Paper 107-29

Improving Performance: Accessing DB2 Data with SAS 9

Scott Fadden, IBM, Portland, OR

ABSTRACT

This paper provides the DBA and SAS user with configuration details on how to obtain peak performance when running SAS applications with data from an IBM DB2 Universal Database for Linux, UNIX, and Windows data warehouse. We look at how SAS/ACCESS processes relational data and how to improve read, write, and load performance. This includes tuning tips and discussion of best practices in designing and tuning a data warehouse for use with SAS. The discussion is supported with test results to demonstrate the impact of the changes.

INTRODUCTION

Service level agreements, batch processing windows or breathing down your neck are all good reasons to improve the performance of your SAS/DB2 solution. With data growing faster than processor speed we can no longer rely on the latest hardware to solve a performance problem. Developing high performance solutions involves looking at all areas of data processing. This paper focuses on getting the best data access performance for a SAS application that accesses DB2 data. There are three main methods of improving solution performance.

- Reduce the data transferred
- Tune each processing step
- Parallelize where possible

When optimizing a solution using SAS and DB2 all three of these areas are very important. This paper provides examples in each of these areas and explores the impact of SAS and DB2 configuration options by outlining different methods of accessing your DB2 database. Examples are given to highlight the performance trade-offs of choosing different access methods, SAS 9 application parameters and DB2 8.1 configuration options It also highlights new features in SAS 9 and DB2 V8.1.

Throughout this paper, examples from a test system are used to demonstrate the performance impact of SAS/ACCESS tuning options. The test environment consists of SAS 9 and DB2 V8.1 running on a single four processor Unix system with 4GB of memory and 40 fibre channel disks.

The performance results in this paper demonstrate the impact of various configuration options. They are not intended as a method of comparing dissimilar configurations. Different configurations may yield different results (i.e., your mileage may vary).

WHAT IS SAS/ACCESS FOR DB2?

SAS and DB2 communicate via the SAS/ACCESS for DB2 product. SAS/ACCESS for DB2 is a member of a large family of data access products offered by SAS Institute. SAS/ACCESS allows the power and flexibility of SAS software to be used for analyzing and presenting data from a DB2 database. Your DB2 database tables appear native to the SAS software so you can use SAS features and functionality to perform extracts of information without having to learn Structured Query Language (SQL).

SAS/ACCESS for DB2 translates read and write requests from SAS into the appropriate calls for DB2. The result of these calls is to retrieve data in one of the following forms: as logical views of the DB2 database or extracts of data into SAS data set form.

SAS/ACCESS engine functionality ranges from "automatic" behind the scenes operations requiring minimal database knowledge to "completely flexible" allowing a DBA to finely tune the data access components of a SAS application. The method you use depends on your infrastructure, database expertise and operational goal. In this paper we examine the translation process from a SAS application to the corresponding SQL required to exchange information with DB2.

In the latest version of SAS/ACCESS for DB2 (SAS 9) and DB2 UDB (DB2 V8.1) the joint processing capabilities have been greatly expanded. These new capabilities include: SAS/ACCESS threaded reads and expanded load support; DB2 CLI LOAD and improved multi-row fetch performance.

WHAT SOFTWARE IS REQUIRED FOR SAS TO ACCESS MY DB2 DATABASE?

There are many configuration options that provide flexibility in designing a unique SAS/DB2 solution. SAS and DB2 may run on the same system or on different systems. Multiple SAS sessions can access a single database server or one SAS session can access multiple database servers. SAS and DB2 run on many different platforms and the platforms need not be the same for these components to interact. For example, your DB2 data warehouse could be running on AIX and your SAS session on Windows. The test environment used for examples in this paper has SAS and DB2 running on the same system.

Required software:

- DB2 V7.1 or higher
- The SAS:
  - Base SAS
  - SAS/Access for DB2

No modifications to your DB2 database are necessary for the addition of SAS/ACCESS. SAS/ACCESS communicates with the database server using the DB2 Call Level Interface (CLI) API included in the DB2 client software. Other SAS application packages that run on top of Base SAS can be added as needed but are not required for running basic SAS applications. However your environment is designed, there are a few basic software components that are required: DB2 V7.1 or higher, Base SAS and SAS/ACCESS for DB2.

ACCESSING DB2 DATA USING SAS

The previous sections defined SAS/Access for DB2 and what software is required. This section takes a more detailed look at the SAS/ACCESS interface for DB2.

To begin, you need to understand how SAS processes data.

Most SAS analysis procedures require that the input be from one or more preprocessed SAS data sets. Other procedures and `DATA` steps are designed to prepare the data for processing. As an example, take a look at the execution steps of a `print` procedure.

```
/*Step 1, Sort the SAS data set */
proc sort data=census.hrecs;
     by state;
run;

/* Step 2, Print the results */
proc print data=census.hrecs(keep=State serialno );
     by state;
run;
```

In this example we want to print the contents of the hrecs data set sorted by `state`. To achieve this using a SAS data set you run `proc sort` first to order the data set then `proc print` to produce the report. In this case using SAS/ACCESS to read directly from DB2 can make executing these procedures more efficient. In this example, if your data source were DB2 you would not need to pre-sort the data for `proc print`. SAS/ACCESS automatically generates the SQL `order by` clause and the database orders the result. This is supported through the SAS/ACCESS translation to SQL engine.

