELECT` will also show the SQL generation of the implicit pass-through performed up to the point of failure. This will usually be an incomplete SQL generation. This SQL will not show the problem condition directly, but will show the last successful part of the query that was generated.

Consider the following simple query:

```
libname mylib db2 db=mysample schema=SAS_ONE;

options debug=dbms_select ;

PROC SQL;
SELECT T1.EMPID, T1.HIREDATE, T2.DEPTCODE, T2.DNAME
  FROM mylib.employees as T1
  INNER JOIN mylib.deptnames(schema=SAS_TWO) as T2
    ON T1.DEPT = T2.DEPTCODE
quit;
```

This query contains an inner join, which is a construct that causes SQL implicit pass-through. However, this query also contains the SAS data set option `SCHEMA=` specified on one of the table references. As mentioned, the specification of SAS data set options will inhibit the implicit pass-through process.

When the above PROC SQL query is submitted the output from the `DEBUG=DBMS_SELECT` option shows the following:

```
12
13      libname mylib db2 db=mysample schema=SAS_ONE;
NOTE: Libref MYLIB was successfully assigned as follows:
      Engine:          DB2
      Physical Name:  mysample
14
15      options debug=dbms_select ;
16
17      PROC SQL;
18      SELECT T1.EMPID, T1.HIREDATE, T2.DEPTCODE, T2.DNAME
19      FROM mylib.employees as T1
20      INNER JOIN mylib.deptnames(schema=SAS_TWO) as T2
```

```

21          ON T1.DEPT = T2.DEPTCODE  ;

DBMS_SELECT: SELECT * FROM SAS_ONE.EMPLOYEES FOR READ ONLY

DBMS_SELECT: SELECT * FROM SAS_TWO.DEPTNAMES FOR READ ONLY

DEBUG: DBMS engine returned an error - NO Implicit Passthru.
DEBUG: Error during prepare of:
DEBUG:  select T1."EMPID", T1."HIREDATE", T2."DEPTCODE", T2."DNAME" from
SAS_ONE.EMPLOYEES T1,

DBMS_SELECT: SELECT  "DEPTCODE", "DNAME"  FROM SAS_TWO.DEPTNAMES  FOR READ ONLY

DBMS_SELECT: SELECT  "DEPT", "EMPID", "HIREDATE"  FROM SAS_ONE.EMPLOYEES  FOR READ
ONLY

22          quit;
NOTE: The PROCEDURE SQL printed page 1.
NOTE: PROCEDURE SQL used (Total process time):
      real time           6.34 seconds
      cpu time             0.53 seconds

```

The conversion to text for this statement halts when the table reference containing the data set option is encountered during the query tree traversal. The implicit pass-through processing for the statement halts when a data set option is encountered. Afterward, processing of the query continues with PROC SQL using the SAS/ACCESS engine API directly. The queries used to extract the database data into SAS are also shown in the DEBUG=DBMS\_SELECT output following the incomplete implicit pass-through conversion to text.

### Using SAS/ACCESS LIBNAME Options That Indicate Member-Level Controls are wanted

The specification of SAS/ACCESS LIBNAME options that request table level controls, such as table locking (READ\_LOCK\_TYPE= - see <http://support.sas.com/91doc/getDoc/acreldb.hlp/a001342364.htm> or UPDATE\_LOCK\_TYPE= - see <http://support.sas.com/91doc/getDoc/acreldb.hlp/a001342392.htm>) can inhibit the ability to pass the SQL statement to the database for direct processing for some SAS/ACCESS products. Often these options cannot be honored using implicit pass-through. If one of the above LIBNAME options is specified, certain SAS/ACCESS products (like DB2 and Oracle) will internally generate SQL LOCK TABLE statements to acquire the requested locks. The SAS/ACCESS engine is also needed to properly manage these locks. Processing the statement in SAS using the SAS/ACCESS engine might be required in order to honor the specified option setting. The default settings for these options should not inhibit the implicit pass-through optimization.

### Implicit Pass-Through Back-down Behavior

---

The implicit pass-through attempt for an SQL query might not succeed. A construct might be encountered that causes the process to terminate, or the database might return syntax when a textualized is prepared.

In the event of a termination, PROC SQL will reexamine the query to determine whether a portion of the query could be passed down. This process will continue until there are no longer any portions of the query remaining that can be passed down. At that point, the SAS/ACCESS engine can process the query.

Consider the following SAS SQL Query as illustration:

```

26          libname dbms oracle user=scott pw=XXXXX path=alien;
NOTE: Libref DBMS was successfully assigned as follows:
      Engine:          ORACLE
      Physical Name:  alien
27
28          option sastrace=",,,d" sastraceloc=saslog no$stsuffix;
29
30          PROC SQL;
31          select * from dbms.LEFT
32          intersect all
33          select distinct * from dbms.RIGHT;

ORACLE_1: Prepared:
SELECT * FROM LEFT

ORACLE_2: Prepared:
SELECT * FROM RIGHT

ORACLE_3: Prepared
select LEFT."X", LEFT."Y" from LEFT intersect all select distinct RIGHT."X", RIGHT."Z"
from RIGHT

ERROR: ORACLE prepare error: ORA-00928: missing SELECT keyword. SQL statement:  select
LEFT."X", LEFT."Y" from LEFT intersect all select distinct RIGHT."X", RIGHT."Z" from
RIGHT.
ACCESS ENGINE:  SQL statement was not passed to the DBMS, SAS will do the processing.

ORACLE_4: Prepared:
select distinct RIGHT."X", RIGHT."Z" from RIGHT

ORACLE_5: Executed:
SELECT statement  ORACLE_1

ORACLE_6: Executed:
SELECT statement  ORACLE_4

ACCESS ENGINE:  SQL statement was passed to the DBMS for fetching data.
34
35          quit;
NOTE: The PROCEDURE SQL printed page 1.
NOTE: PROCEDURE SQL used (Total process time):
      real time          10.18 seconds
      cpu time           0.92 seconds

```

When this query is executed against Oracle data, the initial attempt to pass the query through to Oracle fails with a prepare error. The prepare error occurs because the INTERSECT ALL operator is not supported in the database's SQL syntax.

After the initial implicit pass-through attempt has failed, the query is again examined for additional components that could pass to the database. The second query contributing to the INTERSECT ALL operation contains the DISTINCT keyword:

```
Select distinct * from DBMS.RIGHT
```

Implicit pass-through is attempted for that component of the query. Here the results of the INTERSECT ALL operation must be performed in SAS. We see in the SASTRACE= output that the query has been partially optimized by the implicit pass-through mechanism.

## Guidelines for Writing SQL for Implicit Pass-Through

---

Most SQL implementations vary in terms of the level of compliance with the ANSI SQL standards and the SQL language extensions that they provide. The SQL language is understood by all relational databases, however, variations of implementation can result in writing SQL that is not supported across databases.

SAS SQL supports many of the SAS language elements that are commonly used throughout the SAS system as SQL syntax extensions. When writing a query in SAS SQL, you can increase the likelihood of successful implicit pass-through by observing the following general recommendations.

Avoid the use of SQL extensions that are specific to SAS. Use simple ANSI SQL syntax in generated SAS SQL queries.

Know the SQL syntax limitations or requirements of your target database. The SAS SQL queries that you write should take these into account.

It is important to use `SASTRACE=` SQL tracing to monitor the success of implicit pass-through and to help refine your queries.

### Avoid SAS Specific SQL Language Extensions

The single most important consideration when writing SAS SQL queries for implicit pass-through is to avoid the use of SAS SQL syntax extensions. Use of SAS SQL syntax extensions often results in an inability to pass a query to a database. In general, when a SAS SQL language extension is used in a query, you should expect processing to occur in SAS.

Avoid the use of SAS functions in queries when the SAS functions are not mapped to an equivalent database function. The use of an unsupported SAS function will automatically disqualify an SQL query from implicit pass-through processing.

When used, many of these unsupported SAS functions will not automatically disqualify a query from implicit pass-through, but could be rejected later by the database.



These SAS SQL language extensions could consist of:

- SAS specific operators
- SAS specific expressions
- Special SAS column modifiers
- Variations in SQL implementations

For more information:

SAS Institute Inc., 2006, SAS online documentation, "The SQL Procedure PROC SQL and the ANSI Standard" Available (<http://support.sas.com/onlinedoc/913/getDoc/en/proc.hlp/a002473705.htm>).

### **SAS Specific SQL Operators**

Occasionally, implicit pass-through can provide direct replacement for an SQL operator when there is a difference between SAS SQL and the SQL that is supported by a database. This is not always the case.

For example, a common SAS language filtering operator, which is primarily used to determine if a SAS variable contains a specific character string, is the CONTAINS operator. The CONTAINS operator is supported throughout the SAS programming language and is supported by PROC SQL. The SQL LIKE operator provides the same functionality. In the following query that uses the CONTAINS operator, implicit pass-through attempts to pass the query to the database. The database rejects the SAS specific syntax.

```

67      options sastrace=",,,d" ;
68
69      proc sql;
70          title Attempt to filter using the SAS CONTAINS operator;
71          select distinct component_id, lead, complete_code
72              from dbms.project
73              where Complete_code contains 'Pending';
DB2: AUTOCOMMIT turned ON for connection id 0

DB2_1: Prepared: on connection 0
SELECT * FROM PROJECT FOR READ ONLY

DB2_2: Prepared: on connection 0
select distinct project."COMPONENT_ID", project."LEAD",
project."COMPLETE_CODE" from PROJECT where PROJECT."COMPLETE_CODE" contains
'Pending' FOR READ ONLY

DB2: ROLLBACK performed on connection 0.
ERROR: CLI describe error: [IBM][CLI Driver][DB2/NT] SQL0104N  An unexpected
        token "contains" was found following "JECT."COMPLETE_CODE"".  Expected
        tokens may include: "IN".  SQLSTATE=42601
ACCESS ENGINE:  SQL statement was not passed to the DBMS, SAS will do the
processing.

DB2_3: Prepared: on connection 0
SELECT  "COMPONENT_ID", "LEAD", "COMPLETE_CODE"  FROM PROJECT  FOR READ ONLY

```

```
DB2_4: Executed: on connection 0
Prepared statement DB2_3
```

```
74          quit;
NOTE: The PROCEDURE SQL printed page 1.
NOTE: PROCEDURE SQL used (Total process time):
      real time          1.40 seconds
      cpu time           0.56 seconds
```

If the preceding query is rewritten and you use the SQL LIKE operator, the syntax is accepted by the database.

In this case, the comparison is made more efficient because it can be indicated that the search string must occur at the beginning of the variable. Using the SQL LIKE operator enables you to indicate that the search string must occur at the beginning of the data. The SAS CONTAINS operator does permit this indication, and in this example could have returned unwanted rows. When you have to choose between using a SAS specific operator or a standard SQL operator in an SQL query, it is recommended that you use the latter.

```
76          proc sql;
77             title Attempt to filter using the SQL LIKE operator;
78             select distinct component_id, lead, complete_code
79                from dbms.project
80                where Complete_code like 'Pending%';
```

```
DB2_5: Prepared: on connection 0
SELECT * FROM PROJECT FOR READ ONLY
```

```
DB2_6: Prepared: on connection 0
  select distinct project."COMPONENT_ID", project."LEAD",
project."COMPLETE_CODE" from PROJECT where PROJECT."COMPLETE_CODE" like
'Pending%' FOR READ ONLY
```

```
DB2_7: Executed: on connection 0
Prepared statement DB2_6
```

```
ACCESS ENGINE:  SQL statement was passed to the DBMS for fetching data.
81          quit;
NOTE: The PROCEDURE SQL printed page 2.
NOTE: PROCEDURE SQL used (Total process time):
      real time          0.09 seconds
      cpu time           0.01 seconds
```

### **SAS Specific SQL Expressions**

PROC SQL allows the combination of comparison, Boolean, and algebraic expressions. These expressions are SAS SQL extensions to the ANSI SQL standards. They can be used in filters and as column expressions in the SELECT clause of an SQL query.

Consider the following query:

```
SELECT distinct table0.CLIENTNUM as CLIENT,
      (table0.LIFESTAGE > 6) AS QUALIFIED
FROM orafin.CLIENTS table1;
```

The preceding SAS SQL query generates the Boolean column QUALIFIED based on the value of the column LIFESTAGE. If the value of the column LIFESTAGE is greater than six, the resulting variable QUALIFIED is true. In SAS, true is defined to be 1. If the value of LIFESTAGE is not greater than six, then the value of the variable QUALIFIED is 0 (false).

This type of expression is not supported in database SQL implementations. Using this type of SAS specific expression would cause implicit pass-through to fail. However, this query can easily be rewritten using standard SQL syntax that allows implicit pass-through to succeed. The SAS expression is equivalent to the following standard SQL case expression syntax.

```
SELECT distinct table0.CLIENTNUM as CLIENT,
       case when table0.LIFESTAGE > 6
            then 1 else 0
       end as QUALIFIED
FROM orafin.CLIENTS table1;
```

Implicit pass-through does not re-interpret this syntax or alter the logic of these expressions. It cannot modify the internal query parse tree form because it must be preserved in the event that PROC SQL must process the query. Notice that, when the above queries are processed through implicit pass-through, the expressions that are produced in the generated SQL appear about the same as they were written. The SAS specific syntax is not valid for Oracle, and a prepare error is returned by the database. As a result, this query must be processed in SAS.

```
55 Proc SQL;
56 Create Table work.tempTable1 as
57 SELECT distinct table0.CLIENTNUM as CLIENT,
58 table0.LIFESTAGE > 6 as QUALIFIED
59 FROM orafin.CLIENTS table0;
```

```
ORACLE_8: Prepared: on connection 0
SELECT * FROM CLIENTS
```

```
ORACLE_9: Prepared: on connection 0
select distinct table0."CLIENTNUM" as CLIENT, table0."LIFESTAGE" > 6 as QUALIFIED
from CLIENTS table0
```

```
ERROR: ORACLE prepare error: ORA-00923: FROM keyword not found where expected.
```

```
ACCESS ENGINE: SQL statement was not passed to the DBMS, SAS will do the processing.
```

```
ORACLE_10: Prepared: on connection 0
SELECT "CLIENTNUM", "LIFESTAGE" FROM CLIENTS
```

```
ORACLE_11: Executed: on connection 0
SELECT statement ORACLE_10
```

In the following example, the standard query form that uses the standard SQL Case expression is accepted by the database. Implicit pass-through succeeds, and the database processes the query.

```
62
63 Proc SQL;
64 Create Table work.tempTable2 as
65 SELECT distinct table0.CLIENTNUM as CLIENTt,
66 case when table0.LIFESTAGE > 6 then 1
```

```

67             else 0
68             end as QUALIFIED
69     FROM orafin.CLIENTS table0;

```

```

ORACLE_12: Prepared: on connection 0
SELECT * FROM CLIENTS

```

```

ORACLE_13: Prepared: on connection 0
  select distinct table0."CLIENTNUM" as CLIENT, case when table0."LIFESTAGE" > 6 then
1 else 0 end as QUALIFIED from CLIENTS table0

```

```

ORACLE_14: Executed: on connection 0
SELECT statement ORACLE_13

```

ACCESS ENGINE: SQL statement was passed to the DBMS for fetching data.

### **SAS SQL Column Modifiers Ignored by Implicit Pass-Through**

PROC SQL supports the SAS INFORMAT=, FORMAT=, LABEL=, and LENGTH column modifiers in the SELECT clause of a query. These column modifiers are used to control the display format of output data. Implicit Pass-Through cannot support the use of these column modifiers, but using them will not cause Implicit Pass-Through to reject the query. These SAS language elements are ignored by Implicit Pass-Through.

