

Using the SAS/ACCESS® Libname Technology to Get Improvements In Performance and Optimizations in SAS/SQL Queries

Fred Levine, SAS® Institute, Cary NC

ABSTRACT

This paper highlights the new features in the SAS/ACCESS libname engines that, when used judiciously, can improve overall engine scalability in the areas of loading/extraction, ASYNC I/O, and SQL-based query optimizations.

The new loading/extraction engine features are:

- Multi-row reads
- DBKEY
- Bulk loading
- Multi-row writes

The new ASYNC I/O features are:

- PreFetch
- SAS server task switching

The new SQL-based query optimization features are:

- Implicit SQL-Passthru
- WHERE clause optimizations

Some of the above features are new in 8.2 whereas others existed in prior releases but have been enhanced in 8.2.

INTRODUCTION

All of the SAS/ACCESS libname features discussed in this paper are designed to improve scalability in specific processing environments that warrant such improvements. The more knowledge you have of your underlying data and how your DBMS server and network are tuned to process that data, the more knowledge you will have to be able to make wise decisions about when to use these features to your advantage.

The examples used in this paper come from various SAS/ACCESS libname engines. Not all features have been implemented in all SAS/ACCESS engines. Please see the SAS/ACCESS documentation for engine-specific information where applicable.

Loading/extraction features

The features described below are designed to improve engine performance in the area of loading and extracting data. They are implemented as libname and/or dataset options and are easy to use.

MULTI-ROW READS

The ability to extract your data as fast as possible is of paramount importance. SAS/ACCESS libname engines accomplish this internally by exploiting native API-controlled multi-row read capabilities of the underlying DBMS. To activate this feature, you specify the number of rows returned in a single read operation via libname and/or dataset options. The following example uses SAS/ACCESS to Oracle to specify 1000 rows per read operation using the READBUFF dataset option:

```
libname myora oracle user=scott pw=tiger  
readbuff=1000;
```

All tables from the schema defined by the above libname statement will be read in 1000 row blocks. This feature can improve performance for very large tables when your DBMS server and network are optimally tuned (it should be noted that setting the TCP Packet size on your network can make a big difference in performance.)

The SAS/ACCESS libname engines that support API-controlled multi-row reads are Oracle, Oracle Rdb, DB2/Unix, Sybase, ODBC, and OLE/DB. Please note that not all DBMSs support API-controlled multi-row reads although they often provide this feature transparently. For further engine-specific information about multi-row reads, please see the SAS/ACCESS documentation.

DBKEY

The DBKEY data set option is used to improve performance when joining a very small SAS data set to a large DBMS table. Please note that if this feature is not used prudently it will not improve performance, and in some cases can actually degrade performance. The following conditions must be met to see performance improvements with DBKEY:

- a very small transaction SAS data set
- a large master DBMS table

In addition, performance is often improved when DBKEY is used in conjunction with the DBNULLKEYS data set option (see the DBNULLKEYS section below for details).

HOW DBKEY WORKS

When you join a large DBMS table and a small SAS data set, the DBKEY option enables you to retrieve only the required subset of the DBMS data into SAS for the join. If you do not specify this option, SAS reads the entire DBMS data into SAS and then processes the join.

Using DBKEY can decrease performance when the SAS data set is too large. This is because DBKEY causes each value in the SAS transaction data set to generate a new result set (or open cursor) from the DBMS table. For example, if your SAS data set has 100 rows with unique key values, you request 100 result sets from the DBMS which can be very expensive. You must determine whether using this option is appropriate, or whether you can achieve better performance by reading the entire DBMS table (or creating a subset of that table).

Internally, the SAS/ACCESS libname engine generates a WHERE clause of the form:

where column = host-variable

Note that 'column' is what was specified on the DBKEY= option. The *host-variable* takes the value of 'column' from each row in the SAS data set and generates a separate DBMS result set for each of these values. As a result, only rows in the DBMS table that match this WHERE clause are retrieved. Without DBKEY, the above WHERE clause would not get internally generated

forcing SAS to read all the rows from the DBMS table before processing the join.