When migrating your SAS code to using SAS/ACCESS make sure you remove procedures that do not apply to the database, `proc sort` for example **[R].** In the example above if you run `proc sort` against the database it will exit with an error indicating that the database table does not work in replace mode. This case would not impact performance. If, on the other hand, your sort sends the data to a separate database table SAS/ACCESS would execute a `select` with `order by` statement and insert the results. Database tables are not ordered so this would be an unnecessary processing step.

Performance Tip: Porting [R]

If you are migrating code from using a SAS data set to using DB2, make sure you remove unnecessary procedures like proc sort. Even though they are not necessary they will execute and take time to process.

DATABASE ACCESS FROM SAS

There are two ways to connect to your DB2 database from SAS. You can define a connection using the libname engine or connect directly to the database using the `connect` statement in the SQL procedure.

When accessing the database using the libname engine SAS translates SAS data access application code to SQL. Translation to SQL means that SAS/ACCESS processes the SAS application code and generates the appropriate SQL to access the database.

Connecting directly to the database, using the `connect` statement, allows you to use explicit SQL pass-through. Explicit pass-through is a mechanism that allows you to pass unaltered SQL directly to the database server. Explicit SQL pass-through is useful for adding database only operations to your SAS application and is only accessible using the SQL procedure (`proc sql`).

Most SAS procedures and *DATA* steps use the SAS/ACCESS SQL translation engine. Following is an example of SAS to SQL translation for the `print` procedure above. When this same `proc print` procedure is executed using SAS/ACCESS against a DB2 database the request is translated into SQL for processing by DB2.

```
/*SQL Generated or Proc Print */
Select “state”,”serialno”
From hrecs
Order By State;
```

This is the SQL generated for the `proc print` statement only. As mentioned earlier the `proc sort` would not be necessary in this case. There are various reasons for using each type of access. Let’s take a look at a few examples of each.

WHEN WOULD YOU USE SAS/ACCESS TRANSLATION TO SQL?

You should use SAS/ACCESS translation to SQL when:

- You want to use SAS data access functionality (threaded reads, for example)
- You are joining data from multiple data sources
- The application needs to be portable to different relational databases
- The procedure or data step requires it. (e.g., `proc freq, proc summary`)

WHEN WOULD YOU USE EXPLICIT SQL PASS-THROUGH?

Explicit SQL pass-through should be used when:

- DB2 database processing steps are executed from a SAS application.*
- You want to use DB2 specific SQL

* Non-SAS processing means that SAS does not need to manipulate the data and all the work is done in the DB2 database server.

USING THE SAS/ACCESS LIBNAME ENGINE

The libname engine is used by SAS 9 to access data libraries (a data source). SAS uses the relational database library to access a DB2 database. The objective in this paper is to examine the relational database engine and how it interacts with DB2.

The SAS libname engine allows you to easily modify applications to use different data sources. You can modify a SAS statement that uses a SAS data set, to instead, use a table in your DB2 database simply by changing the libname definition. For example, here is a script that accesses a SAS data set named `mylib.data1` to generate frequency statistics.

```
/*   Define the directory /sas/mydata
as a SAS library */
libname  mylib data “/sas/mydata”;

/*  Run frequency statistics against
     the data1 data set */
proc freq data=mylib.data1;
     where state=’01’;
table state tenure yrbuilt yrmoved msapmsa;
```

```
   run;
```

To run frequency statistics against the data1 table in your DB2 database simply change the libname statement to access the database table instead of the SAS dataset:

```
   libname mylib db=DB2 user=me using=password;
```

Now, when this procedure is executed, SAS/ACCESS generates the proper SQL to retrieve the data from the database.

```
   SELECT "STATE", "TENURE", "YRBUILT", "YRMOVED", "MSAPMSA"
   FROM data1
   WHERE state = '01';
```

Using the libname engine is the easiest way to access DB2 database tables from your SAS application because SAS generates the appropriate SQL for you. To better understand SAS SQL translation let us take a look at how a SAS `DATA` step is processed by SAS.

In this example the `DATA` step reads and filters the data from the source table `source1` and writes the results to the database table `results1`. The result table contains all the housing records for state id '01'.

```
   libname census db=DB2 user=me
   using=password;

   data census.results1;
        set census.source1;
        where state = '01';
   run;
```

Coding Tip: Accessing long or case sensitive table names

If you need to access data from tables that are not in all upper case set the libname option: `PRESERVE_TAB_NAMES=YES`. This allows you to pass case sensitive or long table names to the database. This option is also important when accessing data using the SAS GUI tools. If you do not see the table names using the SAS explorer, for example, try setting this option.

SAS begins by opening a connection to the database to retrieve metadata and create the `results1` table. That connection is then used to read all the rows for `state='01'` from `source1`. SAS opens a second connection to write the rows to the newly created `results1` table. If this were a multi-tier configuration the data would travel over the network twice. First, the data is sent from the database server to the SAS server, then from the SAS server back to the database server. SAS processes the data in this manner to support moving data between various sources. This allows you to read data from one database server and write to the results to a different database. In this example, since a single data source is used and SAS does not need to process the data, this operation can be made more efficient **[R].**

```
   proc sql;
     connect to db2 (database=census);

   execute(Create Table results1
   like source1) as db2;

   execute(Insert Into results1
   Select *
   From source1
   Where state = '01') by db2;

   disconnect from db2;
   quit;
```

In this case, to make this operation more efficient, you can use the explicit SQL pass-through facility of the `sql` procedure. To make this statement explicit create a `sql` procedure with the necessary SQL statements. Connect to the database using the `connect` statement and wrap the SQL using the `execute statement`.