In the following example, the entire SQL statement is not passed down through implicit pass-through; only the query is passed. The query that is submitted to DB2 does not reference the column modifiers. PROC SQL facilitates the use of the column modifiers in the query by associating them with the creation of the DB2 table SERVICE. This can be accomplished because the DB2 table is created outside of implicit pass-through by using the SAS/ACCESS engine.

```

60     options sastrace=",,d" ;
61
62     proc sql;
63
64     create table dbms.service as
65     select distinct empnum label='Employee ID' length=3,
66             empname label='Name' length=25,
67             emptitle length=15 label='Title',
68             hiredate format= date7. length=4
69     from dbms.employee
70     where Status= 'FULL'
71     order by empnum;

```

```

DB2_1: Prepared: on connection 0
SELECT * FROM EMPLOYEE FOR READ ONLY

```

```

DB2_2: Prepared: on connection 0

select distinct employee."EMPNUM" as EMPNUM,
             employee."EMPNAME" as EMPNAME,
             employee."EMPTITLE" as EMPTITLE,
             employee."HIREDATE" as HIREDATE
from EMPLOYEE
where EMPLOYEE."STATUS" = 'FULL'
order by EMPNUM asc FOR READ ONLY

```

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

DB2\_4: Executed: on connection 0  
Prepared statement DB2\_2

ACCESS ENGINE: SQL statement was passed to the DBMS for fetching data.

DB2\_5: Executed: on connection 2

```
CREATE TABLE SERVICE (EMPNUM FLOAT,EMPNAME VARCHAR(25),EMPTITLE VARCHAR(15),HIREDATE
DATE)
```

DB2: COMMIT performed on connection 2.

DB2\_6: Prepared: on connection 2

```
INSERT INTO SERVICE (EMPNUM,EMPNAME,EMPTITLE,HIREDATE) VALUES ( ? , ? , ? , ? )
```

DB2\_7: Executed: on connection 2

Prepared statement DB2\_6

DB2: COMMIT performed on connection 2.

The use of SAS column modifiers is not recommended as a general practice for implicit pass-through. SAS program code with dependencies on these types of changes could behave unpredictably, and this practice could disqualify an SQL statement from participating in future enhancements to Implicit Pass-Through. The preceding example succeeds because the column modifiers can be applied to the table creation by using a separate step. SAS column modifiers could not be supported by implicit pass-through enhancements that attempt to pass the complete SQL statement to the database.

### Database-Specific Syntax Limitations or Requirements

Every database provides slight variations in their SQL implementations. There is also variation in the restrictions that they apply to the syntax. Implicit Pass-Through cannot successfully pass a FULL OUTER join to an underlying database if the database does not support FULL joins. If a database imposes a limit on the number of items that can be specified in an IN clause, then any SAS SQL query that exceeds that limit will not pass to the database.

PROC SQL can be quite forgiving of some SQL syntaxes that do not fully conform to standards. This can be advantageous when executing SQL in SAS, but can lead to complications for passing a query to a database. It is important to be aware of the SQL syntax limitations or requirements of your target databases.

One example of this potential complication can be seen in the support for in-line view table expressions. An *in-line view* is a query-expression that is nested in the FROM clause of a query. In-line views can be useful for encapsulating part of a complex query. When an in-line view is used in a query, most databases require that a corresponding alias for the table expression be provided in the query. In the following query, the in-line view 'b' is highlighted.

```

10      proc sql ;
11      create table work.results as
12          select a.com_r, a.sexe, a.age, b.region
13          from test.result1 as a
14          left join (select region,commune
15                    from test.geol
16                    where region in ('01', '02','03') ) as b
17          on a.com_r = b.commune;

```

```

DB2_5: Prepared:
SELECT * FROM RESULT1 FOR READ ONLY

```

```

DB2_6: Prepared:
SELECT * FROM GEO1 FOR READ ONLY

```

```

DB2_7: Prepared:
select a."COM_R", a."SEXE", a."AGE", b."REGION"
from RESULT1 a
left join ( select geol."REGION", geol."COMMUNE"
            from GEO1
            where ( GEO1."REGION" in ('01', '02', '03') ) ) b
on a."COM_R" = b."COMMUNE" FOR READ ONLY

```

```

DB2_8: Executed:
Prepared statement DB2_7

```

**ACCESS ENGINE: SQL statement was passed to the DBMS for fetching data.**  
NOTE: Table WORK.RESULTS created, with 1000 rows and 4 columns.

PROC SQL does not require aliasing of in-line view table expressions. If the previous query is rewritten without using table aliases, then PROC SQL accepts the syntax. Databases object to this type of query. Implicit Pass-Through fails the query when it determines that the in-line view did not have an alias. It is a good practice to provide table expression aliases when writing queries that contain in-line views.

```

21      proc sql ;
22      create table work.results as
23          select result1.com_r, result1.sexe, result1.age, geol.region
24          from test.result1
25          left join (select region, commune
26                    from test.geol
27                    where region in ('01', '02','03') )
28          on result1.com_r = geol.commune;

```

```

DB2_9: Prepared:
SELECT * FROM RESULT1 FOR READ ONLY

```

```

DB2_10: Prepared:
SELECT * FROM GEO1 FOR READ ONLY

```

**SAS\_SQL: A correlation name (alias) expected for an inline view.**  
**SAS\_SQL: Unable to convert the query to a DBMS-specific SQL statement due to an error.**

ACCESS ENGINE: SQL statement was not passed to the DBMS, SAS will do the processing.

```

DB2_11: Prepared:
SELECT "COM_R", "SEXE", "AGE" FROM RESULT1 FOR READ ONLY

```

```

DB2_12: Executed:
Prepared statement DB2_11

```

```
DB2_13: Prepared:
SELECT  "REGION", "COMMUNE" FROM GEO1 WHERE ( ( "REGION" IN ( '01' , '02' , '03' )
) ) FOR READ ONLY
```

```
DB2_14: Executed:
Prepared statement DB2_13
```

NOTE: Table WORK.RESULTS created, with 1000 rows and 4 columns.

Many SAS SQL internal query forms are presented to Implicit Pass-Through as if they were originally coded as in-line views. It is worthwhile to consider coding in-line views directly in a query. This allows the required aliases to be provided in the query. Use SASTRACE= SQL tracing to refine your queries. If you notice that in-line views appear in the query that is submitted to the database, consider modifying the query that you coded, accordingly.

### Casual Use of External SQL Views

If a SAS SQL query contains a reference to a SAS SQL view, the query form that defines the view is read and reconstituted in memory into the SQL parse tree representation of the query. Generally, this processing involves the logical merging of queries, predicates, and so on, from the SAS SQL view definition with the SQL parse tree query that references it.

SAS SQL queries can nest SAS SQL view references within other SAS SQL views. The resulting output from the multiple integration steps is a further modified SQL parse tree, a tree that ultimately can be used for processing the request.

Often, the final query forms that emerge from this process are complex. Because SAS SQL supports a more relaxed SQL syntax than many databases do, the complex forms can present a problem—the SAS SQL syntax cannot be expressed in an SQL syntax supported by the database. This means that, although SAS SQL views can reduce the complexity and improve the readability of the query, the resulting combined queries can inhibit the ability of implicit pass-through to pass a final query that is acceptable to the database.

Using SAS SQL views allows implicit pass-through to receive SAS SQL internal query forms that do not conform to valid ANSI SQL syntaxes. As an example consider the following simple join of a table to itself. Table aliases are not provided in the query to allow the column references to be uniquely identified. PROC SQL issues an error. This query is not valid due to the ambiguous column references.

```
15      proc sql;
16      Select count(*)
17      from dbms.SALESREP1, dbms.SALESREP1
18      WHERE EMPID = MANAGER ;
```

```
ERROR: Ambiguous reference, column EMPID is in more than one table.
ERROR: Ambiguous reference, column MANAGER is in more than one table.
NOTE: PROC SQL set option NOEXEC and will continue to check the syntax of statements.
19      quit;
NOTE: The SAS System stopped processing this step because of errors.
```

However, if two SAS SQL views are created that describe a query of this table, the SAS SQL views can be joined. When the column names in the join conditional are qualified with the view names, the references are not considered ambiguous. When the SAS SQL views are integrated into the query, the

resulting form still contains ambiguous column references and is rejected by the database.

```

21      proc sql;
22      create view V1 as select * from dbms.SALESREP1;
NOTE: SQL view WORK.V1 has been defined.
23      create view V2 as select * from dbms.SALESREP1;
NOTE: SQL view WORK.V2 has been defined.
24      quit;
25
26      proc sql;
27      Select count(*)
28      from v1, v2
29      WHERE v1.EMPID = v2.MANAGER ;

```

```

ORACLE_3: Prepared:
SELECT * FROM SALESREP1

```

```

ORACLE_4: Prepared:
SELECT * FROM SALESREP1

```

```

ORACLE_5: Prepared:
select COUNT(*) from SALESREP1, SALESREP1 where SALESREP1."EMPID" =
SALESREP1."MANAGER"

```

```

ERROR: ORACLE prepare error: ORA-00918: column ambiguously defined.
ACCESS ENGINE: SQL statement was not passed to the DBMS, SAS will do the processing.

```

```

ORACLE_6: Executed:
SELECT statement ORACLE_4

```

```

ORACLE_7: Executed:
SELECT statement ORACLE_3
30      quit;

```

If the above join of the two SAS SQL views is modified to include table aliases, implicit pass-through will see a join of two aliased in-line view table expressions. This modification allows the query to be passed successfully to Oracle.

```

31
32      proc sql;
33      Select count(*)
34      from v1 EMP, v2 MGR
35      WHERE EMP.EMPID = MGR.MANAGER ;

```

```

ORACLE_8: Prepared: on connection 0
SELECT * FROM SALESREP1

```

```

ORACLE_9: Prepared: on connection 0
SELECT * FROM SALESREP1

```

```

ORACLE_10: Prepared: on connection 0
select COUNT(*) from ( select SALESREP1."EMPID" from SALESREP1 ) EMP, ( select
SALESREP1."MANAGER" from SALESREP1 ) MGR where EMP."EMPID" = MGR."MANAGER"

```

```

ORACLE_11: Executed: on connection 0
SELECT statement ORACLE_10

```

```

ACCESS ENGINE: SQL statement was passed to the DBMS for fetching data.

```



---

## Explicit Pass-Through

---

Use the SAS SQL pass-through facility (commonly referred to as 'explicit pass-through') to issue SQL queries or executable SQL statements directly to a database from PROC SQL. No SQL generation is performed internally in SAS for explicit pass-through; the user supplies the exact SQL syntax passed to the database. The supplied SQL is not modified by SAS in any way.

Explicit pass-through can:

permit access to certain database operations that are not supported by the SAS engine API, such as creating database indexes or views

be useful for situations where the internally generated SQL is not optimal or when a particular query is performance-critical

Explicit pass-through is, generally, the most efficient mechanism available in SAS for issuing complex queries to, or manipulating database data. This section explains how you can use the explicit pass-through facility (referred from this point forward as explicit pass-through) to submit database-specific SQL directly to a database, discusses the best use of the facility, and presents example processing situations.

### Preliminary Considerations

---

The following are some factors to consider when using explicit pass-through.

#### Managing connection activity

Depending on the relational database, making multiple connections can be costly in terms of execution time. Explicit pass-through maintains a database connection until you issue a disconnect statement or the SQL procedure terminates. In contrast, the LIBNAME connection maintains a database connection until you issue a CLEAR for the LIBNAME statement or terminate the SAS session.

Explicit pass-through has the potential for increased connection activity over the LIBNAME connection because each SQL procedure step that contains explicit SQL statements issues a connect statement and a disconnect statement to the database. As a consequence, explicit pass-through connections can be more frequent, and held for shorter durations, than the LIBNAME connections. For more information, see the **Error! Reference source not found.** section of the Advanced Topics chapter of this document.

To minimize the overhead of frequent database connections combine your explicit pass-through work into fewer PROC SQL steps.

#### Issuing database COMMIT and ROLLBACK SQL statements

A SAS/ACCESS engine submits COMMIT and ROLLBACK SQL statements to the database on your behalf. When you use explicit pass-through, you must issue a ROLLBACK statement as needed. Depending on the SAS/ACCESS product, you might have to issue a COMMIT statement as well. Thus,

you control whether your table data is written or not written to the database.

## Anticipating porting your code for reuse with a different database

Explicit pass-through code usually requires more extensive modifications than SAS/ACCESS LIBNAME code to reuse with another supported database. (LIBNAME code can require as minor a change as the LIBNAME definition.) If you want your explicit pass-through code to be portable, specify SQL statements that are compatible with the ANSI standard supported by your target relational database. Additionally, do not use SAS SQL-specific or database-specific SQL extensions.

## When to Use Explicit Pass-Through

---

### For Database Operations That a SAS/ACCESS Engine Does Not Support

Explicit pass-through supports virtually all SQL syntax that the database accepts, including database-specific SQL extensions. Consequently, after you connect to the database, you can submit most or all database-specific SQL supported by vendor tools such as Oracle SQLPlus, DB2 CLP, and Teradata BTEQ. Using database-specific SQL, you can create tables, views, indexes, and integrity constraints; collect statistics; perform database-specific functions; and even call database stored procedures.

PROC SQL supports CREATE INDEX only for native SAS tables. To create an index of a database table, you must use explicit pass-through. The following explicit pass-through example creates an index in DB2:

```
PROC SQL;
connect to db2 (USER =db2 PASSWORD=db2_data4u2c datasrc=sample in=USERSPACE1);
execute ( create unique index xxx on z (i) ) by db2;
quit;
```

Statistics of a database table cannot be collected from within SAS, you must use explicit pass-through. The following example shows how to use explicit SQL pass-through to collect statistics for Oracle:

```
PROC SQL;
connect to oracle (USER=user PASSWORD=password1 PATH=path1);
execute ( ANALYZE TABLE DEPT_FINANCIALS COMPUTE STATISTICS
        FOR TABLE
        FOR ALL INDEXES
        FOR ALL INDEXED COLUMNS) by oracle;
quit;
```

Database stored procedures can provide secure and efficient execution of database operations. The following example shows how to use explicit pass-through to execute a stored procedure in Teradata.

```
PROC SQL;
connect to teradata(user=user pass=password1 server=server1 mode=teradata);
execute (CALL ACCOUNTING.SET_SALARY_GRADE('SENIOR ANALYST', 79000)) by teradata;
quit;
```



**Note:** While the preceding example shows Teradata, if the relational database supports stored procedures then the corresponding SAS/ACCESS interface can call the stored procedure through explicit pass-through.

## For Database-Specific SQL Syntax Extensions

You might choose to specify explicit pass-through to leverage the database functionality. The following example subsets a DB2 table using DB2 functions in the WHERE clause:

```
PROC SQL;
connect to db2 (USER=db2 PASSWORD=db2_password datasrc=sample in=USERSPACE1);
create table saslocal as
select * from connection to db2(
SELECT product,price_variation from pricing_summary
where (days (current_date) - days(last_product_analysis_date) <= 5)
);
quit;
```

(The SQL that is passed to the relational database is in bold in the above example.)

Only the rows filtered by the WHERE clause are transmitted to SAS, instead of all the rows of all the tables involved.

Another use of explicit pass-through can be to turn off or on database logging. Database logging for some databases such as DB2 can have adverse effects, such as diminishing performance by adding the processing overhead of logging. Additionally, performing large table operations when logging can grow the log size to fill the disk partition can cause system problems.

In the following example, we turn off logging when creating a new DB2 table; we then can efficiently populate the table using another DB2 table. Our example shows a two-step process: first, creating an empty table; and, second populating the table.

```
PROC SQL;
connect to db2 (USER=db_user PASSWORD=db2_password datasrc=sample in=USERSPACE1);
/*
 * Create the empty DB2 table, with logging turned off.
 */
execute(
create table I_PRIMARY_CPTY_CR_MITIGANT as (
select COUNTERPARTY_RK,
CREDIT_RISK_MITIGANT_RK
from bbt.I_COUNTERPARTY_X_CR_MITIGANT
where CR_MITIGANT_REL_TYPE_CD = 'PRM' ) with no data not logged initially
) by db2;
/*
 * Populate the table.
 */
execute(
insert into I_PRIMARY_CPTY_CR_MITIGANT
select COUNTERPARTY_RK,
CREDIT_RISK_MITIGANT_RK
from I_COUNTERPARTY_X_CR_MITIGANT
where CR_MITIGANT_REL_TYPE_CD = 'PRM' with ur
) by db2;
execute( commit ) by db2;
quit;
```

(The SQL that is passed to the relational database is in bold in the preceding example.)