USING THE DBNULLKEYS OPTION WITH DBKEY

The DBNULLKEYS data set option is used in conjunction with DBKEY and has a direct effect on the internally generated WHERE clause described above. If there is NULL data in your DBMS table, then the generated WHERE clause will account for NULLs as follows:

```
where ((column = host-variable)
      OR ((column IS NULL) AND (? IS NULL)))
```

This WHERE clause generated the extra NULL conditions in response to DBNULLKEYS=YES which tells the SAS/ACCESS libname engine that the DBKEY column from the DBMS table contains NULL data. However, when your DBKEY column from the DBMS table does not contain NULL data, you should specify DBNULLKEYS=NO which causes the SAS/ACCESS libname engine to generate a simpler form of the WHERE clause without the check for NULLs:

```
where ((column = host-variable)
```

It should be noted that when DBNULLKEYS=YES, the more complicated WHERE clause that checks for NULLs has the potential to be much less efficient than the simpler form of the WHERE clause. This is why DBKEY works best for DBMS tables that do not contain NULL data. Consider the following example:

```
/* SMALL SAS DATA SET */
5? data saslib.small;
6?  do id=1 to 1000000 by 100000;
    smalltext='Fra small'; output; end;
    run;

7?
NOTE: The data set SASLIB.SMALL has 10
      observations and 2 variables.

/* LARGE ORACLE TABLE */
33? data oralib.master;
34?  do id=1 to 200000;
    text='Master table'; output; end; run;
35?
NOTE: The data set ORALIB.MASTER has 200000
      observations and 2 variables.
```

Neither the SAS table nor the Oracle table contain NULL data.

The following data step join uses DBKEY with DBNULLKEYS=YES:

```
11? data temp;
    set saslib.small;
    set oralib.master (dbkey=id
dbnullkeys=yes) key=dbkey ;
    run;

NOTE: There were 10 observations read from
the data set SASLIB.SMALL.
NOTE: There were 10 observations read from
the data set ORALIB.MASTER.
NOTE: The data set WORK.TEMP has 10
      observations and 3 variables.
NOTE: DATA statement used:
      real time          20.55 seconds
      cpu time           0.03 seconds
```

The following data step join uses DBKEY with DBNULLKEYS=NO:

```
38? data temp;
    set saslib.small;
    set oralib.master (dbkey=id dbnullkeys=no)
key=dbkey ; run;
40?
NOTE: There were 10 observations read from
the data set SASLIB.SMALL.
NOTE: There were 10 observations read from
the data set ORALIB.MASTER.
NOTE: The data set WORK.TEMP has 10
      observations and 3 variables.
NOTE: DATA statement used:
      real time          0.03 seconds
      cpu time           0.03 seconds
```

Notice the difference in real time. When DBNULLKEYS=YES the extra NULL conditions on the generated WHERE clause caused this job to take 20.55 seconds. When DBNULLKEYS=NO, yielding the simpler WHERE clause, the real time dropped to a mere .03 seconds.

The SAS/ACCESS libname engines will also check to see if a DBKEY column is NON-nullable, and if so, generate the simpler, more efficient WHERE clause regardless of the DBNULLKEYS value. However, if the DBKEY column was not originally declared as NON-nullable but you do not have any NULL data for that column, then you will need to set DBNULLKEYS=NO to reap the maximum performance benefits of DBKEY.

Please see the SAS/ACCESS documentation for further information about DBKEY.

BULK LOADING

The purpose of bulk loading is to provide the highest possible load performance utilizing native DBMS load utilities. The ability to exploit native load utilities has huge performance implications when loading a large data warehouse. The SAS/ACCESS libname engines allow you to easily invoke these native load extensions via libname and/or dataset options.

Below is an example of bulk load syntax using SAS/ACCESS to Teradata:

```
libname mytera teradata database=john
user=john pw=doe bulkload=yes;
```

The above syntax tells the Teradata engine to use the native FastLoad to insert rows into tables scoped to the connection defined by the above libname statement. There are many other options that are used in conjunction with BULKLOAD=YES. See the SAS/ACCESS documentation for further information on BULKLOAD options.

Note that DBMS-specific bulk load facilities are not transactional in that they do not use programmatic SQL insert statements to load data. Rather, they cause the data from the input data set to be bulk copied as a unit to the DBMS table. As a result, error conditions behave differently under bulk load, i.e., no rollbacks are issued.

In addition to bulk copying to empty DBMS tables, some bulk loaders also allow appending rows to existing DBMS tables.

We have compared loading a modest size 32,000 row SAS dataset into an Oracle table using both the native Oracle SQL*Loader and standard SQL inserts. We have observed that Oracle's SQL*Loader was 3 times faster than using conventional SQL inserts. If a modest size table can produce these

DATA WAREHOUSING

performance improvements, you could expect to see much greater performance gains with a very large table.

It should be noted that using a DBMS's native bulk loader can also impede performance for very small tables due to the processing overhead of setting up the loader. It is therefore recommended that bulk loading be reserved for larger tables. You may need to experiment using bulk load with different size tables to determine how many rows are required to yield meaningful performance gains.