On the test system the original `DATA` step executed in 33 seconds, the explicit `proc sql` version executed in 15 seconds. Changing to explicit processing improved the performance of the operation by 64%. Since all the work here can be done by the DB2 database itself, explicit SQL is the most efficient way to process the statement.

EXPLICIT PASSTHROUGH VS SQL TRANSLATION

Improving Performance: Accessing DB2 Data with SAS 9

So why use explicit SQL instead of SQL translation (also called implicit SQL pass-through) when using `proc sql`? Explicit SQL is used in this case because an implicit `proc sql` statement is processed using a similar SQL translation engine as a `DATA` step.

If we pass this same statement as implicit SQL, SAS breaks the statement into separate select and insert statements. In this example, the performance gain realized using explicit SQL resulted from all the processing being handled by the database. A general rule of thumb would be: If SAS does not need to process it, let the database do the work. Explicit SQL is the best way to force this to happen.

Performance Tip: Coding

Let the database do as much work as possible if SAS does not need to process the data. Explicit SQL is the best way to force this to happen.

When SAS is executing a procedure it is important to understand which operations are done in DB2 and which operations the SAS server is processing.

LOADING AND ADDING DATA

SAS provides powerful extraction transformation and load (ETL) capabilities. Therefore, it is often used to load data into the database. SAS supports three methods of loading data into DB2: `Import, Load` and `CLI Load`. The load options are accessed through the bulk load interface.

If you have a procedure or `DATA` step that creates a DB2 table from flat file data, for example, using the bulkload engine the default load type is `IMPORT`. This option is best for small loads because it is easy to use and the user requires only insert and select privileges on the table. To enable bulk load using import set the option `BULKLOAD=yes . BULKLOAD` is a data step and libname option. Which one you choose depends on your application. For example, if you want all load operations against a libref to use import you would use the libname option.

To load large amounts of data quickly you should use the `LOAD` or `CLILOAD` options.

To use the DB2 Load feature, add the `BL_REMOTE_FILE DATA` step option. The `BL_REMOTE_FILE` option defines a directory for SAS to use as temporary file storage for the load operation. To process a load, SAS reads and processes the input data and writes it to a DB2 information exchange format (IXF) file. The BULKLOAD facility then loads the IXF file into the database using DB2 Load. Using the Load option requires the `BL_REMOTE_FILE` directory to have enough space to store the entire load dataset. It also requires the directory defined by `BL_REMOTE_FILE` be accessible to the DB2 server instance. This means it is on the same machine as DB2, NFS mounted or otherwise accessible as a file system.

New in DB2 V8.1 and SAS 9 is support for DB2 CLI Load. CLI Load uses the same high performance Load interface but allows applications to send the data directly to the database without having to create a temporary load file. CLI Load saves processing time because it does not create a temporary file and eliminates the need for temporary file system space. CLI Load also allows data to be loaded from a remote system. To enable the CLI Load feature using SAS, set the `BL_METHOD=CLILOAD DATA` step option instead of `BL_REMOTE_FILE`.

Tests ran comparing the different load options to give you an idea of the performance differences. This test executes a `DATA` step that loads 223,000 rows into a single database table. The following three code examples illustrate the DATA step bulk load options for each load method.

```
/* Method: Import */
data HSET(BULKLOAD=YES);
<…DATA step processing …>
run;

/* Method: Load */
data HSET(BULKLOAD=YES
 BL_REMOTE_FILE="/tmp”);
 <…DATA step processing …>
run;

/* Method: CLI Load */
data HSET( BULKLOAD=YES
     BL_METHOD=CLILOAD );
 <…DATA step processing …>
run;
```

Load Method	Time (seconds)
Import	76.69
Load	55.93
CLI LOAD	49.04

The table shows that by using CLI Load there is a 36 percent performance gain over import in this test. All load options require the table does not exist before the load.

RETRIEVING THE DATA INTO SAS

When reading from a DB2 database with SAS/ACCESS you can control access to partitioned tables, use multiple SAS threads to scan the database and utilize CLI multi-row fetch capabilities. SAS 9 and DB2 V8.1 have great improvements in the read performance of SAS applications. Threaded read is one of these new SAS 9 features. Threaded read allows SAS to extract data from DB2 in parallel, which is helpful for faster processing of a large or partitioned database. In DB2 V8.1 multi-row fetch has been improved, increasing read performance up to 45% over single-row fetch.

WHAT ARE THE PERFORMANCE IMPACTS OF THESE OPERATIONS?

There are three different SAS tuning parameters that can improve the speed of data transfer from DB2 to SAS: `READBUFF`, `DBSLICEPARM` and `DBSLICE`. These parameters correspond to the DB2 functions multi-row fetch, `mod()` and custom where-clause predicates respectively.

