Paper 3269-2015

SASTRACE: Your Key to RDBMS Empowerment

Andrew Howell, ANJ Solutions Pty Ltd

ABSTRACT

For the many relational databases supported by SAS/ACCESS® products (Oracle, Teradata, DB2, MySQL, SQL Server, Hadoop, Greenplum, PC Files, to name but a few), there are a myriad of RDBMS-specific options at your disposal, but how do you know the right options for any given situation?

- How much data should you transfer at a time?
- Which SAS® functions can be passed through to the database? Which cannot?
- How do you verify your processes are running efficiently?
- How do you test and validate any changes?

The answer lies with the feedback capabilities of the **SASTRACE** system option.

INTRODUCTION

Part of the beauty of SAS/ACCESS is its transparency; the user can interrogate different data sources using similar SAS code (if not identical, apart from LIBNAME statements), without the need to re-code to take into account the nuances of each different data source.

However, this transparency can also serve as a trap, as SAS/ACCESS may apply default decisions or values on the user's behalf when it might be better for the user to have more explicit control over some RDBMS operations.

The global options **SASTRACE** & **SASTRACELOC** are invaluable in helping a SAS developer determine (and change, test & verify) how much work is being done by SAS, and how much by the database.

The preferable solution is one which achieves the desired result with the most efficient transference of data between SAS & the database. Efficiency is all too often compromised by options, where clauses, joins, etc, which fail to differentiate between which processes are SAS-specific and which processes can be "passed through" to the underlying database.

This paper discusses the more common **SASTRACE** option settings, and offers examples where **SASTRACE** can be used to assist in refining SAS processes running on relational data.

For the demonstrations used in this paper:

- The SAS client is SAS Enterprise Guide 6.1 (64-bit) running on a Windows 7 (64-bit) workstation.
- The SAS server SAS 9.4 M2 on Linux 64-bit server.
- Greenplum data is stored on a separate server in a Greenplum database schema, with a **"GPDB"** Greenplum libname defined; this schema contains a copy of the SASHELP.ZIPCODE table.

SASTRACE & SASTRACELOC GLOBAL OPTIONS

The table below is a summary of the more common **SASTRACE** options; it is taken from the SAS ACCESS 9.4 for Relational Databases, Reference:

Setting	Description
',,,d'	Displays SQL statements, API calls and any parameters sent to the RDBMS. These include (but are not limited to) SELECT, DELETE, CREATE, SYSTEM CATALOG, DROP, COMMIT, INSERT, ROLLBACK, UPDATE
',,,db'	Displays a summary of the SQL statements that the ',,,d' option normally generates.
',,,s'	Displays timing summaries of the SQL statemements.
',,,sa'	Displays timing details & summaries of SQL statements.
',,t,'	Displays information of SQL threaded reads and/or writes, including the number of threads, and the number of observations each thread processes.
',,d,'	Displays all API routine calls, including all function enters, exits, parameters and return codes. Generally only used in major troubleshooting.
'd,'	Displays all DBMS calls - API and client calls, connection information, column bindings, column error information, and row processing) are sent to the log. Again, generally only used in major troubleshooting.

Table 1. Summary of the more common SASTRACE options

The **SASTRACELOC** option defines the output destination of **SASTRACE** messages. It can be set to SASLOG (i.e, the SAS log), stdout, or to an external text file.

For the code examples used in this paper, **SASTRACELOC** has been set to SASLOG.

PASS-THROUGH OF SAS FUNCTIONS & WHERE CLAUSES

Each database has its own native functions (as does SAS), used for field calculations and filtering of rows and/or results. Where possible, the SAS/ACCESS engine will endeavour to “pass-through” such functions to the underlying database. Where this is not possible, data must be returned for SAS to perform such calculations and/or filters. **SASTRACE** can be used to return SAS/ACCESS SQL results to the log, which can aid in identifying where code might be modified to reduce unnecessary transfers of large amounts of data.

Below is a table containing sample code to extract non-missing data from two identical tables – one stored as a SAS data set, the other a table in a Greenplum database schema:

	Missing() function	Missing operator

SAS data store	`data A;`
	`  set sashelp.zipcode;`
	`  where not missing(zip_class);`
	`run;`
		`data B;`
		`  set sashelp.zipcode;`
		`  where zip_class ne ' ';`
		`run;`

Greenplum data store	`data C;`
	`  set gpdb.zipcode;`
	`  where not missing(zip_class);`
	`run;`
		`data D;`
		`  set gpdb.zipcode;`
		`  where zip_class ne ' ';`
		`run;`

Table 2. Sample code to extract non-missing records

Judging by the code, these would all appear to achieve the same result, regardless of method.

The time taken to perform both “SAS data store” extractions is almost identical, regardless of whether the MISSING() function or the missing operator is used.