The SAS/ACCESS engines that support native bulk loading are Oracle, DB2/Unix, DB2/MVS, Sybase, Teradata, ODBC, and OLE/DB.

Please see the SAS/ACCESS documentation for further information about bulk loading.

MULTI-ROW WRITES

Although native bulk loading provides the fastest possible load performance, it is also possible to get faster load performance using conventional transactional processing. SAS/ACCESS libname engines allow you to do this by exploiting native API-controlled multi-row write capabilities of the underlying DBMS. To activate this feature, you specify the number of rows to be inserted for a single write operation via libname and/or dataset options. These options allow you to insert multiple rows at a time by specifying the number of rows to be inserted as a unit. The following example uses SAS/ACCESS to DB2/Unix to specify 100 rows per write operation using the INSERTBUFF dataset option:

```
libname mydb2 db2 database=sample user=john
      using=doe insertbuff=100;
```

The above syntax instructs the DB2/Unix engine to insert 100 rows at a time when loading data into tables scoped to the connection defined by the above libname statement.

It should be noted that multi-row writes have an impact on error handling since errors are associated with buffers rather than with individual rows. Errors are therefore not discovered until a later point in the processing.

The optimal value for multi-row write options such as INSERTBUFF vary with factors such as network type and available memory. You may need to experiment with different values to determine the best value for your site.

The SAS/ACCESS libname engines that support API-controlled multi-row writes are Oracle, DB2/Unix, ODBC, and Oracle Rdb. Just like multi-row reads, not all DBMSs support API-controlled multi-row writes.

For further information about multi-row writes see the SAS/ACCESS documentation.

ASYNCH I/O features

ASYNCH I/O is the area of processing that allows SAS/ACCESS libname engines to exploit asynchronous execution of calls into the underlying DBMS for the purpose of improving performance and optimizing client requests in a SAS server environment. In general terms, asynchronous execution refers to events that are not coordinated in time, such as starting the next operation before the current one is completed.

PREFETCH

PreFetch is a SAS/ACCESS facility that can speed up a multi-

step SAS job by exploiting the asynchronous processing capabilities of an underlying DBMS. The SAS job must be read-only to use this facility, that is, any SAS statement that creates, updates, or deletes DBMS tables would not be a candidate for PreFetch. It should also be noted that at the time of this writing, PreFetch is only supported by SAS/ACCESS to Teradata.

When reading tables, SAS/ACCESS programmatically submits DBMS-specific SQL statements on your behalf to the DBMS. Each of these SQL statements has an execution cost. When PreFetch is enabled, the first time you run your SAS job SAS/ACCESS will identify those SQL statements with a high execution cost and store them in a DBMS-defined macro. On subsequent runs of the SAS job, SAS/ACCESS will extract the stored SQL statements from this macro and submit them in advance to the DBMS which will "prefetch" the rows selected by these stored SQL statements. It should be noted that PreFetch improves elapsed time only on subsequent runs of a SAS job since on the first run, SAS/ACCESS merely stores the selected SQL statements for subsequent use. For this reason, PreFetch should be used only for static SAS jobs that are run frequently.

Below is an example of using the SAS/ACCESS to Teradata PreFetch facility:

```
libname mytera teradata database=john
      user=john pw=doe prefetch='tr_store1';

proc print data=mytera.emp where emp.salary >
      100000;
proc print data=mytera.sales where
      sales.commission > 0;
proc print data=mytera.sales where
      sales.product = 'truck';
proc print data=mytera.newsales;run;
```

The first time you submit the above job SAS/ACCESS to Teradata will create a 'tr_store1' macro to store the SQL statements associated with the above proc prints if it is determined that they have a high execution cost.

For subsequent runs of this job, you need only specify:

```
libname mytera teradata database=john
      user=john pw=doe prefetch='tr_store1';
```

SAS/ACCESS to Teradata will now "prefetch" the rows associated with the stored SQL statements by utilizing the native asynchronous processing capabilities of Teradata. PreFetch can also be specified as a SAS global option.

For further information see the SAS/ACCESS to Teradata documentation.

SAS SERVER TASK SWITCHING

SAS server task switching is a mechanism designed to maximize multi-client throughput in a concurrent SAS server environment. This SAS server environment can either be a SAS/SHARE server or a SAS Integrated Object Model (IOM) server (please see SAS documentation for more information on SAS/SHARE® and IOM).