**Note:** The preceding example shows a two-step process due to DB2's functional design. Some other databases can create and populate a table in a single step. See your target database's documentation for more information.

The next examples of explicit pass-through involve different situations: the first example uses the Oracle specific "hints" feature, and the second example uses the Oracle specific upsert operation to improve performance of the query. Oracle hints have become less important with newer, optimized Oracle versions, but the capability can assist Oracle performance. SAS can generate a select count(\*) from employees statement but cannot embed hints in the SQL that is generated.

This example copied from the SAS/ACCESS documentation uses explicit pass-through with PRESERVE\_COMMENTS to pass additional information, called hints, to Oracle for processing.

For more information: SAS Institute Inc., 2003, "SAS/ACCESS to Oracle Pass-Through Facility Specifics for Oracle": <http://support.sas.com/91doc/getDoc/acreldb.hp/a001355943.htm>

```
PROC SQL;
connect to oracle as mycon(user=testuser
      password=testpass preserve_comments);
select *
      from connection to mycon
      (select /* +indx(empid) all_rows */
      count(*) from employees);
quit;
```

An upsert operation updates master table rows with transaction table rows that match on a specified key. Where no match occurs, the transaction rows are inserted into the master table. This example taken from Oracle online documentation shows how to perform an upsert operation on the database.

```
PROC SQL;
connect to oracle (user=scott password=tiger path=hulkp1);
execute(
merge into bonuses d
using (select employee_id, salary, department_id from employees
where department_id = 80) s
on (d.employee_id = s.employee_id)
when matched then update set d.bonus = d.bonus + s.salary*.01
when not matched then insert (d.employee_id, d.bonus)
values (s.employee_id, s.salary*0.1)
);
execute (commit) by oracle;
quit;
```



**Note:** The hints feature shown in the first example is Oracle-specific. The upsert function is supported by Oracle, DB2 and Teradata.

### For Database Processing That the Implicit Pass-Through Facility Does Not Translate

The implicit pass-through facility might not produce correct SQL in every situation for every database. If a join or query written in ANSI-standard SQL fails to pass to the database, you can use explicit pass-through instead.

**Recoding an Inner Join Followed by an Outer Join**

The first example presents an unusual database-specific failure: the implicit pass-through facility generates correct non-ANSI standard join syntax for Oracle but does not generate the correct ANSI syntax required by DB2 and Teradata. As a result, the example join succeeds in Oracle, but the join fails in Teradata or DB2. Because the implicit pass-through facility does not generate the standard ANSI SQL expected, Teradata rejects the SQL and only part of the operation is pushed to the database.

The rewrite from SAS SQL to explicit SQL is made simple by a helpful trick: in the SAS SQL we name the libref identically to the Teradata schema, "teraschm". The rewrite to explicit SQL then consists of just wrapping the SAS SQL in explicit SQL syntax. Because librefs are limited to eight characters, this trick only works when the database schema name is eight characters or less.

The partial failure of Implicit Pass-Through to pass the entire join is shown in **bold** type in the SASTRACE information.

```
libname teraschm teradata user=user pass=password1 server=server2 schema=teraschm;
options sastrace=',,,' sastraceloc=saslog no$stsuffix;
PROC SQL;
create table saslocal as
select y,z from teraschm.join1 inner join teraschm.join2 on join1.x=join2.x left join
teraschm.join3 on join1.x=join3.x;
quit;
```

**SASTRACE output**

```
TERADATA_121: Prepared:
select "teraschm"."join1"."y", "teraschm"."join2"."z" from "teraschm"."join1",
"teraschm"."join2" left join "teraschm"."join3" on "teraschm"."join1"."x" =
"teraschm"."join3"."x" where "teraschm"."join1"."x" = "teraschm"."join2"."x"

ERROR: Teradata prepare: Improper column reference in the search condition of a joined table.
ACCESS ENGINE: SQL statement was not passed to the DBMS, SAS will do the processing.

TERADATA_123: Executed:
select "teraschm"."join1"."x", "teraschm"."join1"."y", "teraschm"."join2"."z" from
"teraschm"."join1", "teraschm"."join2" where "teraschm"."join1"."x" =
"teraschm"."join2"."x"

TERADATA_125: Executed:
SELECT "x" FROM teraschm."join3"
```

The following example uses explicit pass-through:

```
PROC SQL;
connect to teradata(user=user pass=password1 server=server2);
create table saslocal as
select * from connection to teradata(
select y,z from teraschm.join1 inner join teraschm.join2 on join1.x=join2.x left join
teraschm.join3 on join1.x=join3.x
);
quit;
```

## SASTRACE output

```
TERADATA_127: Executed:
select y,z from teraschm.join1 inner join teraschm.join2 on join1.x=join2.x left join
teraschm.join3 on join1.x=join3.x
```

### *Using a WHERE Clause on a Large Numeric Database Column*

When SAS generates SQL the 18-digit numeric literal that is specified in the join is not represented precisely. A SAS float data type cannot maintain numeric precision when the numeric literal exceeds 15 digits. This problem of numeric imprecision affects all SAS processing, not just implicit pass-through.

In the following example showing numeric imprecision on a long numeric literal, implicit pass-through fails to return the data specified because an exact database match cannot be found for the value specified (shown in bold in the following example).

```
libname cardacct db2 USER=db2_user PASSWORD=db2_passwd datasrc=sample in=USERSPACE1;
PROC SQL; create table sas as
select test2.account_no, cardholder, expenditures from cardacct.test,cardacct.test2
where (test.account_no = test2.account_no and test2.account_no=925512340233844230);
quit;
```

The SASTRACE output shows the value that SAS generates via implicit pass-through (shown in **bold** type). Because of the numeric imprecision in the SQL value, no table rows are retrieved; the value generated doesn't match exactly the value that is stored in the database.

```
DB2_8: Prepared:
select test2."ACCOUNT_NO", test2."CARDHOLDER", test."EXPENDITURES" from TEST, TEST2
where (TEST."ACCOUNT_NO" = TEST2."ACCOUNT_NO") and (TEST2."ACCOUNT_NO" =
925512340233844224) FOR READ ONLY
```

```
DB2_9: Executed:
Prepared statement DB2_8
```

NOTE: Table WORK.SAS created, with 0 rows and 3 columns.

In the following example, we recode the implicit pass-through code example above to use explicit pass-through so we can successfully deliver the problem row.

```
PROC SQL;
connect to db2(USER= db2_user PASSWORD= db2_passwd datasrc=sample in=USERSPACE1);
create table sas as select * from connection to db2 (
select test2.account_no, cardholder, expenditures from db2.test,db2.test2
where (test.account_no = test2.account_no and test2.account_no=925512340233844230)
);
quit;
```

## SASTRACE output

```
DB2_10: Prepared:
select test2.account_no, cardholder, expenditures from cardacct.test, cardacct.test2
where (test.account_no =test2.account_no and test2.account_no=925512340233844230)
```

```
DB2_11: Executed:
```

Prepared statement DB2\_10

NOTE: Table WORK.SAS created, with 1 rows and 3 columns.

For more information, see the "Difference in Behavior section.

---

## Heterogeneous Joins

---

A join of two SAS data sets using the SAS engine API requires independent open and retrieval of both data sets into SAS. This join of two different data sets in SAS is often referred to as a *Heterogeneous Join*. This section explores alternatives to a standard SAS SQL heterogeneous join when using database data:

- Upload and Join in the database: Copy portions of one or more SAS data sets to the database, and perform the join on the database.
- Key Subsetting Equijoin: Perform the join in SAS but minimize the amount of database data retrieved into SAS by leveraging the join key.

Uploading the SAS table and joining in the database can be particularly effective when the database table is much larger than the SAS table.

Key subsetting equijoin to minimize data transfer might be appropriate when you are confident of certain attributes of the data being joined, such as join key cardinality and database indexes on the join column.

Choosing between an upload and join in the database and a key subsetting equijoin requires experimenting and testing. The key factors might vary from installation to installation so that no one approach is optimal across various environments.

The following is a partial list of key factors that influence the best approach for a heterogeneous join:

- Relative size of SAS tables versus database tables
- If a database join column has an index or is used as the table partitioning key
- Join key cardinality in the SAS and database tables: number of unique join key values and number of matches that occur when joined
- Selectivity of WHERE clauses asserted against database tables

For example; if your SQL contains a highly selective WHERE clause against a database table and this clause is passed to the database, then a small number of rows might be selected from a very large database table. In this situation, standard heterogeneous join might perform adequately because the data transferred to SAS is relatively small.



**Note:** Often you don't know many of the key factors. A general rule of thumb is to Upload and Join on the database when SAS tables are expected to be relatively small and database tables potentially very large – hundreds of millions or billions of rows.

## Uploading and Joining in the Database

---

Upload (copying a SAS table to the database) and joining the tables in the database can increase join performance an order of magnitude or more, depending on the size of the tables. A common example is joining a small SAS table to a very large database table with billions of rows. A standard heterogeneous join might copy a very large amount of the data from the database table into SAS for the join.

In the material that follows, we refer to the SAS table uploaded to the database as an upload table. That upload table might be either a permanent database table or a temporary database table. A permanent database table is stored on disk and the space it consumes deducted from the permanent space quota of the database account. You must delete the temporary table to reclaim the space. A temporary database table might be stored in memory by the database because the database expects a table declared temporary to be transient in nature. A temporary table will always be deleted.

The database administrator might prefer applications to create temporary database tables. Permanent space quotas might be tight, and some accounts might not have permissions to create permanent tables. Before considering Upload and Join you should investigate any issues in how SAS and databases handle NULL and MISSING values differently. Upload tables can help achieve large performance gains in situations including but not limited to:

- Joining a small SAS table to a large database table
- Updating a large database table with a SAS table

This section examines the use of upload tables for joining small SAS tables to large database tables because, in general, this use is seen more frequently.

### Using Temporary and Permanent Tables to Upload and Join in the Database

This discussion and the examples in this section demonstrate both types of upload tables: temporary database tables and permanent database tables. How to use these two types of upload tables is presented in this section.



#### *Using Temporary Database Tables*

Using temporary database table requires two LIBNAME statements to share a single connection to the database:

- One LIBNAME statement contains the option DBMSTEMP=YES that creates and references the temporary table.
- The other LIBNAME statement references the database table.
- Both LIBNAME statements contain the option CONNECTION=GLOBAL, thereby allowing one connection to be shared between two librefs to the database. See “**Error! Reference source not found.**” in the [Advanced Topics](#) section.




To perform a database join with a temporary database table:

1. Establish a shareable connection using the CONNECTION=GLOBAL option to the database.  
 **Note:** Clearing both librefs closes the global connection and drops all database upload tables.
2. Create a database temporary table and load it with the SAS data.
3. (Optional) Run statistics on the newly created table to allow the optimizer to do the join more efficiently.  
 **Note:** Might need to contact the DBA for permissions.
4. Perform a join on the database.
5. Process the join result with SAS.
6. (Optional) Drop the temporary database table

#### ***Using Permanent Database Tables***

A permanent database table requires only one LIBNAME statement with no extra options.

To perform database joins with a permanent database table:

1. Create a permanent table and load it with the SAS data.
2. Run statistics on the new, created table to allow the optimizer to do the join more efficiently.  
 (Optional)  **Note:** Might need to contact the DBA for permissions.
3. Perform a join on the database.
4. Process the join result with SAS.
5. Drop the permanent table.

#### **Factors Influencing Choice of Database Temporary Table or Database Permanent Table for Upload and Join**

As mentioned, a tightly managed database environment can require a database temporary table for the upload table, however, in Teradata a permanent table might perform better. The chart contrasts the uploading to a temporary database table versus uploading to a permanent database table.

Temporary Database Table	Permanent Database Table
Avoids possible restrictions on permanent database table creation	Restrictions creating permanent tables possible in a tightly managed database environment
Does not impact the quota of permanent space quota allotted to the user's database account	Allocates permanent database space
Possible restrictions on defining indexes and collecting statistics	All database join optimizations available
Cannot be bulk loaded	Bulk loading can speed the load of a sizable upload table
Table is automatically dropped when SAS connection is closed (except in some Oracle versions)	Reusable across connections
Requires two LIBNAME statements: one for temporary tables, one for permanent tables	Requires only one LIBNAME reference

### Optimizing Load of the Upload Table

If the SAS table is sizable, loading the upload table into the database can take a large amount of time and affect the performance time of the query. Consider the following recommendations when loading the upload table:

- Upload only needed rows and columns
- Consider bulk loading if the upload table is a database permanent table - temporary database tables cannot be bulk loaded
- If not using bulk load, apply as-needed options that expedite insert, such as DBCOMMIT, INSERTBUFF, or MULTISTMT.

### Upload and Join in the Database with Data Integration Studio

In the release of SAS Data Integration Studio 3.4, both a new SQL join transformation as well as the previous version is available for use. By setting one or more of the properties listed below, the new SQL join transform will automatically create the code to upload one or more sources to the database when generating the explicit pass-through SQL. The table options available for uploading the table are as follows:

**Update Library Before SQL** – a defined database library is selected here that points to where you want to upload the table (this option turns on uploading)

**Enable Case Sensitivity on Upload Table** – turns quoting on the column names and table names to preserve case sensitivity

**Enable Special Characters on Upload Table** – turns quoting on the column names table names to preserve special characters.

**Pre-Upload Action** – if the table exists before upload, do you want to: delete and re-create the table, including all indexes and authorizations, or truncate all records in the table, but preserve any existing indexes and authorizations.

**Use Bulk load for Uploading** – use the BULKLOAD= option to upload the table

**Bulk load Options** – any additional bulk load options

**Additional Upload Options** – any additional upload options to then be added



**Note:** If you want a temporary database table, set the appropriate options in the LIBNAME statement.

### Examples of a Join on a Single Numeric Column

This section presents four versions of a query performing a heterogeneous equijoin on a single numeric column. The initial code performs the join in SAS, and the two following examples perform the join on the database. The fourth query example shows Teradata specific optimizations.

For all databases, the following practices are recommended:

- Indexing join columns
- Keeping fresh statistics might benefit the join of the upload table and database table
- Forcing the upload table join key to the identical type as the database table key

For the example scenario, you are building a data warehouse and your reference tables are in SAS data sets; customer data which will be used in the lookup with the main tables resides in a database table. You want to join a column in the reference table to a column in the customer data and place the join result in another SAS data set. There are 100 values in the lookup table; there are 10,000,000 customers.

The join key CLIENTID is declared DECIMAL(12) in the database table. The table is not optimized for the join key due to the following:

- CLIENTID is not the DB2 or Oracle partition key or the Teradata primary index
- No index is defined on CLIENTID
- No statistics are collected on CLIENTID
- CLIENTID is a nullable column

#### **Performing the Join in SAS**

This first version of the query contains no restricting WHERE clause in the query to the database table. Therefore, all the rows in the database table are copied to SAS and the join is then performed in SAS. The time to transfer the entire table and the resources used by the created temporary table negatively impact the query's performance.

```

Libname sasdata `.`;
Libname dbmsdata db2 datasrc=production ...;
proc sql;
create table sasdata.respout as
select response.RESPCODE, response.RESPDATE,
       client.LASTCONTACTDATE, client.GROSSINCOME,
       client.AGE, client.OCCUPATIONGROUP
from sasdata.response as response, dbmsdata.client_repository as client
where response.CLIENTID = client.CLIENTID
order by response.RESPCODE;
quit;

```

Elapsed time: 2:35.84

### ***Joining in the Database with Temporary Upload Table and Default Join Key Data Type***

This second version of the query transfers the survey responses into a database temporary table, performs the join in the database, and extracts the join result to SAS. Assuming no SAS format is asserted on CLIENTID in the SAS data set SASDATA.RESPONSE, CLIENTID is created FLOAT in the database upload table. The join key type mismatch (FLOAT versus DECIMAL(12)) causes less than optimal performance on DB2.