Functionality Match-up	
SAS Function	**DB2 Function**
`READBUFF`	Multi-Row Fetch
`DBSLICEPARM`	`mod()`
`DBSLICE`	Custom where-clause predicates

To examine the performance differences between these options frequency statistics were run against a database table. To generate frequency information SAS retrieves all the rows in the table. Since the math is not complex this a good test of I/O performance between SAS and DB2.

The first test was run using the default read options: single-row fetch and non-threaded read.

```
libname census db2 db=census user=db2inst1 using=password;

proc freq data=census.hrecs_db
    table state tenure yrbuilt yrmoved msapmsa;
run;
```

This test ran in 72.02 seconds.

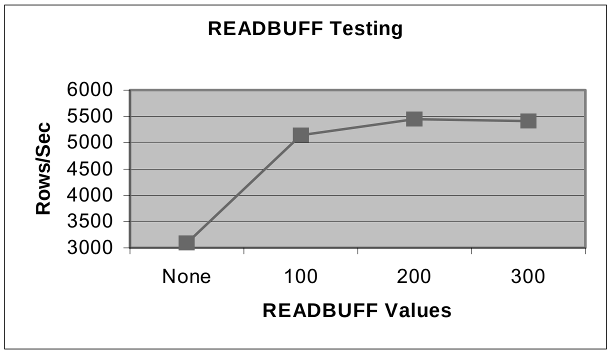
READBUFF

Using the default options SAS executes a single thread that reads one row at a time through the DB2 CLI interface. Sending one row at a time is not an efficient way to process large result sets. Transfer speed can be greatly improved by sending multiple rows in each request. DB2 supports this type of block request in CLI using multi-row fetch. SAS/ACCESS supports the DB2 multi-row fetch feature via the libname `READBUFF` option. `READBUFF=100` was added to the libname statement and the test run again.

```
libname census db2 db=census user=db2inst1 using=password READBUFF=100;
```

This time the frequency procedure took only 43.33 seconds to process. That is a 40% performance improvement over a single row fetch. This procedure was tested with other values of `READBUFF`. The testing indicated that the optimal value for this configuration is somewhere between 200 and 250, which allowed the query to run in 40.97 seconds. So from here on `READBUFF` is left at 200 and the new multi-threaded read options are tested using the same procedure.

Improving Performance: Accessing DB2 Data with SAS 9



Performance Note: READBUFF Results

For consistency, the performance using READBUFF and no threaded read options was tested in the multiple partition configuration. The results of the READBUFF test were the same as the single node so the results are comparable. **[**

THREADED READ

SAS 9 introduces a new data retrieval performance option called threaded read. The threaded read option works on the divide and conquer theory. It breaks up a single select statement into multiple statements allowing parallel fetches of the data from DB2 into SAS. SAS/ACCESS for DB2 supports the `DBSLICEPARM` and `DBSLICE` threaded read modes.

On a single-partition DB2 system you can use the `DBSLICE` or `DBSLICEPARM` option. Testing started using the automatic threaded read mode by setting `DBSLICEPARM`. When you use this option, SAS/ACCESS automatically determines a partitioning scheme for reading the data using the `mod()` database function. The `freq` procedure is tested using `dbsliceparm=(all,2)` which creates two threads that read data from the database.

```
proc freq data=census.hrecs_db  (dbsliceparm=(all,2));
table state tenure yrbuilt yrmoved msapmsa;
run;
```

When this statement is executed SAS/ACCESS automatically generates two queries: The first with the `mod(serialno,2)=0` predicate and the second with the `mod(serialno,2)=1` predicate. These queries are executed in parallel. Using the DBSLICEPARM option the same statement ran in 36.56 seconds, 10% faster than using `READBUFF` alone.

```
/* SQL Generated By SAS/Access */

SELECT  "STATE", "TENURE", "YRBUILT",   "YRMOVED", "MSAPMSA"
FROM HRECS_DB
WHERE
     {FN MOD({FN ABS("SERIALNO")},2)}=0
OR "SERIALNO" IS NULL ) FOR READ ONLY

SELECT  "STATE", "TENURE", "YRBUILT",   "YRMOVED", "MSAPMSA"
FROM HRECS_DB
WHERE
({FN MOD({FN ABS("SERIALNO")},2)}=1
OR "SERIALNO" IS NULL ) FOR READ ONLY
```

Using DBSLICEPARM works well in a single partition database, in a database with multiple partitions the most efficient way to retrieve data is directly from each partition.

When running a partitioned database you can use the DBSLICE threaded read mode to read directly from each database partition. DBSLICE allows you to define a custom method of partitioning requests by defining your own SQL where-clause predicates. To demonstrate the DBSLICE threaded read mode a two-partition database was set up and tested using the DB2 NODENUMBER predicate. When the DBSLICE option is specified SAS opens a separate connection for each query and retrieves the data. If your database partitions are on separate physical nodes, you can achieve the best performance by allowing allow SAS to open a connection directly to each node. To allow direct connections to each node you need to create a database alias on the DB2 client used by SAS for each node.

Performance Tip: Threaded Reads

For best read performance use the SAS 9 threaded read option whenever possible. DBSLICEPARM is the automatic threaded read option.

Using DBSLICEPARM=(ALL,2)

ALL: Makes all read-only procedures eligible for threaded reads.