The basic idea behind task switching in a SAS server environment is that a SAS/ACCESS libname engine running on a SAS server must not impede response time for other clients while the engine waits for a lengthy DBMS operation to complete for a specific client. By default, when the engine is processing a request for a specific client other client requests are suspended since the SAS server does not implement time-slicing.

To get around this potential problem, SAS/ACCESS has

implemented a SAS invocation option 'DBSRVTP' that enables a SAS/ACCESS libname engine to voluntarily give up control of a client task to another client task for targeted DBMS operations in a SAS server environment. The primary benefit of this task-switching mechanism is to allow the engine in the SAS server to respond to many different clients more efficiently.

It should be noted that the individual DBMS operations that enable task-switching may vary from engine to engine due to DBMS-specific differences in the execution time of these operations.

Below is an example of invoking SAS in a server environment using the 'DBSRVTP' option to enable task-switching using SAS/ACCESS to Sybase:

```
sas -dbsrvtp sybase
```

Multiple engines can also be specified. The example below invokes SAS using the 'DBSRVTP' option to enable task-switching using SAS/ACCESS to Sybase, ODBC, and Informix:

```
sas -dbsrvtp '(sybase ODBC informix)'
```

The SAS/ACCESS libname engines that support SAS server task switching are DB2/Unix, Informix, ODBC, OLE/DB, Oracle, Sybase, and Teradata. Please see the SAS/ACCESS documentation for further information on the DBSRVTP option.

SQL-based query optimizations

SQL query optimizations is a performance improvement feature that transparently offloads SQL processing that normally would occur in SAS to the underlying DBMS. There are two contexts in which these transparent SQL optimizations occur:

- Proc SQL
- WHERE clauses surfaced from any SAS procedure

The facility that allows this to happen in proc SQL is called Implicit SQL Passthru. The facility that allows this to happen in SAS WHERE clauses is referred to as the WHERE optimizer.

IMPLICIT SQL PASSTHRU

Implicit SQL Passthru (hereafter referred to as Implicit Passthru) is a proc SQL feature that, for performance-sensitive SQL operations, will transparently convert your proc SQL query into a DBMS-specific SQL query and directly pass this converted query to the DBMS for processing. This mechanism has several advantages over traditional passthru:

1. You often get similar performance improvements to traditional passthru while having your queries seamlessly integrated into SAS.
2. If your site uses more than one SAS/ACCESS libname engine, you need only be familiar with SAS SQL syntax to get performance improvements for multiple DBMS engines.
3. Java and MFC-based thin client applications that submit generated SAS SQL to a SAS server (such as Enterprise Guide) will transparently get the performance benefits of Implicit Passthru on the SAS server without having to be cognizant of DBMS-specific SQL syntax.

For a proc SQL query to be a candidate for Implicit Passthru, it must reference a single SAS/ACCESS engine libref and contain one or more of the following:

- the DISTINCT keyword

- JOINS (inner and outer)
- SQL aggregate functions
- UNIONS

The above SQL operations have been targeted as key performance-sensitive operations that can often be processed faster by a DBMS.

It should also be noted that, beginning in release 8.2 of SAS (and beyond), SAS/ACCESS libname engines support a subset of SAS functions for which underlying DBMSs have equivalents. This means that supported SAS functions in proc SQL queries will get translated into their DBMS equivalents and will be passed along with the query to the DBMS.

Note that SAS functions do not by themselves trigger a pass-through. Instead, a query must be a candidate for pass-through as listed in the above bulleted list. Once the query is a candidate, then any supported SAS function for which the DBMS engine has an equivalent will simply be passed along with the rest of the query.

DISTINCT PROCESSING

The DISTINCT keyword triggers Implicit Passthru since in many cases the result set will be much smaller than the initial size of the table. By offloading this processing to a DBMS server, only the result set rows get transmitted across the network back to proc SQL, hence greatly reducing network time when the number of rows in the result set is much smaller than the number of rows in the table. Consider the following example using SAS/ACCESS to Ingres:

```
proc sql;
  libname ing ingres database=clifftop;
  select distinct state from ing.creditcard;
```

In the above example, a user wants to determine how many states are represented by a group of credit card customers. The result set could have a maximum of 50 rows. If this table contained 5 million rows, then you can quickly see the advantage of passing this query to Ingres for processing. The advantage is that transmitting a maximum of 50 rows across the network is much faster than proc SQL having to read all 5 million rows into SAS and then do its own DISTINCT processing.

PASSING DOWN JOINS

JOINS are another SQL operation that can normally be processed more efficiently by the DBMS when the result sets are much smaller than the input tables. Since in SAS SQL it is possible to join as many as 32 tables, you can quickly see the benefit of letting the DBMS do the processing since the performance cost of transmitting all rows from all join tables into SAS for processing can be formidable. Implicit Passthru can pass both INNER and OUTER joins to the DBMS for processing.