DB2 librefs are shown. CONNECTION=GLOBAL and DBMSTEMP=YES are bolded for emphasis.

```

Libname sasdata `.`;
Libname dbmsdata db2 connection=global datasrc=production ...;
Libname dbmstemp db2 connection=global dbmstemp=yes datasrc=production ...;

/* Copy the SAS data set to a DBMS upload table */
proc sql;
create table dbmstemp.respjoin as
select response.CLIENTID, response.RESPCODE, response.RESPDATE from sasdata.response
as response;

/* Join on the database, with the result returned to SAS.*/
create table sasdata.respout as
select response.RESPCODE, response.RESPDATE,
       client.LASTCONTACTDATE, client.GROSSINCOME,
       client.AGE, client.OCCUPATIONGROUP
from dbmsdata.respjoin as response, dbmsdata.client_repository as client
where response.CLIENTID = client.CLIENTID
order by response.RESPCODE;

/* Drop the database copy of the SAS table.*/
drop table dbmstemp.respjoin;
quit;

```

Elapsed time, DB2: 1:16.36

### ***Joining in the Database with Permanent Upload Table and Exact Match of Join Key Data Type***

In the third version of the query, we assume we know CLIENTID in the table to be DECIMAL(12). Therefore, we set the format of CLIENTID in the upload table to DECIMAL(12). By matching the formats of the columns, needless data conversion is not performed, and DB2 performance improves. Applying the format of DECIMAL(12) to the upload table key column is bolded for emphasis.

```

Libname sasdata `.`;
Libname dbmsdata db2 connection=global datasrc=production ...;

```

```

Libname dbmstemp db2 connection=global dbmstemp=yes datasrc=production ...;

/*
 * Hardcode the format of CLIENTID to DECIMAL(12).
 */
%let format=12.;

/* Copy the SAS data set to a DBMS upload table */
proc sql;
create table dbmstemp.respjoin
as select response.CLIENTID format &format, response.RESPCODE, response.RESPDATE from
sasdata.response as response;

/* Join on the database, with the result returned to SAS.*/
create table sasdata.respout as
select response.RESPCODE, response.RESPDATE,
       client.LASTCONTACTDATE, client.GROSSINCOME,
       client.AGE, client.OCCUPATIONGROUP
from dbmsdata.respjoin as response, dbmsdata.client_repository as client
where response.CLIENTID = client.CLIENTID
order by response.RESPCODE;

/* Drop the database copy of the SAS table.*/
drop table dbmstemp.respjoin;
quit;

Elapsed time, DB2: 21.54

```

### **Joining in Teradata**

This example query presents Teradata specific optimizations. With an integer join key, the following enhancements optimize upload and join to Teradata. Omission of any one enhancement degrades Teradata performance. Omission of more than one enhancement degrades performance further.

1. Use a permanent Teradata table for the upload table (because statistics cannot be collected on a Teradata temporary table).
2. Choose one join key column and:
  - a) Match its upload table data type to the main table type.
  - b) Declare it as the upload table primary index.
  - c) Collect statistics on it.

Bolding and code comments emphasize the enhancements. Notice also the DROP TABLE statement, which is required to delete a permanent upload table.

```

Libname sasdata `.`;
Libname dbmsdata teradata user= ...;

/*
The dbmsdata LIBNAME statement does not contain the DBMSTEMP=YES option;
therefore dbmsdata.respjoin is created as a permanent Teradata table.
CLIENTID is the chosen join key column (the only choice
in this example). The 12. format matches it to the DECIMAL(12) type
for CLIENTID in main table CLIENT_REPOSITORY. It is asserted as the
upload table primary index.

```

```

Using explicit SQL, we collect statistics on it.
*/

proc sql;
create table dbmsdata.respjoin(dbcreate_table_opts = PRIMARY INDEX(CLIENTID)) as
select response.CLIENTID format 12., response.RESPCODE, response.RESPDATE from
sasdata.response as response;

connect to teradata( user= ... );
execute ( collect statistics on respjoin column CLIENTID ) by teradata;
execute( commit ) by teradata;

/* Join on the database, with the result returned to SAS.*/
create table sasdata.respout as
select response.RESPCODE, response.RESPDATE,
       client.LASTCONTACTDATE, client.GROSSINCOME,
       client.AGE, client.OCCUPATIONGROUP
from dbmsdata.respjoin as response, dbmsdata.client_repository as client
where response.CLIENTID = client.CLIENTID
order by response.RESPCODE;

/* Drop the database copy of the SAS table.*/
drop table dbmsdata.respjoin;
quit;

```

## Using Key Subsetting Equijoins

---

A common type of join is an equijoin, in which values from a column in the first table match to values of a column in the second table. This section focuses on key subsetting equijoins, where only database rows that match SAS join key values are transferred to SAS. This approach applies only to equijoins, and uses SAS/ACCESS options created for this approach. To ensure good performance, key subsetting equijoin requires:

- The join key must be indexed on the database
- Low cardinality of distinct SAS key values

Some knowledge of the data is helpful to ensure good performance and to avoid potential performance pitfalls mentioned later.



**Note:** For a key subsetting equijoin, it is imperative the database join column(s) be indexed, and database statistics on the join column(s) should be kept up-to-date.

When SAS performs a standard heterogeneous join, it must read the data to be joined into SAS, and then process the join internally. Key subsetting equijoins can optimize SAS join processing by retrieving fewer rows into SAS. However, due to overhead involved with key subsetting, better performance is not guaranteed. This section presents three options usable with key subsetting equijoins, and tips for when they might be appropriate.

- MULT\_DATASRC\_OPT=
- DBINDEX=
- DBKEY=

The MULTI\_DATASRC\_OPT=, DBINDEX=, or DBKEY= options can optimize the equijoin of a SAS table when the SAS table contains few unique key values to a large database table indexed on a join key.

These three options interact in the following ways:

- DBINDEX= and DBKEY= are mutually exclusive. If you specify them together, DBKEY= overrides DBINDEX=.
- If you use DBKEY= and MULTI\_DATASRC\_OPT= together, DBKEY= overrides MULTI\_DATASRC\_OPT=.
- If you use DIRECT\_SQL=NONE or NOWHERE, the IN clause cannot be built and passed to the database, regardless of the value of MULTI\_DATASRC\_OPT=.



**Note:** DBKEY=, unless used with a small SAS data set, can degrade performance. To understand why the option is affected by the DBNULLKEYS data set option see:


Levine, Fred "Paper 110-26 Using the SAS/ACCESS Libname Technology to Get Improvements in Performance and Optimizations in SAS/SQL Queries" Proceedings of SUGI 26 2002 Conference. Cary, NC: SAS Institute Inc. Available at <http://www2.sas.com/proceedings/sugi26/p110-26.pdf>.

### Using Database Indexes in Key Subsetting Equijoins

Database indexes are critical for key subsetting equijoins. When indexes are defined on the join keys of large database tables, you obtain a tremendous boost in performance on database data selection operations. The following table shows the effect on SAS/ACCESS processing when the option is used with and without a usable index on the database join key.

Option	Index Defined on Join Key	No Index Defined on Join Key
MULTI_DATASRC_OPT=	Indexed retrieval from the database table on rows that match the join key values.	One or more database full table scans to match the SAS join key values.
DBINDEX=	Indexed retrieval from the database table on rows that match the join key value.	SAS reverts to default behavior and selects all database rows after any WHERE filtering is applied.
DBKEY=	Indexed retrieval from the database table on rows that match the join key value.	Possible database full table scan for each distinct SAS join key value.

Table 2: Effect of defined database indexes on MULTI\_DATASRC\_OPT=, DBKEY=, and DBINDEX options


 **Caution:** As *Table 2* shows, you must define an appropriate database index and collect statistics on the affected equijoin database columns before using any of these options. An appropriate index might need to be unique and might require that the key column be the only column in the index. Without a database index MULTI\_DATASRC\_OPT= is the only option that *might* provide reasonable performance. If a database index cannot be defined, then you should consider a different join approach.

MULTI\_DATASRC\_OPT= works only when the WHERE clause contains a single equijoin condition. You can specify DBKEY= in WHERE clauses with multiple join conditions, as long as DBKEY= is specified only for database columns that participate in the equijoin condition. DBINDEX is also capable of handling multiple equijoin conditions.

MULTI\_DATASRC\_OPT= is the most efficient option for a single equijoin condition because many key values are processed at once. In contrast, DBINDEX= and DBKEY=, because they generate a separate result set for each unique SAS key value, are less efficient. (DBINDEX= and DBKEY= options operate similarly, with the exception that the DBINDEX= option checks for indexes in the database.)

### MULTI\_DATASRC\_OPT= Option

SAS constructs one or more IN clauses containing the unique key values. SAS passes the IN clause to the database and retrieves the matching rows. SAS/ACCESS products might generate multiple IN clauses depending on the number of distinct key values in the SAS data set. If the number of unique key values is greater than 4500, the processing is performed in SAS.

 **Note:** The IN clause is passed whether an index exists, and multiple IN clauses can cause multiple full table scans if the join key is not indexed. Make sure your join key is indexed on the database.

The following example shows using a MULTI\_DATASRC\_OPT= for an Oracle equijoin. The sasdata.response contains 3 distinct values for CLIENTID (24486, 855433, 3459982).



```

Libname sasdata `.';
Libname dbmsdata oracle datasrc=production MULTI_DATASRC_OPT=IN_CLAUSE ...;
proc sql;
create table sasdata.respout as
select response.RESPCODE, response.RESPDATE,
       client.AGE, client.OCCUPATIONGROUP
from sasdata.response as response, dbmsdata.client_repository as client
where response.CLIENTID = client.CLIENTID;

ORACLE_6: Prepared:
SELECT  "AGE", "OCCUPATIONGROUP", "CLIENTID" FROM CLIENT_REPOSITORY
WHERE   ( ("CLIENTID" IN (24486 , 855433 , 3459982 ) ) )

ORACLE_7: Executed:
SELECT statement ORACLE_6

```

The last line of the code example shows retrieval of one result set.

### DBINDEX= Option

When you set the DBINDEX= option, SAS/ACCESS first queries the indexes on a database table. If a usable index on the join key is found, SAS passes a series of SELECT statements with a specially constructed WHERE clause to the database. A separate database search occurs for each distinct SAS key. The database should use indexes on each search column, especially if database statistics are kept fresh.

A usable index is a database index whose component columns all appear in equijoin conditions in the SAS SQL statement being processed. For example, assume your SAS join is:

```

proc sql;
select * from dbmsdata.client_repository as client, sasdata.respout as response
where response.CLIENTID = client.CLIENTID
and
response.jobcode = client.jobcode
quit;

```

A simple index on (CLIENTID), a simple index on JOBCODE, or a composite index on CLIENTID and JOBCODE causes the selective retrieval of database rows.

In the following example, there is a simple database index on CLIENTID. SAS issues the SELECT with special WHERE clause to retrieve database rows that match the distinct SAS CLIENTID values. The sasdata.response contains 3 distinct values for CLIENTID (24486, 855433, 3459982).

```

Libname dbmsdata oracle datasrc=production ...;
proc sql;
create table sasdata.respout as
select response.RESPCODE, response.RESPDATE,
       client.AGE, client.OCCUPATIONGROUP
from sasdata.response as response, dbmsdata.client_repository (dbindex=yes) as client
where response.CLIENTID = client.CLIENTID;

ORACLE_20: Prepared:
SELECT  "AGE", "OCCUPATIONGROUP", "CLIENTID" FROM CLIENT_REPOSITORY WHERE
((("CLIENTID"=: "CLIENTID") OR (("CLIENTID" IS NULL ) AND ( : "CLIENTID" IS NULL ))))

```

```
ORACLE_21: Executed:
SELECT statement ORACLE_20
```

```
ORACLE_22: Executed:
SELECT statement ORACLE_20
```

```
ORACLE_23: Executed:
SELECT statement ORACLE_20
```

The last lines of the code example show retrieval of three result sets.

### DBKEY= Option

Similar to the DBINDEX= option, the DBKEY= option causes SAS to build a SELECT with a specially constructed WHERE clause to search for key matches in the database table. Unlike DBINDEX=, DBKEY= does not query for database indexes. Rather, DBKEY= specifies the key columns, and SAS constructs the special WHERE clause based on the specified columns.

DBINDEX= is preferable to DBKEY= when you do not have information on indexes. However, if you know the defined indexes, you can use DBKEY= to specify an index column. For example, DBINDEX= might not collect information on an implicit index such as a hashed column that partitions the database table, so you use DBKEY= to specify that column.

In the following example, SASDATA.RESPONSE contains three distinct values for CLIENTID (24486, 855433, 3459982).

```
Libname dbmsdata oracle datasrc=production ...;
proc sql;
create table sasdata.respout as
select response.RESPCODE, response.RESPDATE,
       client.AGE, client.OCCUPATIONGROUP
from sasdata.response as response, dbmsdata.client_repository(dbkey= CLIENTID) as
client
where response.CLIENTID = client.CLIENTID;

ORACLE_20: Prepared:
SELECT "AGE", "OCCUPATIONGROUP", "CLIENTID" FROM CLIENT_REPOSITORY WHERE
((( "CLIENTID" = "CLIENTID" ) OR ( "CLIENTID" IS NULL ) AND ( : "CLIENTID" IS NULL )))
```

```
ORACLE_21: Executed:
SELECT statement ORACLE_20
```

```
ORACLE_22: Executed:
SELECT statement ORACLE_20
```

```
ORACLE_23: Executed:
SELECT statement ORACLE_20
```

The last lines of the code example show retrieval of three result sets.

For more information about these options:

SAS Institute Inc., 2007, SAS online documentation, "Optimizing Your SQL Usage Using the DBINDEX=, DBKEY=, and MULTI\_DATASRC\_OPT= Option" Available

<http://support.sas.com/onlinedoc/913/getDoc/en/acreldb.hlp/a002253008.htm>

---

## Advanced Topics

---

### WHERE Clause Processor versus the Implicit Pass-Through Facility

---

Where possible, the SAS/ACCESS products pass WHERE clauses to the database to reduce the quantity of data that the SAS/ACCESS product must transfer from the database into SAS. By passing WHERE clauses to the database, SAS need not read in the entire database table to perform the filter specified. The impact on performance can be dramatic, especially when processing a large database table.

Implicit pass-through passes queries to the database that contain WHERE clauses. When implicit pass-through is not used, SAS/ACCESS products can use another facility to pass WHERE clauses to the database: the SAS/ACCESS product's WHERE clause processor. This section describes this facility, from this point forward referred to as the WHERE processor, specifically:

- What invokes the WHERE processor;
- How the WHERE processor relates to implicit pass-through processing;
- How to determine whether the WHERE processor passed a WHERE clause;
- Constraints that the WHERE processor shares in common with implicit pass-through;
- Differences between the WHERE processor and implicit pass-through;
- How to code efficient WHERE clauses for the WHERE processor.

#### Invoking the WHERE Processor

The SAS supervisor calls the SAS/ACCESS product's WHERE processor whenever it encounters a SAS PROC or DATA step that specifies SAS WHERE clauses. If the SAS/ACCESS product cannot process the WHERE clause, filtering will occur in SAS.

For example:

```
libname myoralib oracle user=testuser password=testpass;
proc print data=myoralib.personnel;
    where hourlywage <10;
run;
```

In the preceding example, the SAS/ACCESS product generates SQL to access the database table specified in the SAS procedure. The WHERE processor will then parse the WHERE clause, if possible convert it to a database-specific WHERE clause, and then append any converted WHERE clause to the SQL statement.

#### How the WHERE Processor Relates to Implicit Pass-through

The SAS/ACCESS product's WHERE processor and implicit pass-through are independent facilities. However, the two share both similarities and differences.

PROC SQL first attempts to pass a query to the database using implicit pass-through. If implicit pass-through fails, the WHERE processor is then called to attempt to process WHERE clauses in the query to

database. If both implicit pass-through and the WHERE processor fail to process the WHERE clause, then filtering occurs in SAS.

### Determining Whether the WHERE Processor Has Passed a WHERE Clause

As with implicit pass-through, you use the SASTRACE option with the syntax, `sastrace=",,,d"`, to see the SQL that is sent to the database. You can then determine whether the WHERE processor has passed the filter to the database.