2: Starts two read threads.

```
-- Catalog a node
CATALOG TCPIP NODE dbnode1 REMOTE myserver SERVER 70000

-- Catalog a database
CATALOG DATABASE dbname AT NODE dbnode1
```

Using a two-logical partition DB2 database on the test server the DBSLICEPARM option was replaced with the DBSLICE syntax in the freq procedure script. Since the test database consists of multiple partitions on a single system, we only need to catalog a single node. The freq procedure was executed using the DBSLICE option along with the NODENUMBER syntax ().

To execute the request SAS opens two connections to the database server and executes the SQL necessary to retrieve data from each partition.

```
/* SQL Generated By SAS/Access */
SELECT  "STATE", "TENURE", "YRBUILT",
        "YRMOVED", "MSAPMSA"
FROM HRECS_DB
WHERE NODENUMBER(serialno)=0
FOR READ ONLY

SELECT  "STATE", "TENURE", "YRBUILT",
        "YRMOVED", "MSAPMSA"
FROM HRECS_DB
WHERE NODENUMBER(serialno)=1
FOR READ ONLY
```

This time the query ran in 35.13 seconds, 14% faster than using the READBUFF option alone.

As you can see, these threaded read options improve data extraction performance. This was a small two-partition system. As the database size or number of partitions increases, these threaded read options should help even more.

Not all operations take advantage of threaded reads. A statement does not qualify for threaded reads if:

- A BY statement is used
- The OBS option is specified
- The FIRSTOBS option is specified
- The KEY/DBKEY= option is specified
- There is no valid input column for the mod() function (effects DBSLICEPARM).
- All valid columns for the mod() function are also listed in the where-clause (effects DBSLICEPARM)
- If NOTHREADS option is specified
- The value of DBSLICEPARM option is set to NONE.

If your statement does not qualify for threaded reads and you attempt to enable it then the saslog will contain an error, for example, "Threading is disabled due to the ORDER BY clause or the FIRSTOBS/OBS option …".

Improving Performance: Accessing DB2 Data with SAS 9

PUTTING DATA INTO YOUR DB2 DATABASE

Read performance is most important when using SAS to extract data from DB2 but it is also important to tune your applications that add rows to the database. This paper examines two ways to improve the performance of inserting data: `DBCOMMIT` and `INSERTBUFF`.

Performance Tip: Coding

It is always best to limit the amount of data transferred between DB2 and SAS. If you only need a few columns for your analysis list the columns you need from the source table. In the example above changing

```
   set census.source1;
```

to

```
   set census.source1 (keep=state puma);
```

tells SAS to only retrieve the columns needed.

DBCOMMIT

`DBCOMMIT` sets the number of rows inserted into the database between transaction commits. To understand how this works let's look at what is happening behind the scenes when inserting a row into the database.

To insert one row into a database table there are many operations that take place behind the scenes to complete the transaction. For this example we will focus on the transaction logging requirements of an SQL `insert` operation to demonstrate the impact of the SAS `DBCOMMIT` option.

The DB2 database transaction log records all modifications to the database to ensure data integrity. During an insert operation there are multiple records recorded to the database transaction log. For an insert, the first record is the insert itself, followed by the commit record that tells the database the transaction is complete. Both of these actions are recorded in the database transaction log in separate log records. For example, if you were to insert 1000 rows with each row in its own transaction it would require 2000 (1000 insert and 1000 commit) transaction log records. If all these inserts were in a single transaction you could insert all the rows with 1001 transaction log records (1000 insert and 1 commit).

You can see that there is a considerable difference in the amount of work required to insert the same 1000 rows depending on how the transaction is structured. By this logic, if you are doing an insert, you want to set `DBCOMMIT` to the total number of rows you need to insert. This requires the least amount of work, right? Not quite; as with any performance tuning there are tradeoffs.

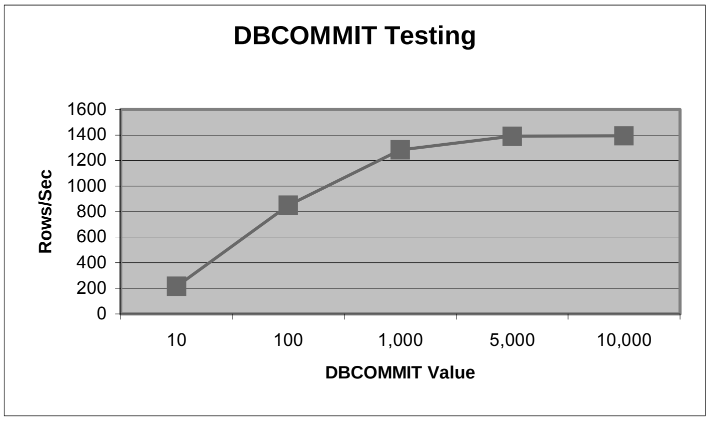
For example, if you were to insert 1 million rows in a single transaction, this will work but it requires a lock to be held for each row. As the numbers of locks required increases the lock management overhead increases. With this in mind you need to tune the value of `DBCOMMIT` to be large enough to limit commit processing but not so large that you encounter long transaction issues (locking, running out of log space, etc). To test insert performance a `DATA` step is used that processes the `hrecs` table and creates a new `hrecs_temp` table containing all the rows where state = '01'.

```
libname census db2 db=census user=db2inst1 using=password;

data census.hrecs_temp (DBCOMMIT=10000);
    set census.hrecs;
    where state = '01';
run;
```

The default for `DBCOMMIT` is 1000. Testing started at 10 just to see the impact.