INNER JOINS

Passing INNER join queries to a DBMS is straight forward since all datasources support ANSI 1992 INNER join syntax.

The following INNER join query uses SAS/ACCESS to Informix and will get passed to Informix for processing:

```
proc sql;
  libname nfx informix user=john pw=doe
    database=rockbridge server=server1;

  select * from nfx.cust, nfx.sales,
    nfx.orders where cust.custnum
```

DATA WAREHOUSING

```
= orders.ordernum and  
sales.salesrep = 'SMITH';
```

OUTER JOINS

Passing down OUTER join queries is more complicated than passing down INNER joins since some datasources do not support ANSI 1992 OUTER join syntax. The datasources that have non-standard OUTER join syntax are Oracle, Sybase, Informix, and ODBC.

The Implicit Passthru facility has the capability to pass down OUTER join queries for all SAS/ACCESS libname engines, regardless of whether they support ANSI 1992 OUTER join syntax. For ANSI-compliant data sources, passing down OUTER join queries is straight forward with no restrictions. For those engines whose underlying datasources support non-standard OUTER join syntax, Implicit Passthru will convert SAS SQL ANSI-compliant OUTER join syntax to the non-standard datasource-specific OUTER join syntax, although with some restrictions (see below).

The following OUTER join query exemplifies this conversion to non-standard syntax using SAS/ACCESS to Oracle:

```
proc sql;  
  select * from eng.JOIN11 left join  
    eng.JOIN22 on JOIN11.x=JOIN22.x right join  
    eng.JOIN33 on JOIN11.x=JOIN33.x;
```

Implicit Passthru will convert the above SAS ANSI-compliant OUTER join text to:

```
select JOIN11."X", JOIN22."X", JOIN33."X"  
  from JOIN11, JOIN22, JOIN33  
  where JOIN11."X" (+) = JOIN33."X"  
    and JOIN11."X" = JOIN22."X" (+)
```

in keeping with Oracle-specific OUTER join syntax which uses the '+' operator in the WHERE clause to tag non-preserved tables in OUTER join queries.

RESTRICTIONS ON NON-ANSI OUTER JOIN SYNTAX

Although Implicit Passthru can generate non-standard datasource-specific OUTER join syntax for those datasources that require it (Oracle, Sybase, Informix, and ODBC), there are some restrictions.

1. For queries that use SAS/ACCESS to Sybase, any OUTER join that references more than two tables AND contains a WHERE clause will not get passed. *Note that more than two table OUTER joins will get passed without a WHERE clause, as will a two table OUTER join with a WHERE clause.* This restriction exists since Sybase can return different results than SAS when a WHERE clause is applied to an OUTER join with more than two tables.
2. For queries that use SAS/ACCESS to Informix, only two table OUTER joins will get passed. Informix uses a WHERE clause in lieu of an ON clause which makes the integration of ON clauses and WHERE clauses from a SAS SQL query difficult to convert into an Informix-specific query.
3. For queries that use SAS/ACCESS to ODBC, more than two table OUTER joins are supported as long as there are no INNER joins specified in that same query. This restriction exists due to limitations in ODBC OUTER join syntax.
4. Oracle, Sybase, and Informix do not support FULL OUTER joins.

It should be noted that in spite of the restrictions listed above, in many cases these restrictions will not be an issue and you should often be able to reap performance benefits from these non-standard datasources.

SQL AGGREGATE FUNCTIONS

SQL aggregate functions represent another area of SQL operations that are processed more efficiently by a DBMS. This is because typically, they search a table and perform behind-the-scenes calculations, yielding a single row result set. These SQL aggregate functions include:

- MIN
- MAX
- AVG
- SUM
- COUNT

Below is an example of passing down a query that contains an SQL aggregate function using SAS/ACCESS to Oracle:

```
libname ora oracle user=john pw=doe;  
proc sql;  
  select count(*) from ora.employees;
```

UNIONS

UNIONS will also get passed to the DBMS. Since typically a UNION eliminates duplicate rows, this processing is more efficient when passed to a DBMS.

Below is an example of a UNION using SAS/ACCESS to DB2/Unix:

```
libname mydb2 db2 database=sample user=john  
  using=doe;  
proc sql;  
  select * from mydb2.music_titles  
  union  
  select * from mydb2.discontinued_CDs;
```

PARTS OF A QUERY CAN GET PASSED DOWN

In the event that a query that gets passed down to a DBMS results in a failure returned from the SAS/ACCESS engine, proc SQL will attempt to pass down a simpler version of that query. Any portions of the query that cannot be handled by the DBMS are handled by proc SQL [Church 1999].