Example: Verifying a WHERE clause passed to Oracle

```
options sastrace=",,,d" sastraceloc=saslog no$stsuffix;
proc print data=db2.joindate; where dateval > '01jan1960'd;
run;
```

The following SASTRACE output shows an initial prepare statement of the database table, followed by a second prepare statement of the same table. The second statement includes an Oracle specific WHERE clause indicating that the SAS date was converted to an Oracle specific date. Thus, the SASTRACE output verifies that the WHERE clause was passed down to Oracle.

```
proc print data=db2.joindate; where dateval > '01jan1960'd; run;
DEBUG: Open Cursor - CDA=2058853128
DEBUG: PREPARE SQL statement:
SELECT * FROM JOINDATE
DEBUG: PREPARE SQL statement:
SELECT "DATEVAL" FROM JOINDATE WHERE ("DATEVAL"
>TO_DATE('01JAN1960','DDMONYYYY','NLS_DATE_LANGUAGE=American')) )
```

For more information, see the [SASTRACE Threaded Read trace](#)

section.

### Constraints the WHERE Processor Shares in Common with Implicit Pass-Through

When SAS performs the filter, problems that affect Implicit Pass-Through, such as processing large numeric types in SAS, will also affect the WHERE processor. (See [Understanding the Differences between SAS and Database Native Data Types](#)

section.)

The WHERE processor and implicit pass-through have in common the processing constraints listed below. The WHERE processor is more restrictive in what it can pass to the database because it lacks back-down behavior of implicit pass-through. For more information, see the [Difference in Behavior](#) section.

#### ***Cannot pass SAS functions specified in the WHERE clause that do not have a database equivalent***

When the WHERE clause contains a SAS function that does not have a database equivalent the SAS/ACCESS product cannot pass it to the database. In the first example, SAS/ACCESS to ORACLE successfully generates the WHERE clause because it maps the SAS LOWCASE function to the ORACLE equivalent 'LOWER' function:

Example 1: Specifying a SAS function that has a database equivalent

```
proc print data=ora.joinchar;
where LOWCASE(name) = 'Alison';run;
```

### SASLOG Output

```
18          proc print data=ora.joinchar;
19          where LOWCASE(name) = 'Alison';run;
```

```
ORACLE_2: Prepared:
SELECT  "NAME", "AGE", "CLASS", "RLVL", "MLVL", "GPA" FROM JOINCHAR WHERE (
LOWER("NAME") =
'Alison' )
```

```
ORACLE_3: Executed:
SELECT statement ORACLE_2
```



**Note:** Using PROC PRINT in this query means that Implicit Pass-Through does not occur.

In the second example, SAS must perform the filter because the WHERE clause contains the 'TRIM' function that SAS/ACCESS to DB2 does not map to a DB2 function.

### Example 2: Specifying a SAS function that has no database equivalent

```
proc print data=db2.joinchar;
where TRIM(name) = 'Alison';run;
```

### SASLOG Output

```
18          proc print data=db2.joinchar;
19          where TRIM(name) = 'Alison';run;
```

```
DB2_2: Prepared:
SELECT  "NAME", "AGE", "CLASS", "RLVL", "MLVL", "GPA" FROM JOINCHAR FOR READ ONLY
```

```
DB2_3: Executed:
Prepared statement DB2_2
```

For more information about the use of SAS functions, see the Restricting SAS Functions section.

### **Cannot perform arithmetic expressions on non-numeric columns**

The WHERE processor cannot perform arithmetic operations on non-numeric data columns; the processing must be performed in SAS.

The following example contains a WHERE clause that specifies relative date processing. Because the WHERE processor cannot perform arithmetic operations on the date field, the WHERE clause cannot be passed to the database; instead, it must be processed in SAS.

```
/* Find orders that shipped more than seven days after order */
select account, ship_date from dbms.neworders where WHERE ship_date > order_date + 7
```

***NULL values in the database table data can produce different results***

When a database column is specified in a WHERE clause that contains NULL values, you can obtain different results depending on whether SAS or the database processes the WHERE clause. The database tends to remove NULL values from consideration when processing a WHERE clause while SAS does not. To get around the problem, you can obtain SAS behavior by including COLUMN IS NULL or COLUMN is not NULL when you code your WHERE clause.

[For more information reference [NULL Values and MISSING Values](#)



**Note:** If you need to prevent WHERE clauses entirely from being passed to the database, use DIRECT\_SQL= NOWHERE.

**Differences between the WHERE Processor and Implicit Pass-through**

There are differences between the WHERE processor and Implicit Pass-Through in WHERE-clause processing:

***Accesses a Single Database Table***

The WHERE processor operates only on a single database table.

***Interacts With More SAS Options***

Both Implicit Pass-Through and the WHERE processor interact with the LIBNAME option DIRECT\_SQL= (explained below). However, the WHERE processor interacts with several other SAS options, such as DBSLICE=, DBSLICEPARAM=, and DBCONDITION=. (Recall that data set options disable implicit pass-through.)

DIRECT\_SQL=

This option specifies whether the SQL generated by the SAS/ACCESS product should be passed to the database for processing. (The default is YES.) The following values affect the WHERE processor. The settings operate only on elements that are eligible for database processing:

DIRECT\_SQL=NONE

Specifies that all query processing be performed in SAS. This setting also turns off Implicit Pass-Through and the WHERE processor.

DIRECT\_SQL=NOWHERE

Specifies that eligible WHERE clauses be performed in SAS. This setting is less restrictive than NONE, and affects Implicit Pass-Through and the WHERE processor.

**DIRECT\_SQL=NOFUNCTIONS**

Specifies that eligible SAS functions in a WHERE clause be processed in SAS. This setting is less restrictive than NOWHERE, and affects Implicit Pass-Through and the WHERE processor.

For additional information:

SAS Institute Inc., 2007, SAS online documentation, "The LIBNAME Statement for Relational Databases DIRECT\_SQL= LIBNAME Option" Available

(<http://support.sas.com/onlinedoc/913/getDoc/en/acreldb.hlp/a002205898.htm>)

**DBSLICE= and DBSLICEPARM=**

These threaded read options are explained in the "Advanced Topics: Threaded Reads" section.

**DBCONDITION=**

This data set option enables you to pass database-specific SQL conditions to the database. The DBCONDITION is appended to the SAS WHERE clause.

Example: Use of DBCONDITION= with a SAS WHERE clause for SAS/ACCESS to Oracle:

```
proc print data=ora.join1(dbcondition="order by x1"); where x1 > 0;
run;
```

**Log Output**

The SQL that is passed to Oracle:

```
SELECT "X1" FROM JOIN1 WHERE ("X1" > 0 ) ORDER BY x1
```

For more information, see: SAS Institute Inc., 2007, SAS online documentation, SAS/ACCESS: Relational Databases/ Data Set Options for Relational Databases/DBCONDITION= Data Set Option Available

(<http://support.sas.com/onlinedoc/913/getDoc/en/acreldb.hlp/a001371534.htm>)

***Performing Partial WHERE-Clause Filtering***

The WHERE processor can selectively process filters specified in eligible WHERE clauses. This capability which is not well known can be best explained with an example.

Assume that you have the following WHERE clause:

```
where emptitle = 'salesrep' and month(hiredate) = 3 and status = 'FULL';
```

This WHERE clause contains three filter expressions: employee title, employee status, and month hired, which uses the SAS function MONTH(). Implicit Pass-Through fails the query for SAS/ACCESS interfaces such as Oracle that cannot map MONTH to an equivalent database function. Similarly, the WHERE processor can fail the month hired filter for the same reason.

However, in contrast to Implicit Pass-Through, the WHERE processor successfully passes the two other

filter expressions (employee title and employee status) to the database—this is partial WHERE clause filtering.

The WHERE processor improves performance because the data that SAS must transfer from the database to SAS is reduced. As mentioned, whenever SAS can successfully pass filters to the database for processing, SAS avoids having to transfer the entire table to perform the filter in SAS. Instead, the database performs the work, and then returns to SAS the filtered data, usually a much smaller subset of the table.

Example:

```
proc sql;
/* salesreps hired in March */
select empnum, empname from dbms.employee
where emptitle = 'salesrep' and month(hiredate) = 3 and status = 'FULL' ;
quit;
```

### SASLOG Output

```
DB2: AUTOCOMMIT turned ON for connection id 0
DB2_1: Prepared:
SELECT * FROM EMPLOYEE FOR READ ONLY
DB2: COMMIT performed on connection 0.
DB2_2: Prepared:
SELECT "EMPNUM", "EMPNAME", "EMPTITLE", "HIREDATE", "STATUS" FROM EMPLOYEE
WHERE ( ( "EMPTITLE" = 'salesrep' ) AND ( "STATUS" = 'FULL' ) ) FOR READ ONLY
DB2: COMMIT performed on connection 0.
DB2_3: Executed:
Prepared statement DB2_2
```

### Coding Efficient WHERE Clauses For the WHERE Processor

In general, the guidelines that are suggested in the Constructing a WHERE Clause Correctly section will optimize WHERE clauses processed by the WHERE processor as well.

To encourage partial filtering, the order of the expressions might be important when you specify a WHERE clause with multiple predicates. Place WHERE predicates that can be passed first, followed by predicates that cannot be passed. This ensures that all possible WHERE predicates will be passed to the database.

### Preventing Unneeded SQL Dictionary Queries

---

A PROC SQL Dictionary query provides detailed table metadata for one or more librefs. An improperly constructed query might accidentally retrieve table metadata for every libref, including librefs that are assigned to a database. Fetching table metadata from a database means querying database dictionary or system tables. While occasional database queries are necessary, excessive queries affect database performance and are a red flag to the database administrator.

Avoid excessive database system table queries with two best practices:

1. Use a WHERE clause to restrict Dictionary queries to only the needed libref(s).



- In the WHERE clause, avoid applying functions to the LIBNAME column.

A common mistake is unnecessarily using the UPCASE function. UPCASE or other functions on the LIBNAME column cause all librefs to be queried for table information. This causes unnecessary database dictionary queries and degrades PROC SQL Dictionary performance.



**Note:** Because SAS internally stores librefs in uppercase, UPCASE is unnecessary. Never apply the UPCASE function to the LIBNAME column. However, the UPCASE function can be applied to other Dictionary columns without adverse effect.

The following code presents a good example of a well-constructed Dictionary query:

```
PROC SQL;
create table table_info as
    select MEMNAME as tabname, NAME as colname, LENGTH as collen
    from DICTIONARY.COLUMNS
    where LIBNAME = "WORK" and MEMNAME="ZTEMPZ";
quit;
```

```
NOTE: Table WORK.TABLE_INFO created, with 2 rows and 3 columns.
NOTE: PROCEDURE SQL used (Total process time):
      real time          0.06 seconds
      cpu time           0.01 seconds
```

In contrast to the first example, the following code presents a poorly constructed Dictionary query.

The UPCASE function on the LIBNAME column forces information collection on all assigned librefs. In this example, the librefs TERA1 and TERA2 are logged into the Teradata SASUSER1 and SASUSER2 accounts. . The UPCASE inadvertently forces queries of each Teradata libref. (For clarity, extraneous trace information was deleted from the example.)

```
PROC SQL;
create table table_info as
    select MEMNAME as tabname, NAME as colname, LENGTH as collen
    from DICTIONARY.COLUMNS
    where UPCASE(LIBNAME) = "WORK" and MEMNAME="ZTEMPZ";
quit;
```

```
TERADATA_1: Executed:
SELECT TableName, CASE TableKind WHEN 'V' THEN 'VIEW' ELSE 'TABLE' END FROM
DBC.TablesX WHERE
((TableKind = 'T' OR TableKind = 'V') AND UPPER(DatabaseName)='SASUSER1')
```

```
TERADATA_3: Executed:
SELECT TableName, CASE TableKind WHEN 'V' THEN 'VIEW' ELSE 'TABLE' END FROM
DBC.TablesX WHERE
((TableKind = 'T' OR TableKind = 'V') AND UPPER(DatabaseName)='SASUSER2')
```

```
NOTE: Table WORK.TABLE_INFO created, with 2 rows and 3 columns.
```

```
NOTE: PROCEDURE SQL used (Total process time):
      real time          2.48 seconds
      cpu time           0.09 seconds
```

The following code presents a good example of using the UPCASE function with Dictionary columns:

```

PROC SQL;
create table table_info as
    select MEMNAME as tabname, NAME as colname, LENGTH as collen
    from DICTIONARY.COLUMNS
    where LIBNAME = "WORK" and UPCASE(MEMNAME)="ZTEMPZ";
quit;

```

NOTE: Table WORK.TABLE\_INFO created, with 2 rows and 3 columns.

NOTE: PROCEDURE SQL used (Total process time):

```

real time      0.14 seconds
cpu time       0.03 seconds

```

## Bulk Loading the Relational Database

One of the fastest ways to load large data volumes into a relational database is to use the bulk loading capabilities of the database. By default, the SAS/ACCESS engines load data into tables by preparing an SQL INSERT statement, executing the INSERT statement for each row, and periodically issuing a COMMIT. If you specify **BULKLOAD=YES** as a data set or LIBNAME option, a database bulk load method is used, which can significantly enhance performance. This section presents the advantages and disadvantages of using bulk loading methods. Additional information is available in SAS/ACCESS online documentation, SAS/ACCESS white papers, and database vendor bulk load documentation. (The term “bulk load” is not useful in finding or navigating database documentation; instead, look for the load methods such as SQL\*Loader and FastLoad as shown in Table 3.)



**Note:** Some SAS/ACCESS engines implement BULKLOAD=YES as a LIBNAME option, however, this is not supported across all SAS/ACCESS engines. Therefore, specify BULKLOAD= as a data set option for reliable behavior across SAS/ACCESS engines.

Each SAS/ACCESS engine invokes a different loader and uses different options. Some SAS/ACCESS engines support multiple loaders. The following table shows the different loaders that are available when the option BULKLOAD is set to YES. (For more information about how to use bulk load for each SAS/ACCESS engine, see the SAS online documentation.)

SAS/ACCESS Product	Load methods available for BULKLOAD=YES
Sybase	Open Client bulk load API
Oracle	SQL*Loader Utility
ODBC	Bulk Copy facility (BCP) <sup>[1]</sup>
Teradata	FastLoad API MultiLoad Utility
DB2 UDB	IMPORT API LOAD API CLILOAD API

DB2/zOS	DB2 Load Utility
Netezza	Remote External Table interface
HP Neoview	Transporter API
OLE/DB	IRowSetFastLoad API

<sup>[1]</sup> Supported only for Microsoft SQL Server on Windows.

Table 3. SAS/ACCESS Bulk Load Methods

### Why Bulk Load?

The primary advantage of bulk load is the speed of the load. The difference in processing time between a bulk load and a standard insert can be significant. For large load processes, a shorter processing time can mean the difference between completing a project within the required time frame or failing to meet the required deadline.

In general, bulk loading of indexed database tables is performance-optimal. When you use a bulk loader, both data and index table components are loaded in bulk. Thus, you may avoid the costly standard insert alternatives of dropping and re-creating indexes or database index updates on each row that is inserted.



**Note:** Teradata bulk load methods have limited index support. See the section “Database Indexes Might Inhibit Bulk Load For Teradata”.

### Bulk Load Overhead Cost

Using a database bulk loader incurs some additional overhead. A bulk loader engages in a special multi-phase protocol with the database server. The protocol phases take time. By comparison, standard insert is relatively overhead-free, therefore, on very small numbers of rows, standard insert will outperform bulk load.

Overhead can be experienced with some database bulk load utilities for index and constraint processing after the load completes. For example, after an append operation with the Oracle SQL\*Loader Utility using the Direct Path load method, the existing index is copied when merged with the new index keys. Costs could outweigh the savings for appending a small number of rows into a large table. The default load method for SAS/Access to Oracle is Direct Path loading.

### Cost for Concurrent Access

When used to append rows into a table, access to the table or its indexes may be restricted for other users. For example, the DB2 LOAD utility by default locks the table for exclusive access until the load completes. No other application can access the target table that is being loaded.