However, when it comes to extracting data from a database, is one method particularly more or less efficient that the other? This is where **SASTRACE** is useful to show how much work is done by the database, and how much by SAS.

Below is the SAS log of the Greenplum extraction code; relevant code, **SASTRACE** messages and DATA STEP timings have been highlighted. **SASTRACE** has been set to `',,,db'` to display a summary of all generated SQL code:

```
23          options sastrace=',,,db' sastraceloc=saslog ;

35          data _null_;
36            set gpdb.zipcode;
37            where not missing (zip_class);
38          run;

GREENPL_2540: Prepared: on connection 0 21305 1427337958 no_name 0 DATASTEP
SELECT  "zip", "y", "x", "zip_class", "city", "state", "statecode",
"statename", "county", "countynm", "msa", "areacode", "areacodes", "timezone",
"gmtoffset", "dst", "poname", "alias_city", "alias_cityn", "city2",
"statename2"  FROM gp_stg.ZIPCODE  FOR READ ONLY  21306 1427337958 no_name 0
DATASTEP

NOTE: There were 11455 observations read from the data set GPDB.ZIPCODE.
      WHERE not MISSING(zip_class);
NOTE: DATA statement used (Total process time):
      real time           2.23 seconds
      user cpu time       0.28 seconds
      system cpu time     0.00 seconds

40          data _null_;
41            set gpdb.zipcode;
42            where zip_class ne ' ';
43          run;

SELECT  "zip", "y", "x", "zip_class", "city", "state", "statecode",
"statename", "county", "countynm", "msa", "areacode", "areacodes", "timezone",
"gmtoffset", "dst", "poname", "alias_city", "alias_cityn", "city2",
"statename2"  FROM gp_stg.ZIPCODE  WHERE  ( "zip_class" <> ' ' AND  "zip_class"
IS  NOT  NULL  ) FOR READ ONLY  21318 1427337960 no_name 0 DATASTEP

NOTE: There were 11455 observations read from the data set GPDB.ZIPCODE.
      WHERE zip_class not = ' ';
NOTE: DATA statement used (Total process time):
      real time           0.70 seconds
      user cpu time       0.08 seconds
      system cpu time     0.00 seconds
```

Output 1. Partial SAS Log, comparing where clause functions & operations

From the SAS log, we can note the following:

- When using the MISSING() function - a function not supported in Greenplum – **the database returns ALL rows to SAS**, which then performs the filter**.** This can be particularly inefficient if the intended subset represents a small percentage of the source table.
- When using the missing operator, the SAS/ACCESS engine successfully converts the WHERE statement to a filter supported by the database, **returning only the filtered rows**, resulting in a significantly more efficient process.

USING SASTRACE FOR MORE EFFICIENT DATA TRANSFER

Libname & data set options control how many rows are read from or written to a database table: **READBUFF** (for reading) & **INSERTBUFF** (for writing) are the primary options.

*(For uploading of large data sets to a data warehouse, there are **FASTLOAD** & **BULKLOAD** options; these are not covered in this paper.)*

Different databases have different default values for **READBUFF** & **INSERTBUFF** - some are set at specific (and typically conservative) values; others are dynamically calculated based on information pertaining to the data source (observation length, and so on); in some cases, the default value is 1 !

Below is sample code used to test various values of the **INSERTBUFF** data set option. **SASTRACE** has been set to **`',,t,dsab'`** to display a summary of all generated SQL code, any threaded read/write operations, and timing summary & details:

```
options sastrace=',,t,dsab' sastraceloc=saslog;

%macro Upload(x,obs=1000);
       %put Upload(&x);
       data GPDB.UploadTest(insertbuff=&x);
              set sashelp.zipcode(obs=&obs);
       run;
       proc sql noprint;
              drop table GPDB.UploadTest;
       quit;
%mend;
%Upload(20);
%Upload(40);
%Upload(60);
%Upload(80);
%Upload(100);
%Upload(200);
%Upload(300);
%Upload(400);
%Upload(500);
```

Output 2. SAS sample, demonstrating SASTRACE with various INSERTBUFF options.