Below is an example of a part of a query getting passed down using SAS/ACCESS to Sybase. It was mentioned above that there is a restriction on Sybase queries where more than two tables in an OUTER join cannot be specified with a WHERE clause due to differences in the way Sybase evaluates these queries. The query below contains a three table OUTER join with a WHERE clause:

```
libname syb sybase user=john pw=doe;  
proc sql;  
  select emplinfo.department, emplinfo.lastname  
    from syb.employees left join syb.emplinfo  
      on employees.empid=emplinfo.employee  
    left join syb.dept  
      on dept.deptno=emplinfo.department  
  where employees.empid > 100;
```

Based on the above query, Implicit Passthru will generate the following to pass down to Sybase:

```
select emplinfo.department, emplinfo.lastname
  from employees, emplinfo
 where employees.EMPID *= emplinfo.employee
    and (employees.EMPID > 100)
```

Note that Implicit Passthru generates Sybase-specific OUTER join syntax which uses the '*' operator to tag preserved tables in OUTER join queries.

In this example the first two tables in the OUTER join along with the WHERE clause gets passed to Sybase since there is a restriction of specifying more than two tables in an OUTER join with a WHERE clause. After the result set of this query is returned from Sybase, proc SQL processes the remaining join of the 'dept' table from the original SAS query. So you can still get performance benefits even when pass-down restrictions exist for the non-standard datasources.

PASSING DOWN SAS FUNCTIONS

It was mentioned earlier that, starting in release 8.2 of SAS, SAS/ACCESS libname engines support passing down a subset of SAS functions for which the underlying datasource has equivalents. Although the presence of SAS functions in proc SQL queries do not trigger a pass-down by themselves, they will get converted and passed down as part of queries that meet the aforementioned Implicit Passthru criteria.

Below is an example of a query using SAS/ACCESS to Oracle that contains the SAS function 'UPCASE':

```
proc sql;
  select distinct UPCASE(joinchar.name) from
    ora.joinchar;
```

This query meets the Implicit Passthru criterion of DISTINCT processing and is therefore a candidate for getting passed down. It also contains the SAS function 'UPCASE'. Since SAS/ACCESS to Oracle supports Oracle's equivalent function 'UPPER', the SAS function 'UPCASE' gets converted to its Oracle equivalent and gets passed along with the rest of the query. Below is the generated query that gets passed to Oracle:

```
select distinct UPPER(joinchar."NAME")
  from JOINCHAR
```

If a SAS function that is not supported by the DBMS engine appears in the query, then the query (or possibly just the part of the query that contains the SAS function) will not get passed.

It should be noted that the subset of SAS functions supported in SAS/ACCESS libname engines varies for each engine. Please see the SAS/ACCESS documentation for information about which SAS functions are supported for a specific engine.

It should also be noted that the current list of supported SAS functions will potentially be expanded in subsequent releases of the SAS system.

WHAT DISQUALIFIES A QUERY FROM GETTING PASSED?

Any query that contains more than one SAS/ACCESS libref will be disqualified since different librefs may refer to different DBMS connections. For example, two different connections from the same SAS/ACCESS libname engine could possibly point to two different servers, hence precluding the passing of JOINS.

In addition, any query that contains one or more of the following will be disqualified [Church 1999]:

- data set options
- the INTO clause
- the COALESCE function
- remerging
- SAS functions (prior to 8.2)

The above constructs are disqualified because they are either SAS-specific or non-standard in an underlying DBMS.

For example, the COALESCE function is not supported in all DBMSs.

Data set options are generally too SAS-specific to be usefully converted, so they also preclude a pass-down. This includes the DBCONDITION option which does not get processed for Implicit Passthru.

As discussed above, the SAS function restriction has been lifted for 8.2 since SAS functions often have DBMS equivalents.

WHERE CLAUSE OPTIMIZATIONS

The WHERE clause optimizer is the facility that passes down SAS WHERE clauses. This facility has been in SAS/ACCESS products since V6. SAS WHERE clauses can be surfaced in any SAS procedure that operates on rectangular data, i.e.,

```
libname ing ingres database=musicalia;
proc print data=ing.orders;
  where dateordered > '01jan1996'd;run;
```

SAS/ACCESS engines internally generate programmatic DBMS SQL when accessing DBMS tables specified in SAS procs. The SAS/ACCESS WHERE processor will parse through the SAS WHERE clause, convert it to a DBMS-specific WHERE clause, and append the converted WHERE clause to the programmatic SQL statement. In the above example, the SAS DATE9. value gets converted into an Ingres-specific date value before getting passed to Ingres.