### Complexities of Bulk Load

Bulk load reduces load time but introduces complexity. The following sections explain the complexities

that are associated with bulk load and include as-needed specifics for Oracle, DB2, and Teradata.

- Additional database-vendor-supplied software might be required
- SAS might need to be configured to allow the X command
- Database-enforced referential integrity (RI) might inhibit bulk load
- Database indexes might inhibit bulk load for Teradata
- Deferred error detection can cause unexpected behavior
- Error correction might be difficult
- The ERRLIMIT option might not be honored

***Additional Database-Vendor-Supplied Software Might Be Required***

SAS/ACCESS engines either call a vendor API or invoke a vendor-supplied bulk load utility program to execute a bulk load. The latter case requires that the appropriate utility program be installed. This is an additional software requirement to vendor-supplied native libraries such as OCI for Oracle and CLIV2 for Teradata.

**Oracle:** Bulk load for SAS/ACCESS to Oracle uses the SQL\*Loader Utility. SQL\*Loader (typically named SQLLDR.EXE) must be included in your Oracle client software install.

**DB2:** No additional software is required.

**Teradata:** The MultiLoad method for SAS/ACCESS to Teradata uses the MultiLoad Utility. The MultiLoad Utility (typically named mload.exe) must be included in your Teradata client software install.

***SAS Might Need to be Configured to Allow the X Command***

The X command temporarily exits the SAS System to the host system or operating environment to allow operating system-level commands to be issued. For example, you can use the X command to run another program that you might otherwise run from an operating system command prompt.

The two SAS/ACCESS engines (Oracle and Teradata using MultiLoad) that invoke a vendor-supplied bulk load utility program essentially generate an X command internal to SAS. By default, SAS allows the X command but the SAS Business Intelligence environment does not. If this default has been modified, adding `-XCMD` to the SAS invocation enables the X command.

When deployed as a back-end server such as a SAS Workspace Server or a Stored Process Server in SAS®9 Business Intelligence, SAS defaults disallow the X command. Enabling the X command to allow bulk loading can be a security risk because it allows SAS users to reach a command prompt. You might want system administrator agreement before enabling the X command.

To enable the X command, perform the following tasks:

- Stop the object spawner service and then remove it.

- In the file that defines the service (typically named ObjectSpawner.bat on Windows), add `--allowxcmd` to the line that launches the spawner (the line typically begins with `start /b "object spawner"` on Windows).
- Re-install the service and restart the spawner.

**Oracle:** Bulk load always uses the SQL\*Loader Utility, which requires the X command be enabled. With SAS® 9.2 the SAS/Access to Oracle interface briefly enables the X command internally to invoke the SQL\*Loader utility, then restores the SAS Business Intelligence environment default setting. Configuring the SAS System to allow the X command as described above should be unnecessary for bulk loading of Oracle tables on back-end server deployments. This behavior is available for SAS® 9.1.3 though a SAS Technical Support supplied hot fix.

**DB2:** All DB2 bulk load methods are via API. The X command need not be enabled.

**Teradata:** The Teradata MultiLoad method uses a utility and requires that the X command be enabled. The FastLoad method does not require that X command be enabled.

#### **Database-Enforced Referential Integrity (RI) Might Inhibit Bulk Load**

Most data models that are implemented in a database enforce referential integrity (RI) with database foreign-key constraints. RI complicates inserting data into the database whether you use standard insert or bulk load. However, there are special considerations for bulk loaders.



**Note:** Some SAS solutions enforce RI in the application, rather than asserting database foreign keys. Because there is no database-enforced RI, there are no additional considerations for bulk load.

Some bulk load methods, such as Teradata FastLoad and MultiLoad, require that you remove foreign-key constraints before bulk loading. Other bulk load methods, such as Oracle SQL\*Loader and DB2 CLILOAD and LOAD, disable foreign keys on your behalf. With foreign-key constraints disabled or removed, illegal data (that is, data that violates RI constraints), can be loaded into tables. The problem of illegal data can be addressed with either load pre-processing to ensure sound data, or load post-processing to remove illegal data.

**Oracle:** Foreign-key constraints do not prevent bulk loading with Oracle. Conventional Path loading maintains RI constraints during the load. Direct Path loading automatically disables check and foreign-key constraints before the load. Constraints must be applied to the table after the load completes.

Load performance may be improved by disabling constraints, performing the load, then reinstating the constraints. Overall performance may be improved by using the Conventional Path load method, or standard inserts when a small number of rows are to be appended to a very large table.

For more information, see the `BL_LOAD_METHOD` option in the Oracle Bulk Loading section of SAS Institute Inc., 2007, SAS online documentation, "SAS/ACCESS: Relational Databases: SAS/ACCESS Interface to Oracle Reference" at <http://support.sas.com/onlinedoc/913/getDoc/en/acreldb.hlp/a001234460.htm>.

**DB2:** Depending on which bulk load method is used, constraints can cause an issue with bulk load. There are no constraint issues if you use the IMPORT method. However, the LOAD and CLILOAD

methods load the DB2 table but put it in a “check pending (SQLCODE 57016)” state.

An additional step is required to release the table from the “check pending” state. This step consists of a single DB2-specific SQL statement that can be issued with explicit SQL. Let’s assume that the table is named “products”, then one of the following two SQL forms can be issued to release the “check pending” state:

```
SET INTEGRITY FOR products IMMEDIATE CHECKED
```

releases the “check pending” state and performs the deferred integrity checking

```
SET INTEGRITY FOR products ALL IMMEDIATE UNCHECKED
```

releases the “check pending” state but omits integrity checking

\_ **Note:** The LOAD and CLILOAD methods automatically turn off integrity checking. Therefore, they will load rows that violate constraints. Constraint violations are flagged by the CHECKED variant of SET INTEGRITY.

**Teradata:** Foreign-key constraints prevent both Teradata bulk load techniques (FastLoad and MultiLoad). To bulk load a table with foreign-key constraints, first submit the appropriate SQL to drop the constraints, bulk load the table, and then reinstate the constraints. This technique requires data that is verified as clean; otherwise, you risk loading rows that violate the constraints.

#### ***Database Indexes Might Inhibit Bulk Load For Teradata***

Oracle and DB2 permit bulk loading of database tables that have indexes defined. The effect of having indexes defined varies for Teradata based on the bulk load method that is used.

Both Teradata bulk load techniques might fail if indexes are defined on the table that is being loaded. For FastLoad, no indexes may be defined. For MultiLoad, indexes are allowed if they are non-unique. For more information, see the Teradata FastLoad Reference and the Teradata MultiLoad Reference.

#### ***Deferred Error Detection Can Cause Unexpected Behavior***

When all the data is clean, the result of a bulk load is usually identical to standard insert. One exception is the Teradata FastLoad method that drops any duplicate rows. However, if the database generates an error on one or more rows, bulk load behaves differently than standard insert under default conditions.

SAS defaults the ERRLIMIT option to 1, which terminates a SAS step when the first error condition is generated. With standard insert, if the database generates an error inserting a row, the error code is immediately returned to SAS and the step is terminated. For example, if the database generates an error on observation 200,000, the SAS step terminates and a rollback is issued to the database table. Between 0 and 199,999, rows are actually committed to the database table based on the DBCOMMIT option setting.

With bulk load, SAS is not informed of errors on an “as you go” basis. Instead, all rows are passed to the bulk loader and the bulk loader passes them to the database. The result is that all clean rows are loaded into the database table, and error rows are logged in a database-specific way by the bulk loader. This behavior is similar to standard insert with ERRLIMIT=0 (no error limit).

**Error Correction Might Be Difficult**

As noted earlier, SAS passes all rows to the bulk loader, and the bulk loader passes them to the database. Clean rows are loaded into the database table, and error rows are logged in a database-specific way (usually in flat files or in database error tables). Therefore, error correction will involve database-specific techniques. Generally, helpful information about bulk load errors, such as where error rows are logged, is output to the SAS log by the SAS/ACCESS engines. Additional information to help in error correction is available in SAS/ACCESS documentation and database vendor documentation.

**ERRLIMIT Might Not Be Honored**

For standard insert, the ERRLIMIT option in SAS/ACCESS specifies the number of errors that are allowed before SAS stops processing and issues a rollback. Some SAS/ACCESS bulk loaders ignore the ERRLIMIT option that stops processing.

The performance and disk space penalty of many rejected rows can be high, depending on how the bulk loader and database intercept and log error rows. Limiting the allowed number of error rows can prevent a runaway step with bad data. Also, you might not want to continue loading rows after one or more error rows is detected.

Here are database-specific details about limiting error rows.

**Oracle:** The ERRLIMIT option is not honored. By default, the limit is 50 error rows. SQL\*Loader limits errors by using the ERRORS keyword. You can pass the ERRORS keyword setting to SQL\*Loader by using the option BL\_OPTIONS.

**DB2:** The ERRLIMIT option is not honored. By default, there is no error limit. To limit error rows, use the option BL\_WARNING\_COUNT. .

**Teradata:** The FastLoad method honors the ERRLIMIT option. The MultiLoad method does not honor the ERRLIMIT option. By default, there is no error limit. No override is available.

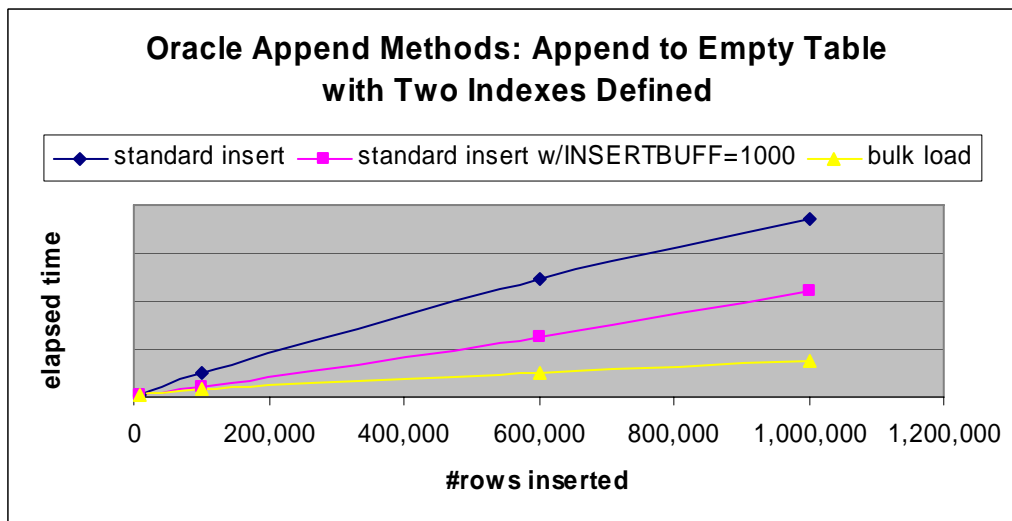
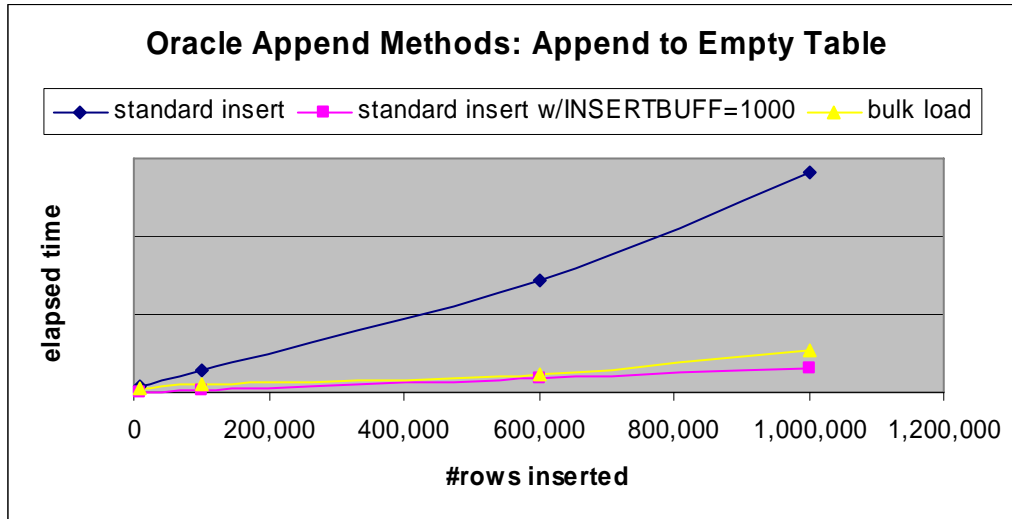
**Bulk Load Performance**

This section discusses the relative performance of bulk load methods and standard insert by database. It gives an indication of comparison points, for example, the number of rows where one method might begin to outperform another method. This information was derived from very limited testing. It cannot be considered accurate across all operating environments. Therefore, it is only a starting point for performance tuning.

**Oracle**

The default bulk load converts all the data to be loaded to SQL\*Loader format and stores it in a flat file. SQL\*Loader then loads the data from the flat file.

On non-indexed database tables, standard insert with the INSERTBUFF= options set, can perform as well as either bulk load or bulk load with the pipe option.



On UNIX, bulk load supports a pipe option by specifying `BL_USE_PIPE=YES` in addition to `BULKLOAD=YES`. The pipe option converts rows to SQL\*Loader format and transfers them to SQL\*Loader via an operating system pipe. An advantage of the pipe option is that it eliminates the disk space requirement for a temporary copy of all the data. The pipe option will underperform default bulk load on very narrow tables (for example, 40 bytes wide), but it will outperform default bulk load on very wide tables (for example, 1500 bytes wide). Experimentation is required to determine the performance comparison point.



**Note:** The pipe option in the SAS/ACCESS Interface to Oracle on UNIX is not depicted in the charts below.

#### DB2

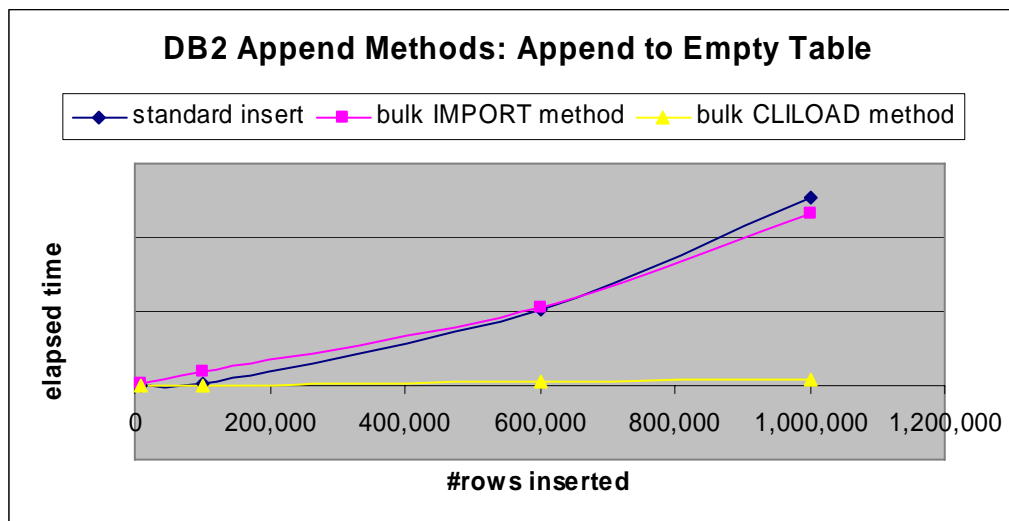
In general, tuning standard insert with the `INSERTBUFF` option is unnecessary. SAS/ACCESS attempts to optimize the `INSERTBUFF` option on your behalf.



As a rule-of-thumb, the CLILOAD method will outperform other bulk load methods.

If the option BULKLOAD=YES is the only option that is specified, the IMPORT method is used as the default method. IMPORT is a slow bulk load method. If you specify BULKLOAD=YES, you should also specify BL\_METHOD=CLILOAD or the appropriate options for the LOAD method if high performance is your objective.