DBCOMMIT	Time (sec)
10	397.10
100	101.13
1,000 (default)	67.03
5,000	61.98
10,000	61.77



As you can see the default works pretty well. In some situations, like this one, larger values of `DBCOMMIT` can yield up to a 16% performance improvement over the default. This test shows that the best value was between 1,000 and 5,000 rows per commit.

You can see that at some point, increasing the value of `DBCOMMIT` no longer improves performance (Compare 5,000 to 10,000).

In fact, if the dataset were large enough performance would probably decrease with extremely high values of `DBCOMMIT`. Now that `DBCOMMIT` is tuned let's take a look at another parameter that impacts insert performance: `INSERTBUFF`.

Performance Tip: Tuning

Start with DBCOMMIT between 1,000 and 5,000 and tune from there.

INSERTBUFF

`INSERTBUFF` is another tunable parameter that affects the performance of SAS inserting rows into a DB2 table. `INSERTBUFF` enables CLI insert buffering similar to read buffering (using `READBUFF)` but for inserts. It tells the CLI client how many rows to send to the DB2 server in each request. To enable insert buffering you need to set the libname option `INSERTBUFF`.

```
libname census db2 db=census user=db2inst1 using=password;

data census.hrecs_temp (INSERTBUFF=10 DBCOMMIT=5000);
    set census.hrecs;
    where state = '01';
run;
```

`INSERTBUFF` is an integer ranging from 1 to 2,147,483,648. Different values of `INSERTBUFF` were tested to see what impact it would have on the preceding `DATA` step.

INSERTBUFF	Time (sec)
1	61.77
5	60.64
10	60.22
25	60.36
50	60.36
100	60.86

Performance Tip: PROC APPEND

You can use `proc append` to add data to an existing table. The append procedure uses the same insert process as the previous `DATA` step so the same tuning rules apply.

Increasing the value of `INSERTBUFF` from 1 to 25 improved the performance only 2.7% over the tuned DBCOMMIT environment . Increasing the value over 25 did not have a significant impact on performance. So why use INSERTBUFF? As mentioned earlier, parameters effect various environments differently. To show these differences the same test was run with SAS and DB2 running on separate servers using a TCP/IP connection instead of a local shared memory connection.

INSERTBUFF	TCP/IP Time (sec)
1	150.75
5	109.05
10	104.52
25	99.35
50	98.23
100	98.11

In this case increasing the value of `INSERTBUFF` from 1 to 25 had a significant impact, improving performance 51% over the default. These results show that when you are running in a local (shared memory) configuration INSERTBUFF has little impact on performance and you may want to tune DBCOMMIT. On the other hand when you are connecting from a separate server INSERTBUFF can have a significant impact on performance.

Notice that good values of INSERTBUFF are about 10 times smaller than good values for READBUFF. You should keep this in mind when you are tuning your system.

FUNCTION PUSHDOWN

You have seen that SAS/ACCESS can push down where-clause processing and join operations to DB2. SAS/ACCESS can also push down function processing. If the database supports the function you request, SAS can pass that function processing to the database. To enable the most functions to be pushed down to DB2 set the `SQL_FUNCTIONS=all` libname option.

Applying these functions in the database can improve analysis performance. Each aggregate (vector) function (examples: AVG, SUM) that DB2 processes means fewer rows of data are passed to the SAS server. Processing non-aggregate (scalar) functions (ABS, UPCASE etc…) can take advantage of DB2 parallel processing.

ABS	FLOOR	LOWCASE (LCASE)
ARCOS (ACOS)	LOG	UPCASE (UCASE)
ARSIN (ASIN)	LOG10	SUM
ATAN	SIGN	COUNT
CEILING	SIN	AVG
COS	SQRT	MIN
EXP	TAN	MAX

SAS will push down operations to the database that limit the number of rows returned. For example, if you have a simple query that does a `select` with an absolute value (abs) function. The function will be applied in SAS because applying it at the database does not limit the number of rows returned to SAS.

```
Proc SQL Statement:

Create table census.FUNCTABLE as
select
    abs(RHHINC) as sumcol
from  census.hrecs_test2;

Generated SQL:

SELECT "RHHINC"
FROM "HRECS_TEST2" FOR READ ONLY
```

On the other hand, if the query included a limiting function(in this case a distinct clause was added), the operation will be pushed down to the database.

```
Proc SQL Statement:

create table census.FUNCTABLE as
select
    distinct abs(RHHINC) as sumcol
from  census.hrecs_test2;

Generated SQL:

select
  distinct {fn ABS("HRECS_TEST2"."RHHINC")} as sumcol
 from "HRECS_TEST2"
```

Note that when you a create table containing a derived column (i.e. calling a function to generate a value) in the database you must name the derived column using a column alias.

To take further advantage of single-pass processing you can add a view to your data in DB2. Views in DB2 allow you to join tables, transform variables (generate ranks for a variable, for example) and group results in a single pass of the data. You can further prepare the data before SAS processing by creating a Materialized Query Table (MQT) or by applying Multidimensional Clustering (MDC) to your table.