It should be noted that the SAS/ACCESS WHERE optimizer is a different facility than Implicit Passthru, that is, any WHERE clause that is part of an Implicit Passthru query gets processed by Implicit Passthru. If a proc SQL query is not a candidate for Implicit Passthru but nevertheless contains a WHERE clause, then that WHERE clause will get passed to the SAS/ACCESS WHERE optimizer. This is because in the NON-Implicit Passthru context proc SQL is just another SAS procedure.

SAS FUNCTIONS GET PASSED IN THE WHERE OPTIMIZER

Beginning in 8.2, SAS functions will also get passed down in the SAS WHERE clause (providing that the SAS/ACCESS engine supports a DBMS-equivalent just like in Implicit Passthru).

So now you can specify:

```
proc print data=ora.joinchar;
  where LOWCASE(name) = 'Alison';run;
```

and SAS/ACCESS to Oracle will generate the WHERE clause with the Oracle equivalent function 'LOWER'.

If a SAS function specified in the SAS WHERE clause is not supported by the SAS/ACCESS engine, then the SAS/ACCESS WHERE optimizer will return an error and SAS will evaluate the WHERE clause.

DATA WAREHOUSING

DBCONDITION CAN BE USED WITH THE WHERE OPTIMIZER

It should also be noted that the DBCONDITION dataset option (which lets you pass DBMS-specific SQL conditions to the DBMS) works in concert with the WHERE processor. That is, any DBCONDITION WHERE clause will be ANDed to the SAS WHERE clause. All other DBCONDITION subsetting will be appended to the SAS WHERE clause. Below is an example of using DBCONDITION with a SAS WHERE clause using SAS/ACCESS to Oracle:

```
proc print data=ora.join1 (dbcondition="order
by x1"); where x1 > 0; run;
```

The following is what gets passed to Oracle:

```
SELECT "X1" FROM JOIN1 WHERE ("X1" > 0 )
ORDER BY x1
```

HOW DO I KNOW MY QUERY IS GETTING PASSED DOWN?

It is easy for you to determine if your query has been passed to the underlying DBMS using the SASTRACE option. SASTRACE is a SAS system option that has SAS/ACCESS specific behavior. SASTRACE shows you the commands sent to your DBMS by the SAS/ACCESS engine.

The SASTRACE syntax used for SAS/ACCESS engines is:

```
SASTRACE = '.,.,d';
```

The '.,.,d' gives information about SAS/ACCESS engine calls to a relational DBMS. The following example shows how SASTRACE can be used to determine that an Implicit Passthru query got passed to the DBMS:

```
5? options sastrace='.,.,d';
6? proc sql;
7? select distinct * from ora.join1;

DEBUG: Open Cursor - CDA=2059746056 0
979854457 orusti 299 SQL
DEBUG: PREPARE SQL statement: 1 979854458
orprep 63 SQL
SELECT * FROM JOIN1 2 979854458 orprep 64
SQL
Prepare stmt: select distinct join1."X1"
from JOIN1 3 979854463 prepare 671 SQL
DEBUG: Open Cursor - CDA=2062403848 4
979854463 orusti 299 SQL
DEBUG: PREPARE SQL statement: 5 979854463
orprep 63 SQL
select distinct join1."X1" from JOIN1 6
979854463 orprep 64 SQL
SQL Implicit Passthru stmt prepared is: 7
979854463 ip_util 378 SQL
select distinct join1."X1" from JOIN1 8
979854463 ip_util 379 SQL
DEBUG: Close Cursor - CDA=2059746056 9
979854463 orustt 370 SQL
```

In the SASTRACE example above you can see all statements passed to the DBMS. Since individual tables in Implicit Passthru are separately prepared before the query of which they are a part get passed, we can see the programmatic SQL statement 'SELECT * FROM JOIN1' in the SASTRACE output. Note that SASTRACE also tells you that a prepared statement comes from

Implicit Passthru with the statement "SQL Implicit Passthru stmt prepared is:". So we can see from this example that the proc SQL query was passed to Oracle for processing.