The LOAD method requires that SAS and DB2 be located on the same machine. To use the LOAD method, you must have system administrator authority, database administrator authority, or load authority on the database. The LOAD method is not depicted in the following chart.



### Teradata

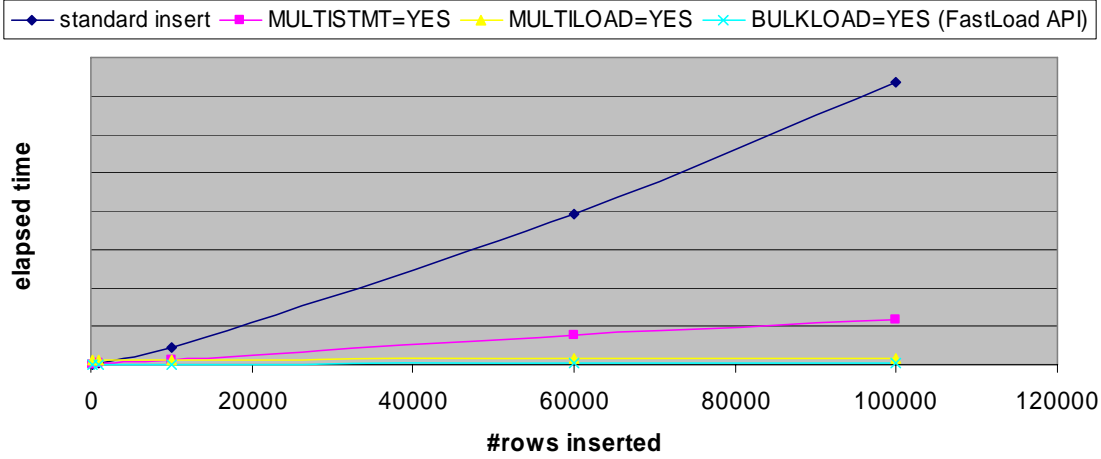
Introduction of the term “secondary index” is necessary for Teradata. A *secondary index* in Teradata is comparable to a standard user-defined index in other databases. Teradata calls the table hash partition key the “primary index” (all Teradata tables are hash partitioned, which means that all the tables have a primary index). Whereas the primary index is required, secondary indexes are optional, as in other databases.

The FastLoad method loads only to an empty database table that has no secondary indexes defined. An attempt to load a non-empty table or a table that has a secondary index defined fails with no rows processed. FastLoad drops duplicate rows (a *duplicate row* is a row whose column values all match those of a row already inserted into the table), which might be an undesirable behavior.

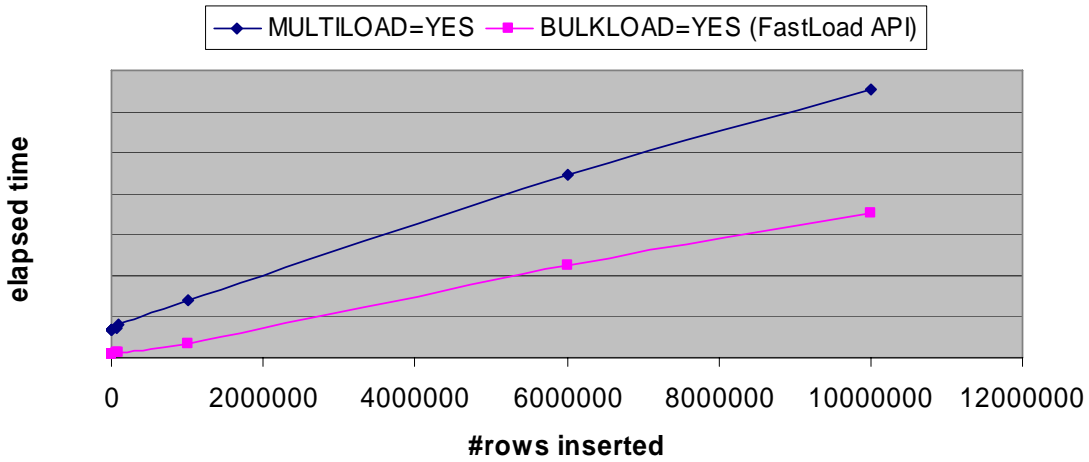
The MultiLoad method cannot load a table that has a unique secondary index defined. The MultiLoad method has a documented recovery mechanism whereby an interrupted load can be restarted.

The database limits the number of system-wide simultaneous FastLoad and MultiLoad executions to as few as 15. SAS is only one application that might FastLoad or MultiLoad. Your bulk load might either fail or be delayed if 15 executions are already active. To be eligible for delay rather than failure, consider the TENACITY option in SAS/ACCESS Interface to Teradata.

### Teradata Append Methods: append to empty table



### Teradata Append Methods: FastLoad versus MultiLoad comparison



## Threaded Reads

---

A threaded read can reduce the elapsed time your SAS step takes to complete by increasing throughput from the database to SAS. In contrast to a standard read that uses a single database connection, the SAS/ACCESS engine generates multiple threads that retrieve database table data across multiple connections. Each thread opens a separate connection, reads a partition or 'slice' of the database result set. SAS commonly uses a WHERE clause to effect auto-partitioning of the result set, but other means can be used as well. The threads then—in parallel—transfer that data across the multiple connections to SAS. Not all SAS/ACCESS products offer support for threaded reads. For more information, see the [Threaded Reads](#) section of this paper.



**Note:** Writes are always single-threaded.

### Setting Performance Expectations

A threaded read does not guarantee enhanced performance; and, in some cases, can even diminish read performance as explained below.

#### *Potential Gains*

The potential gain for a threaded read over a standard single-connection database read varies by database, as well as by how well the data is set up for partitioning. The factor that can provide the best threaded read characteristics for key databases is listed below:

- Teradata: The Teradata utility, FastExport must be running on the same machine as SAS. When SAS/ACCESS Interface to Teradata detects the utility, it calls FastExport appropriately to perform table autopartitioning reads. If SAS threaded reads are turned off, FastExport is not used.
- Oracle: The Oracle table should be evenly partitioned, with uniform distribution.
- DB2/UDB: A knowledgeable DB2 user in a distributed or partitioned DB2 environment, can specify WHERE clauses with DB2-specific options that outperform the WHERE clauses generated with autopartitioning. The use of DB2-specific options is discussed in database-specifics in the "Optimizing Threaded Reads" section.

#### *Potential losses*

A threaded read has the potential to slow performance because it consumes more resources in SAS and on the database than a single thread. For example, the overhead associated with the multiple database connections for threaded reads is higher than that for single threads.



**Note:** You should always test to see whether a threaded read actually provides better performance than a standard read.

### Automatic Threaded Reads Versus User-Controlled Threaded Reads

Threaded reads can be generated by SAS/ACCESS via threaded procedures; they can also be generated by the user.

For automatic threaded reads, the SAS/ACCESS engine determines the data partitioning schemes, along with default values for the threaded read options. By default, threading is set to ON for procedures that support threaded reads.

For user-controlled reads, the user specifies threaded read options with WHERE clauses.

This section will discuss both categories of threaded reads in detail.

For complete information on threaded reads:

SAS Institute Inc., 2007, SAS online documentation, "SAS/ACCESS: Relational Databases: Concepts: Threaded Reads" Available

(<http://support.sas.com/onlinedoc/913/getDoc/en/acreldb.hlp/a002238650.htm>)

### Understanding When Automatic Threaded Reads Occur

In SAS®9 and later, the SAS/ACCESS engine automatically threads database reads when both of the following conditions are met:

- A SAS 'I/O threaded' application (explained next) is executed
- The SAS/ACCESS engine is able to auto-partition the database table. Auto-partitioning is explained later in this section.

#### ***SAS Threaded Procedures That Activate Database Threaded Reads***

By default, automatic threaded reads are generated only for SAS procedures that have a threaded I/O subsystem. These procedures are referred to as 'I/O threaded' applications. The list, which includes the SAS procedures REG, DMREG, DMINE, DMDB, and SORT, is likely to expand in the future.



**Note:** PROC SQL is not an I/O threaded procedure, but it can use threaded reads in some situations.

#### ***Auto-partitioning Schemes***

SAS/ACCESS products that support threaded reads share an auto-partitioning scheme based on modulo arithmetic, using the MOD function. Additionally, Teradata and Oracle can use other auto-partitioning schemes.

#### **Auto-partitioning schemes using modulo-arithmetic (MOD function) on a table column**

In this scheme SAS/ACCESS checks whether the database table contains a non-NULL numeric column (identity; integer, or certain decimal types) that is suitable for modulo arithmetic. If the table contains an eligible column, SAS appends to your SQL a WHERE clause that applies the MOD function to that column. The WHERE clause creates a subset of the result set; then combines the subsets to form the result set for your original SQL query.

In the following example from SAS/ACCESS online documentation, the original SAS generated SQL is

```
SELECT CHR1, CHR2 FROM DBTAB
```

After the SAS/ACCESS engine API determines that the DBTAB table contains a NOT NULL integer column IntCol, it generates two threads and issues:

```
SELECT CHR1, CHR2 FROM DBTAB WHERE (MOD(INTCOL,2)=0)
```

and

```
SELECT CHR1, CHR2 FROM DBTAB WHERE (MOD(INTCOL,2)=1)
```

### Other schemes

Table 4 shows the default behavior of the auto-partitioning that is done by the SAS/ACCESS engine. Note that for Teradata and Oracle, if the default behavior cannot be performed the fallback is to use modulo arithmetic.

Database	Default	Fallback
Teradata	Detects whether the Teradata FastExport utility is available; if so, it uses the utility to auto-partition the table.	If FastExport is unavailable, performs modulo arithmetic on an appropriate table column in a WHERE clause.
Oracle	Detects whether Oracle table is physically partitioned in Oracle; if so, it auto-partitions the table.	If the Oracle table is not physically partitioned, performs modulo arithmetic on an appropriate table column in a WHERE clause.
DB2	Performs modulo-arithmetic on an appropriate table column in a WHERE clause.	

Table 4. Auto-partitioning Schemes by Database



**Note:** The default column that is selected for modulo arithmetic varies because each SAS/ACCESS engine has its own rules for column selection. See the SAS/ACCESS documentation for more information.

### Controlling Threaded Reads

As explained above, SAS/ACCESS products generate threaded reads when you run an 'I/O threaded' SAS procedure and when SAS/ACCESS can auto-partition the database table. However, you can control reads are generated and also how many, and you can control this behavior whether you are using an I/O-threaded SAS procedure.

Some reasons why you might want to control whether threads are generated and how many threads are generated:

- You find that threaded reads enhance performance and want to expand their use.
- You find that threaded reads degrade performance for a table and want to turn threaded reads off.
- You find that SAS/ACCESS is unable to automatically generate threaded reads, but you want to try using and controlling them.
- You need to limit or narrow the scope of threaded reads. For example, your site might have a limit on the number of connections that can be made to a database.

You can use the DBSLICE= option to generate threaded reads by specifying your own partitioning WHERE clauses, or you can use the DBSLICEPARM= option to expand or limit the threaded reads that SAS/ACCESS generates.

#### ***Use of the DBSLICE= Data Set Option For User-Controlled Threaded Reads***

If the SAS/ACCESS engine cannot generate a threaded read using auto-partitioning, you can enable a threaded read by specifying your own partitioning rules with the DBSLICE= data set option. You might also choose to use this option to specify threaded reads that can outperform the threaded reads that the SAS/ACCESS engine generates.

For details of user-specified WHERE clauses to perform threaded reads, see online documentation SAS/ACCESS Data Set Options for Relational Databases: DBSLICE= Data SET Option at <http://support.sas.com/91doc/getDoc/acreldb.hlp/a002154105.htm>. Each SAS/ACCESS product has its own values, see the SAS/ACCESS documentation for more information.

#### ***Use of the DBSLICEPARM= Option***

The DBSLICEPARM=option can be used as a configuration, a system, a LIBNAME statement, or a data set option. Table 5 shows the effect of the DBSLICEPARM= value on threaded reads:

Option Value	Effect on Threaded Reads
DBSLICEPARM=THREADED APPS	(Default) Attempts a threaded read only when the SAS/ACCESS engine is called by a SAS 'I/O threaded' procedure (explained above).
DBSLICEPARM=ALL	Generates the most threaded reads. SAS/ACCESS engine attempts to generate a threaded read even if there is no SAS 'I/O threaded' application invocation.
DBSLICEPARM=NONE	Turns off all threaded reads.

Table 5. Settings for the DBSLICEPARM= Option

The default number of threaded reads is 2. You can set the number of threaded reads with a second parameter for DBSLICEPARM=. See the online documentation for more information.

**When Threaded Reads Do Not Occur**

Regardless of the DBSLICE= and DBSLICEPARM= options settings, a threaded read will not be performed in any of the following situations:

- When a BY statement is used in a DATA step or a PROC other than PROC SORT
- When the OBS or the FIRSTOBS option is specified
- When the KEY or the DBKEY option is specified
- When the Implicit Pass-Through Facility processes the query

Further, with unless FastExport is available for Teradata or tables are partitioned in Oracle, SAS/ACCESS engines use modulo arithmetic for generating automatic threaded reads. Modulo-based automatic threaded reads cannot occur if either:

- No column in the table is eligible for modulo arithmetic
- All table columns eligible for modulo arithmetic are referenced in a WHERE clause

For more information about auto-partitioning, see the online documentation SAS/ACCESS Relational Databases/Threaded Reads/Auto-partitioning Techniques in SAS/ACCESS at: <http://support.sas.com/onlinedoc/913/getDoc/en/acreldb.hlp/a002235529.htm>

**Determining Whether a Threaded Read Occurred**

Before running your code, turn on SAS/ACCESS tracing before the step where you expect the threaded read to occur. The SASTRACE output will help you determine whether a threaded read actually was performed. To turn on SASTRACE, specify:

```
options sastrace=",,t,d" sastraceloc=saslog;
```

For more information, see the “**SASTRACE Threaded Read trace**” section of this paper.

If a threaded read has occurred, the SAS log will show a line for each thread generated, along with the number of rows that the thread has retrieved. The following log examples show threaded reads occurred on two databases.

**Example 1: an Oracle threaded read**

```
ORACLE: Thread 1 contains 500 obs.
ORACLE: Thread 2 contains 500 obs.
```

### Example 2: a Teradata threaded read

```
TERADATA: Thread 1 contains 143213 obs.
```

```
TERADATA: Thread 2 contains 104699 obs.
```

When a SAS/ACCESS engine generates modulo arithmetic WHERE clauses to autopartition the database data, the trace output will contain SQL statements that are identical, except for the WHERE clause, as is seen in the following example:

Example: an Oracle threaded read that shows WHERE clauses with a modulo arithmetic on a table column

```
ORACLE_2: Executed:
```

```
SELECT "I", "J" FROM X WHERE ABS(MOD("I",2))=0
```

```
ORACLE_3: Executed:
```

```
SELECT "I", "J" FROM X WHERE (ABS(MOD("I",2))=1 OR "I" IS NULL)
```

### Assessing Threaded Read Performance

You should run a SAS job multiple times—turning threaded reads on and off—to assess performance of a threaded read. After making the runs, compare the elapsed time for each run reported in the SAS log. The following are some best practice recommendations to ensure your comparisons are accurate:

- Turn on SASTRACE SQL tracing and timers before running comparisons so you can obtain performance information.
- For examples of tracing and timing, as well as what you should look for, see SAS/ACCESS online documentation, Threaded Reads/Generating Trace information for Threaded Reads <http://support.sas.com/onlinedoc/913/getDoc/en/acreldb.hp/a002272145.htm>
- When possible, send your output to the \_NULL dataset to isolate the read performance. Isolating the read performance can help determine whether the read might be the bottleneck of the overall throughput.
- Run comparisons when your systems and network aren't saturated; that is, avoid peak usage periods. Also test in a busy situation.
- Perform multiple runs and then average the results.
- If the read does not appear to be the source of the problem, examine the write portion.

The following example evaluates a threaded read of an Oracle table. The SASTRACE output shows that modulo arithmetic slicing was used to auto-partition the Oracle table data. As stated earlier, modulo-arithmetic auto-partitioning is used with Oracle only when the table isn't physically partitioned in the database. An Oracle table that is not physically partitioned is a good comparison candidate: to see if a threaded read performs better than a standard read.