THE DBA CORNER

HOW DOES SAS USE MY DATABASE?

To a DB2 DBA, SAS is another consumer of database resources. This section highlights a few topics of interest to the DB2 DBA. It looks at the way SAS connects to the database and how it uses other database resources. Included are some debugging tips for SAS applications in a SAS/ACCESS Interface for DB2 environment.

CONNECTIONS

Each SAS client opens multiple connections to the database server. When a SAS session is started a single data connection is opened to the database. This connection is used for most communication from SAS to DB2. If the SAS application requires metadata from the DB2 catalogs, executing `proc datasets` for example, a second, utility connection is created. This utility connection is designed to allow SAS to collect this information without interfering with the original data connection. It also allows these utility operations to exist in a separate transactional context. By default, these two connections remain active until the SAS session is completed or the libname reference or database connection (opened using the `connect` statement) is explicitly closed. Other connections may be opened automatically during a SAS session. These connections are closed when the operation for which they were opened is completed. For example, if you read from one table and write to a new table, SAS opens two connections: The original connection to retrieve the data and a connection to write the data into the new table. In this example the connection used to write the data will be closed when that `DATA` step is completed.

NOTE: Threaded Reads
Threaded read operations do not abide by the CONNECTION settings. This means that even if CONNECTION is set to SHAREDREAD (which is the default) a single operation using DBSLICEPARM, for instance, will open multiple connections to the database.

You can control how connections are managed in SAS/ACCESS using the `CONNECTION=` libname option. The default connection mode is `SHAREDREAD`, which means all read operations in a single SAS session use a common connection for reading data. If you need to limit the total number of connections opened to the database you can use the SHARED option. The shared mode uses one connection per SAS session for all data operations except the utility connection. This limits each SAS client to two connections: one data connection (read and write) and one utility connection. By default the utility connection stays around for the duration of the session.

There are many different connection modes including: `SHAREDREAD, UNIQUE, SHARED, GLOBALREAD, GLOBAL.` In this section the `SHAREDREAD, UNIQUE and SHARED` options are used to test the performance implications of sharing connections.

The `append` procedure was used to measure the impact of different connection options. Connection testing was done with SAS and DB2 running on separate systems because connection impact is more apparent when using TCP/IP instead of local shared memory.

The first test executes a `proc append` which reads from one table and appends the results to another table in the same database. Proc append is used to allow testing in a mixed read/write environment.

```
libname census db2 db=census user=user1 using=password CONNECTION=UNIQUE;

proc append BASE=census.append1 data=census.hrecsca;
run;
```

This test consisted of executing `proc append` in three different connection modes: SHAREDREAD (default), SHARED and UNIQUE. SHAREDREAD means that each SAS session gets one connection for all read operations and separate connections for each write operation. SHARED opens one connection for each SAS session for reads and writes. In UNIQUE mode a separate connection is opened for each read and write operation. The table to the right shows the results of this test.

As you can see in this case using a SHARED read mode is 241% slower than the other two connection options. In other words before using SHARED you should test the impact on your workload. If your environment is more read oriented, for example, using SHARED may not have as large an impact. In a read/write environment if your goal is to use fewer connections you may try scheduling the jobs to limit concurrent access instead of sharing connections.

Connection Mode	Time (seconds)
SHAREDREAD	38.35
SHARED	130.90
UNIQUE	39.24

RESOURCE CONSUMPTION

DBAs are always interested in understanding what impact an application has on the database. SAS workloads can vary greatly depending on your environment, but here are a few places to start evaluating your situation:

- Each SAS user is the equivalent of a single database Decision Support (DS) user. Tune the same for SAS as you would for an equivalent number of generic DS users.
- Tune to the workload. Just like any other DS application, understanding the customer requirements can help you to improve system performance. For example, if there is a demand for quarterly or monthly data, using a Multidimensional Clustering (MDC) table for the data may be appropriate.
- SAS is a decision support tool; if you need data from an operational data store consider the impact on your other applications. To offload some of the work you may consider creating a data mart to provide data to your SAS customers.
- In most environments the SAS server is located on a separate system from your database. Business analysis often requires many rows to be retrieved from the database. Plan to provide the fastest network connection possible between these systems to provide the greatest throughput.
- As with any workload keep your database statistics up to date using `runstats`. This enables the optimizer generate the optimal access plan.

RULES OF THUMB

- Try to pass as much where-clause and join processing as you can to DB2.
- Return only the rows and columns you need. Whenever possible do not use a “`select * …`” from your SAS application. Provide a list of the necessary columns, using `keep=(var1, var2…)`, for example. To limit the number of rows returned include any appropriate filters in the `where`-clause.
- Provide Multidimensional Clustering (MDC) tables or Materialized Query Tables (MQT) where appropriate to provide precompiled results for faster access to data.
- Use SAS threaded read (DBSLICEPARM, DBSLICE) and multi-row fetch (READBUFF) operations whenever possible.
- When loading data with SAS use the bulk load method CLI Load.