Below is the partial log for the **`%Upload(20)`** macro call; relevant **SASTRACE** messages have been highlighted, as well as the Real & User CPU times:

```
Upload(20)

<< Several rows cut..>>

GREENPLUM: INSERT time in seconds for 20 row(s) is    3.726424 703 1427365198
no_name 0 DATASTEP
  704 1427365199 no_name 0 DATASTEP
GREENPL_83: Executed: on connection 2 705 1427365199 no_name 0 DATASTEP
Prepared statement GREENPL_81 706 1427365199 no_name 0 DATASTEP
  707 1427365199 no_name 0 DATASTEP
GREENPLUM: INSERT time in seconds for 20 row(s) is    0.656055 708 1427365199
no_name 0 DATASTEP
  709 1427365199 no_name 0 DATASTEP
GREENPL_84: Executed: on connection 2 710 1427365199 no_name 0 DATASTEP
Prepared statement GREENPL_81 711 1427365199 no_name 0 DATASTEP
  712 1427365199 no_name 0 DATASTEP

<< Many rows cut..>>
```

```
GREENPLUM: INSERT time in seconds for 20 row(s) is    0.639372 943 1427365227
no_name 0 DATASTEP
  944 1427365227 no_name 0 DATASTEP
GREENPL_131: Executed: on connection 2 945 1427365227 no_name 0 DATASTEP
Prepared statement GREENPL_81 946 1427365227 no_name 0 DATASTEP
  947 1427365227 no_name 0 DATASTEP
GREENPLUM: INSERT time in seconds for 20 row(s) is    0.680664 948 1427365227
no_name 0 DATASTEP
GREENPL: COMMIT performed on connection 2. 949 1427365227 no_name 0 DATASTEP
NOTE: There were 1000 observations read from the data set SASHELP.ZIPCODE.
GREENPL: COMMIT performed on connection 2. 950 1427365227 no_name 0 DATASTEP
NOTE: The data set GPDB.UPLOADTEST has 1000 observations and 21 variables.
GREENPL: COMMIT performed on connection 2. 951 1427365227 no_name 0 DATASTEP
  952 1427365227 no_name 0 DATASTEP
Summary Statistics for GREENPLUM are: 953 1427365227 no_name 0 DATASTEP
Total SQL execution seconds were:                 33.508411 954 1427365227
no_name 0 DATASTEP
Total SQL prepare seconds were:                    0.003808 955 1427365227
no_name 0 DATASTEP
Total SQL row insert seconds were:                33.092145 956 1427365227
no_name 0 DATASTEP
Total seconds used by the GREENPLUM ACCESS engine were    33.605424 957
1427365227 no_name 0 DATASTEP
  958 1427365227 no_name 0 DATASTEP
GREENPL: COMMIT performed on connection 2. 959 1427365227 no_name 0 DATASTEP
NOTE: DATA statement used (Total process time):
      real time           33.63 seconds
      user cpu time       0.12 seconds
```

Output 3. Partial SAS log of the first %Upload() macro call

The table below contains the results for all the **`%Upload()`** macro calls:

Rows	RealTime	CPUTime
20	33.63	0.12
40	33.92	0.09
60	33.73	0.06
80	39.54	0.05
100	33.82	0.05
200	42.13	0.04
300	33.2	0.04
400	41.91	0.05
500	34.25	0.05

Table 3. Summary of %Upload() results

The figure below is a graphical representation of these results:

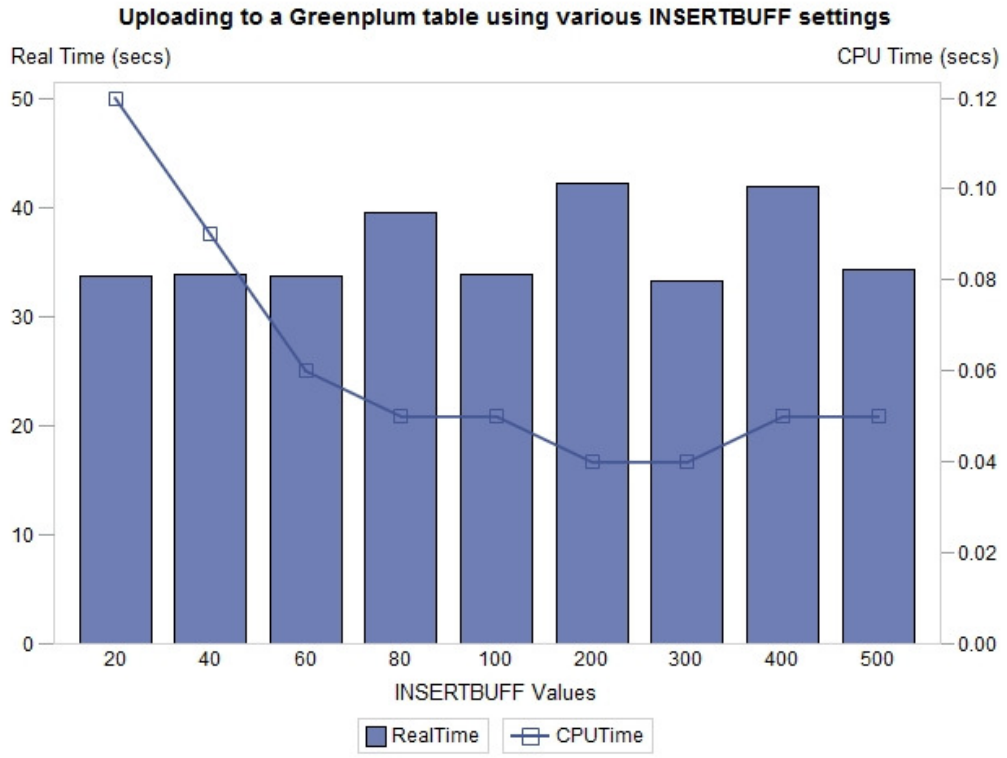

Figure 1. Summary of %Upload() results

From this graph, it would suggest an **INSERTBUFF** setting of approximately 250 would be ideal. **INSERTBUFF** values of less than 250 will result in more fetch operations; greater than 250 may result in caching memory to disk which would adversely affect processing times.