For the WHERE processor, we would see the converted WHERE clause as part of the prepared statement if it got passed to the DBMS. Consider the following example:

```
16? proc print data=ora.joindate; where
dateval > '01jan1960'd; run;
```

```
DEBUG: Open Cursor - CDA=2058853128 21
979855721 orusti 299 PRINT
DEBUG: PREPARE SQL statement: 22 979855721
orprep 63 PRINT
SELECT * FROM JOINDATE 23 979855721 orprep
64 PRINT
DEBUG: PREPARE SQL statement: 24 979855721
orprep 63 PRINT
```

```
SELECT "DATEVAL" FROM JOINDATE WHERE
("DATEVAL"
>TO_DATE('01JAN1960','DDMONYYYY','NLS_DATE_LA
NGUAGE=American')) 25 979855721 orprep 64
PRINT
```

In this example, we first see the initial prepare of the table followed by the prepare of the same table that now includes an Oracle-specific WHERE clause that has converted the SAS date to an Oracle-specific date. We now know from the SASTRACE output that this WHERE clause was passed down to Oracle.

ENABLING AND DISABLING QUERY OPTIMIZATIONS

SAS/ACCESS gives you the ability to enable and disable a range of SQL-based query optimizations on a SAS/ACCESS libname statement. The libname option to do this is called DIRECT_SQL and specifies what types of generated SQL you wish to pass down to the datasource. By default, all SQL query optimizations are enabled.

Using this option, you can enable/disable the following types of generated SQL:

- Implicit Passthru
- SAS functions (in SAS WHERE clauses and Implicit Passthru)
- OUTER joins involving more than two tables
- WHERE clauses (both SAS WHERE clauses AND Implicit Passthru WHERE clauses)
- ALL of the above or any combination of the above

For example:

```
libname mydb2 db2 database=sample user=john
using=doe DIRECT_SQL=(NONE);
```

will disable all generated SQL, including Implicit Passthru, SAS functions, multi-table outer joins, and WHERE processing.

Another example:

```
libname mydb2 db2 database=sample user=john
using=doe DIRECT_SQL=(NOFUNCTIONS
NOMULTOUTJOINS);
```

will disable outer joins involving more than two tables and passing

down SAS functions in any context. This means that Implicit Passthru queries that do not contain SAS functions or more than two table outer joins will still get passed, as will WHERE processor queries that do not contain SAS functions.

WHY DISABLE THESE OPTIMIZATIONS?

There are several reasons why at times you may want to disable some or all of these optimizations:

1. NULL data is often processed differently in DBMSs than in SAS. If your DBMS data contains NULLS, there may be times when you need to disable Implicit Passthru and the SAS/ACCESS WHERE clause optimizer. This is because you can potentially get different results depending on whether SAS or the DBMS is doing the processing.
2. There are times when specifying complex OUTER joins with more than two tables can produce different results in SAS and an underlying DBMS. You may wish to disable just this feature without disabling other types of generated SQL. This is not a common occurrence, but it is possible.

For further information concerning 1. and 2., see the SAS/ACCESS white paper "*Potential Result Set Differences between Relational DBMSs and the SAS System*" on the SAS Data Warehousing Web page:

<http://sasprod.unx.sas.com/service/news/feature/15jan01/accessv8.html>

3. Some DBMS equivalents of SAS functions might produce slightly different results than SAS in very specific cases. Again, this is not common, but is possible.
4. In some cases, SAS can process the mathematical functions more efficiently than the DBMS equivalents.

For further information on the DIRECT_SQL libname option, please see the SAS/ACCESS documentation.

CONCLUSION

You have now seen some of the new performance features in the SAS/ACCESS libname engines that, when used judiciously, can improve overall engine scalability in the areas of loading/extraction, ASYNC I/O, and SQL-based query optimizations. Some of these features are already providing benefits for the SAS/ACCESS user community, and we will continue to focus on and improve scalability in our engines into the future.

We welcome your feedback in all areas of SAS/ACCESS development.

REFERENCES

Church, L. (1999). "Performance Enhancements to PROC SQL in Version 7 of the SAS System". *Proceedings of the twenty-fourth Annual SAS Users Group International Conference*, 24.

ACKNOWLEDGMENTS

Thanks to Doug Sedlak and Brian Hess for their technical contributions to the ASYNC I/O section of this paper.

Also thanks to Lewis Church who was my partner in implementing SQL Implicit Passthru. Much of the sophistication of Implicit Passthru can be credited to Lewis's SQL expertise.

CONTACT INFORMATION

Your comments and questions are valued and encouraged.

Contact the author at:

Fred Levine
SAS Institute Inc.
100 SAS Campus Drive
Cary, NC 27513
Work Phone: (919) 531-6826
Fax: (919) 677-4444
Email: fred.levine@sas.com

SAS is a registered trademark or trademark of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.