DEBUGGING

To see what SQL commands SAS is passing to the database, enable the `sastrace` option. For example, here is the syntax to trace SAS/ACCESS SQL calls to DB2:

```
options sastrace “,,,d” sastraceloc=saslog;
```

Applying a “d” in the fourth column of the sastrace options tells SAS to report SQL sent to the database. For example this SAS data Step:

```
data census.hrecs_temp
    (keep=YEAR RECTYPE SERIALNO STATE);
    keep YEAR RECTYPE SERIALNO STATE;
    set census.hrecs_db;
    where state=1;
```

```
run;
```

Is logged as (the SQL commands are highlighted):

```
TRACE: Using EXTENDED FETCH for file HRECS_DB on connection 0 476 1372784097
rtmdoit 0 DATASTEP

TRACE: SQL stmt prepared on statement 0, connection 0 is:
  SELECT *
  FROM HRECS_DB FOR READ ONLY 477 1372784097 rtmdoit 0 DATASTEP

TRACE: DESCRIBE on statement 0, connection 0. 478 1372784097 rtmdoit 0
DATASTEP
622  data census.hrecs_temp (keep=YEAR RECTYPE SERIALNO STATE);
623          keep YEAR RECTYPE SERIALNO STATE;
624          set census.hrecs_db;
625          where state=1;
626  run;

TRACE: Using FETCH for file HRECS_TEMP on connection 1 479 1372784097
rtmdoit 0 DATASTEP
TRACE: Successful connection made, connection id 2 480 1372784097
rtmdoit 0 DATASTEP
TRACE: Database/data source: census 481 1372784098 rtmdoit 0 DATASTEP
TRACE: USER=DB2INST1, PASS=XXXXXXX 482 1372784098 rtmdoit 0 DATASTEP
TRACE: AUTOCOMMIT is NO for connection 2 483 1372784098 rtmdoit 0
DATASTEP
TRACE: Using FETCH for file HRECS_TEMP on connection 2 484 1372784098
rtmdoit 0 DATASTEP
NOTE: SAS variable labels, formats, and lengths are not written to DBMS
      tables.

TRACE: SQL stmt execute on connection 2:
   CREATE TABLE HRECS_TEMP
```

(YEAR DATE,RECTYPE VARCHAR(1),SERIALNO

Hint: To find SQL information search the log for "SQL".

Sastrace displays the processing details of the SAS script. The log includes the exact SQL commands that are submitted to the database. The example above executes a *DATA* step that SAS translates into the following SQL statements.

```
SELECT *  FROM HRECS_DB;

CREATE TABLE HRECS_TEMP
(YEAR DATE,RECTYPE VARCHAR(1),SERIALNO NUMERIC(11,0),STATE NUMERIC(11,0));

SELECT
"YEAR", "RECTYPE", "SERIALNO", "SAMPLE", "DIVISION", "STATE", "PUMA",
"AMORTG2", "AMRTAMT2", "ACNDOFEE", "AMOBLHME"  FROM HRECS_DB
WHERE  ("STATE" = 1 );

INSERT INTO HRECS_TEMP
(YEAR,RECTYPE,SERIALNO,STATE)
VALUES ( ? , ? , ? , ? );
```

Using DB2 you can see what SAS/ACCESS is requesting on the database by enabling CLI trace. The DB2 CLI trace feature is useful for debugging SAS/ACCESS interaction with DB2. There are two ways to enable CLI trace: the DB2 Command Line Processor (CLP) command, "`update cli cfg`" or edit the `sqllib/cfg/db2cli.ini` file. To enable tracing using "`update cli`" enter

```
db2 UPDATE CLI config FOR common USING trace 1
```

Then

```
   db2 UPDATE CLI config FOR SECTION common USING TraceFileName /tmp/mytracefile
```

If you choose to edit the `db2cli.ini` file directly add `trace` and `TraceFileName` in the common section

```
   [COMMON]
   trace=1
   TraceFileName=/tmp/mytracefile
   TraceFlush=1
```

When you enable CLI tracing DB2 begins tracing all CLI statements executed on the server. DB2 will continue to trace CLI commands until you disable CLI tracing by setting `trace` to 0. Be careful: if this is a busy server you could collect huge amounts of output. It is best to run CLI trace on a test server running a single SAS session, if possible.

The saslog and DB2 log (sqllib/db2dump/sqdb2diag.log) are also useful places to look for information when you are troubleshooting.

CONCLUSION

The following is a brief summary of how the topics covered fit into the three main tuning categories:

1. Reduce the data transferred
   - Use predicates to limit the number of rows returned
   - Select only the columns you need
   - Precompile some information using database views or Materialized Query Table (MQT)
2. Tune each processing step by using
   - READBUFF to improve read performance
   - INSERTBUFF and DBCOMMIT to improve write performance
   - Bulk load for loading large data
   - Saslog to analyze your SAS application
3. Parallelize where possible
   - Use DBSLICEPARM or DBSLICE for reading data
   - Utilize DB2 parallel data access

SAS 9 and DB2 V8.1 provide expanded functionality and performance over earlier version in SAS BULKLOAD, and Threaded Read and DB2 multi-row fetch and CLI Load performance. This paper has illustrated how tuning your SAS applications, using these new features, can greatly improve performance.

CONTACT INFORMATION

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

Scott Fadden
IBM
DB2 Data Management Software
sfadden@us.ibm.com

Additional information is available at **ibm.com**/db2 and www.sas.com