Our example SASTRACE output is for a single, threaded read run. In general, you would perform several runs before making a conclusive performance assessment of how well the reads perform.

```
45 option dbsliceparm=all;
```



```

46  data _null_;
47  set x.financial_impact(keep=CONTACT_NM CONTACT_DETAILS DISCOVERY_DT
48                          FINANCIAL_IMPACT_ID IMPACT_DESC
49                          REFERENCE_NO THRESHOLD );
50  run;

```

```

ORACLE_12: Executed:
SELECT  "IMPACT_DESC", "REFERENCE_NO", "DISCOVERY_DT", "CONTACT_NM",
"CONTACT_DETAILS",
"THRESHOLD", "FINANCIAL_IMPACT_ID" FROM FINANCIAL_IMPACT  WHERE
ABS(MOD("FINANCIAL_IMPACT_RK",2))=0

```

```

ORACLE_13: Executed:
SELECT  "IMPACT_DESC", "REFERENCE_NO", "DISCOVERY_DT", "CONTACT_NM",
"CONTACT_DETAILS",
"THRESHOLD", "FINANCIAL_IMPACT_ID" FROM FINANCIAL_IMPACT  WHERE
ABS(MOD("FINANCIAL_IMPACT_RK",2))=1

```

```

ORACLE:  Thread 1 contains 532753 obs.
ORACLE:  Thread 2 contains 532747 obs.
ORACLE:  Threaded read enabled. Number of threads created: 2
NOTE: There were 1065500 observations read from the data set X.FINANCIAL_IMPACT.
NOTE: DATA statement used (Total process time):
      real time          53.68 seconds
      cpu time           46.00 seconds

```

```

103 option dbsliceparm=none;
104 data _null_;
105 set x.financial_impact(keep=CONTACT_NM CONTACT_DETAILS DISCOVERY_DT
106                          FINANCIAL_IMPACT_ID IMPACT_DESC
107                          REFERENCE_NO THRESHOLD );
108 run;

```

```

NOTE: There were 1065500 observations read from the data set X.FINANCIAL_IMPACT.
NOTE: DATA statement used (Total process time):
      real time          38.99 seconds
      cpu time           20.37 seconds

```

Compare the times in the example, shown in bold, to see that a threaded read can reduce performance.

### Optimizing Threaded Reads

This section presents several ways that you can optimize performance of threaded reads:

- DB2 Specific
- Oracle Specific
- Teradata Specific
- Increase the read buffer size with the READBUFF=option
- Specify NOEQUALS processing for SAS sort processing
- Ensure retrieved rows are distributed evenly among the read threads
- Experiment with the number of read threads

**DB2-Specific**

If the DB2 table is physically partitioned, use the DBSLICE= option with the DB2-specific function NODENUMBER to read the table.

Example: DB2 table is physically partitioned on a single machine

```
proc print data=trlib2.MYEMPS(DBSLICE=
("nodenumber(EMPNO) = 0"
"nodenumber(EMPNO)=1"
"nodenumber(EMPNO)=2) );
run;
```

In our example, a threaded read can reduce the read time for the DB2 table because:

- The table data, physically partitioned on column EMPNO; is distributed evenly across three partitions;
- The DB2 instance is running on a single machine.

If the DB2 database is clustered across multiple machines, a more complicated syntax might be required to optimize the read. For more information, see the **Configuring DB2 EEE Nodes on Physically Partitioned Databases** topic in online documentation: SAS/ACCESS for DB2 under UNIX and PC Hosts/Auto-partitioning Scheme for DB2 under UNIX and PC Hosts:

<http://support.sas.com/91doc/getDoc/acreldb.hlp/a002256107.htm>

**Oracle Specific**

If the Oracle table is physically partitioned, allow the SAS/ACCESS engine to handle the details of the threaded read. If the SASTRACE output shows that SAS is generating modulo arithmetic threads, the Oracle table isn't partitioned.

**Teradata-Specific**

Verify that Teradata's FastExport utility is available to obtain the best performance.

**Increase the Read Buffer Size with the READBUFF=Option**

You might get enough performance from altering the size of the read buffer that threaded reads are not necessary. As explained earlier in the paper, the READBUFF= option sizes a database's read buffers. Increasing this buffer for both Oracle and DB2 might expedite threaded reads. This is particularly true for DB2, when you specify the DBSLICE= option.

**Specify NOEQUALS Processing for SAS Sort Processing**

A SAS sort, especially involving threaded reads, might be faster than a database sort. SAS 9 includes a rewritten SORT that incorporates threading and data latency reduction algorithms. When SAS is running on a robust machine (rich in resources), a SAS sort might outperform the database sort. We suggest that you run your code with and without a SAS sort, and with and without threaded reads on each, to see which of the four performs better.

To specify SAS sorting, set SORTPGM=SAS with PROC SORT. This specification makes SAS omit the ORDER BY from the generated SQL, read the database data, and sort the data in SAS.

But, the database read is not threaded unless you also specify the NOEQUALS option with PROC SORT. (The NOEQUALS option allows SAS to sort the data randomly; that is, to ignore the relative order of observations within the BY groups.)

In the following example, we specify SORTPGM=SAS to have SAS sort the database data. We add the NOEQUALS option to generate threaded reads.

```
Option sortpgm=sas;
proc sort noequals
data=ora.table out=local_dataset;
by i;
run;
```

### ***Ensure that Retrieved Rows are distributed evenly among the Read Threads***

A threaded read retrieves rows in parallel on two or more connections. But, when there's a severe imbalance in the number of rows retrieved by the threads, you might not see the boost in performance expected. The reason is a thread that has little or no data to deliver back to SAS becomes inactive, and therefore consumes unused resources.

The imbalance is most likely to occur when there's modulo slicing of the database data. The problem does not occur for Teradata when the Teradata FastExport utility is used. It also does not occur for Oracle when the table data is physically well partitioned in the database.

To determine a thread imbalance, turn on tracing before the threaded read; then check the trace output. Because SASTRACE= shows the number of rows retrieved per thread; a significant thread imbalance will be obvious. If you see an imbalance, you can use the DBSLICE= option to specify your own read partitions of the database data.

### ***Experiment with the Number of Read Threads***

The SAS/ACCESS products strictly limit the number of database read threads. The default for SAS/ACCESS Interface to Sybase is 3; the default for all other SAS/ACCESS products is 2.

You can test increasing the default to see whether increasing the number of threads will give better or worse read performance (the overhead of additional threads can degrade performance.) You might need to make several runs to find the break-even point where the overhead of adding additional threads offsets any potential boost in read performance.

To increase the limit, specify the desired number of threads in the second parameter of the DBSLICEPARM= option. Be aware that SAS 'I/O threaded' procedures, such as PROC REG and PROC SORT, might limit your read threads to the value set with the CPUCOUNT= option. The CPUCOUNT= option specifies the number of processors that the thread-enabled applications should assume will be available for concurrent processing. The default is the actual number of processors on the machine.

Example: Using the DBSLICEPARM= option to increase read threads.

In the following example for Oracle, we increase our thread limit to 4 by setting the second parameter of the DBSLICEPARM= to that number. Because our code is run on a 4-processor system, the CPUCOUNT= value is 4. Consequently, SAS will not reduce the threads that are available for our step to

less than 4 threads.

You should run the code at least twice: first, using the engine's default number; second, with the increased thread number. Only after comparing the SAS TRACE output, and elapsed time of both runs, can we determine first, whether the additional threads were actually created; and, second, if they were, did the additional threads enhance performance of this read operation.

```
26 Option sortpgm=sas dbsliceparm=(ALL,4);
27 proc sort noequals
28 data=x.x out=_null_;
29 By I;
30 quit;
```

```
ORACLE: Thread 1 contains 4990 obs.
ORACLE: Thread 2 contains 4990 obs.
ORACLE: Thread 1 contains 4990 obs.
ORACLE: Thread 2 contains 5040 obs.
ORACLE: Threaded read enabled. Number of threads created: 4
NOTE: There were 20010 observations read from the data set X.X.
NOTE: DATA statement used (Total process time):
```

```
ORACLE_18: Executed:
SELECT "I", "J" FROM X WHERE ABS(MOD("I",4))=0
```

```
ORACLE_19: Executed:
SELECT "I", "J" FROM X WHERE ABS(MOD("I",4))=1
```

```
ORACLE_20: Executed:
SELECT "I", "J" FROM X WHERE ABS(MOD("I",4))=2
```

```
ORACLE_21: Executed:
SELECT "I", "J" FROM X WHERE (ABS(MOD("I",4))=3 OR "I" IS NULL)
```

```
NOTE: PROCEDURE SORT used (Total process time):
```

```
real time          5.25 seconds
cpu time           2.64 seconds
```

The preceding example generates Oracle auto-partitioning with modulo-arithmetic on a table column. For more guidelines on how to assess threading performance, see the online documentation "Threaded Reads/ Performance Impact of Threaded Reads"

(<http://support.sas.com/91doc/getDoc/acreldb.hlp/a002230267.htm>)

## Understanding the SAS/ACCESS Database Connection Model

---

SAS/ACCESS products require a physical attach (connection) to an underlying database to operate on database data. Connection requirements can vary across different database systems. SAS/ACCESS products have developed a connection model to accommodate the variety of database connection requirements, and to effectively manage work performed in SAS.

When you execute a SAS/ACCESS LIBNAME statement, a connection is made to the database. Multiple connections can be made to a database for a single SAS/ACCESS LIBNAME statement depending upon the nature of the work being performed. SAS/ACCESS products request separate database connections for any non-read-only operation, such as Insert or Update, that modify or insert data. When the Update or

Insert operation has completed, the associated database connection is released.

Because database connections can be resource-intensive, consuming resources in SAS and on the database, SAS/ACCESS allows sharing of database connections when possible.

SAS/ACCESS products normally acquire distinct database connections for:

- Query operations (read-only, operations can share connections)
- Individual Update or Insert operations (require unique connections)
- Requests for database system catalog information (require unique connections)

SAS/ACCESS will attempt to use the same database connection across read-only operations performed across a single LIBNAME statement. If multiple SAS/ACCESS LIBNAME statements are executed, by default a database connection is made for each libref.

The following sections describe when SAS/ACCESS products acquire database connections, as well as the SAS/ACCESS options that influence the connection behavior.

### **Why Update and Insert Processes on the Database are Segregated**

Keeping Input-only operations separate from Update or Insert operations results in creating separate database connections. SAS/ACCESS normally separates these processes in order to manipulate database data more efficiently.

SAS/ACCESS by default attempts to isolate all read-only operations on separate database connections from tasks that might issue frequent SQL COMMIT statements. This is done because COMMIT statements can close cursors used to fetch data. SQL COMMIT statements are issued for Update and Insert operations to commit the changes, and to release locks held on database resources. Tables opened for Input operations generally fetch from database cursors, and therefore do not require frequent commit points. Keeping these processes on separate database connections avoids loss of cursor position when reading data. Loss of cursor position when reading data requires repositioning of the cursor which results in decreased performance, and can lead to data integrity issues.

### **SAS/ACCESS Utility Database Connection**

Some SAS application requests require information to be read from database system catalogs. For example, the DATASETS procedure requesting a member list of a SAS/ACCESS library could result in queries directly against the database system catalog. Queries referencing database system catalogs can cause the database to establish read locks on these system resources. These locks must be released quickly to minimize the impact on other database users.

SAS/ACCESS products use a special database connection, called a SAS/ACCESS Utility connection in order to accommodate requests for database system catalog information. By using a separate Utility connection to access database catalog information, SAS/ACCESS products can quickly release locks held on these system resources without delaying other processing.

By default the SAS/ACCESS Utility connection is not released until the SAS session completes or the LIBNAME reference is cleared. The SAS/ACCESS Utility connection would be reused for subsequent requests.

## Using SAS/ACCESS LIBNAME Options to Manage Database Connections

The default SAS/ACCESS database connection behavior usually provides ideal separation between read-only tasks and tasks that frequently commit work. But this default behavior might not be ideal for all situations.

SAS/ACCESS products provide LIBNAME options to allow the connection behavior to be varied for specific situations. Not all SAS/ACCESS products support all the options or settings, and the defaults for these products might vary. To determine whether a connection option applies to a specific SAS/ACCESS product, check the online documentation for that product.

This following table presents an overview of the primary SAS/ACCESS options that influence when database connections are made and the degree they are shared.

LIBNAME Option	Value	Applies to single or multiple librefs	Function or Recommended Purpose
CONNECTION=	SHAREDREAD	Single	Segregates Input operations from Update and Insert operations on a single libname statement.
	UNIQUE	Single	No sharing of database connections. Separate database connections are made for each.
	SHARED	Single	Shares a single database connection for all operations (Input, Update, and Insert) performed using a single libname statement. This setting is used to avoiding potential deadlock situations when one database table is used to populate another database table residing in the same database space.
	GLOBALREAD	Multiple	Shares a single database connection for read operations across librefs that share the same physical connection properties. Separate connections are made for update and insert.
	GLOBAL	Multiple	Shares a single database connection for all operations (Input, Update, and Insert). Connections are shared across librefs that share the same physical connection properties.
CONNECTION GROUP=		Multiple	Input operations are shared across librefs that specify the same connection group name.
DEFER=	NO	Single	Causes a database connection to occur when the LIBNAME statement is executed.
	YES	Single	Delays the database connection until the first request for database data.
UTILCONN_TRANSIENT=	NO		Utility connection is maintained until the Libname statement is cleared.

LIBNAME Option	Value	Applies to single or multiple librefs	Function or Recommended Purpose
	YES		Utility connection is released after each use.



**Note:** These option settings do not control threaded read operations. Threaded read operations can cause multiple read-only connections to be issued to the database.

### Options That Affect SAS/ACCESS Utility Connections

As mentioned, many SAS/ACCESS engines support a Utility connection enabling the engine to issue COMMIT statements to release database locks on database system resources without affecting other processing. Invoking SAS procedures such as DATASETS and CONTENTS can cause the utility connection to be made. SAS/ACCESS connection options generally do not affect Utility connections. However, the following SAS/ACCESS options affect Utility connections:

#### **DEFER=**

The DEFER= option affects when a SAS/ACCESS product acquires an initial database connection for the libref. The default of DEFER=NO indicates to acquire a database connection when the LIBNAME statement is executed. Specification of DEFER=YES indicates that the database connection should not be made until the first request for database data is made. The DEFER=option setting applies to both data and utility connections.

#### **UTILCONN\_TRANSIENT=**

Database connections for data access and SAS/ACCESS Utility connections can be made for each SAS/ACCESS LIBNAME statement executed. Use of multiple SAS/ACCESS LIBNAME statements can result in a large number of database connections. Specifying SAS/ACCESS option UTILCONN\_TRANSIENT=YES will cause the SAS/ACCESS Utility connection to be released after it has been used. As a result, Utility connections are acquired and released as needed, instead of persisting across tasks. Using this option setting can help reduce the number of concurrent connections made to the database.

### Links for More Information

For detailed information about the connection options, see the SAS online documentation, *SAS/ACCESS: Relational Databases/ The LIBNAME Statement for Relational Databases:*

CONNECTION=

<http://support.sas.com/onlinedoc/913/getDoc/en/acreldb.hlp/a001342247.htm>

CONNECTION\_GROUP=

<http://support.sas.com/onlinedoc/913/getDoc/en/acreldb.hlp/a001342257.htm>

DEFER=

<http://support.sas.com/onlinedoc/913/getDoc/en/acreldb.hlp/a001342309.htm>

UTILCONN\_TRANSIENT=

<http://support.sas.com/onlinedoc/913/getDoc/en/acreldb.hlp/a001342402.htm>



---

## Contributing Experts

---

Nancy Wills

Richard Smith

Doug Sedlak

Dave Berger

Margaret Crevar

Donna DeCapite

Kerrie Brannock

Ray Michie

Nancy Rausch

Donna Zeringue

Lewis Church

John Crouch