Of interest is the minimal change in Real Time results, most likely due to not being large enough to have a noticeable impact, plus the possibility that there were other operations taking place on the database server at the same time.

USING SASTRACE TO EXAMINE PARITIONED READ/WRITES

In a multi-threaded architecture, efficient data transference between SAS & relational databases can be achieved by spreading the workload over multiple threads, each of which manages a share of the overall workload. **SASTRACE** can be used to return the results of threaded operations.

Below are (amended) SAS logs of some Greenplum data extraction code; relevant data step options, and **SASTRACE** options & messages have been highlighted. In each case, **SASTRACE** has been set to **`',,t,db'** to display a summary of all generated SQL code, and any threaded read/write operations.

In this first example, no partition options are selected:

```
23          options sastrace=',,t,db';
24
25          data _null_;
26            set GPDB.zipcode ;
27            where zip_class eq " ";
28          run;

GREENPL_2: Prepared: on connection 0 22165 1427351162 no_name 0 DATASTEP
```

```
SELECT  "zip", "y", "x", "zip_class", "city", "state", "statecode",
"statename", "county", "countynm", "msa", "areacode", "areacodes", "timezone",
"gmtoffset", "dst", "poname", "alias_city", "alias_cityn", "city2",
"statename2"  FROM gp_stg.ZIPCODE  WHERE  ( "zip_class" = ' ' OR  "zip_class"
IS  NULL  ) FOR READ ONLY
GREENPL_3: Executed: on connection 0 22169 1427351163 no_name 0 DATASTEP
NOTE: There were 29812 observations read from the data set GPDB.ZIPCODE.
      WHERE zip_class=' ';
NOTE: DATA statement used (Total process time):
      real time           1.38 seconds
      user cpu time       0.19 seconds
```

Output 4. SAS Log, with no partition options.

In the following example, an arbitrary partition is used:

```
30         data _null_;
31           set GPDB.zipcode(DBSLICE=("GMTOFFSET<0" "GMTOFFSET>0"));
32           where zip_class eq " ";
33         run;

SELECT  "zip", "y", "x", "zip_class", "city", "state", "statecode",
"statename", "county", "countynm", "msa", "areacode", "areacodes", "timezone",
"gmtoffset", "dst", "poname", "alias_city", "alias_cityn", "city2",
"statename2"  FROM gp_stg.ZIPCODE  WHERE  ( "zip_class" = ' ' OR  "zip_class"
IS  NULL  ) AND  GMTOFFSET<0 FOR READ ONLY  22178 1427351164 no_name 0 DATASTEP
GREENPL_6: Executed: on connection 0 22181 1427351164 no_name 0 DATASTEP
SELECT  "zip", "y", "x", "zip_class", "city", "state", "statecode",
"statename", "county", "countynm", "msa", "areacode", "areacodes", "timezone",
"gmtoffset", "dst", "poname", "alias_city", "alias_cityn", "city2",
"statename2"  FROM gp_stg.ZIPCODE  WHERE  ( "zip_class" = ' ' OR  "zip_class"
IS  NULL  ) AND  GMTOFFSET>0 FOR READ ONLY  22182 1427351164 no_name 0 DATASTEP
GREENPLUM:  Thread 2 contains 5 obs. 22184 1427351164 no_name 0 DATASTEP
GREENPLUM:  Thread 1 contains 29807 obs. 22185 1427351164 no_name 0 DATASTEP
GREENPLUM:  Threaded read enabled. Number of threads created: 2 22186
1427351164 no_name 0 DATASTEP
NOTE: There were 29812 observations read from the data set GPDB.ZIPCODE.
      WHERE zip_class=' ';
NOTE: DATA statement used (Total process time):
      real time           0.59 seconds
      user cpu time       0.18 seconds
```

Output 5. Partial SAS Log, with basic partition options

From the **SASTRACE** log messages above, two threads were created; one thread contained five observations, the other contains 29807 observations - not a particularly efficient DBSLICE selection.

In the following example, a partition is created for each unique value of the TIMEZONE variable:

```
35         data _null_;
36           set GPDB.zipcode(DBSLICE=(
37                  "TIMEZONE='Alaska'"
38                  "TIMEZONE='Atlantic'"
39                  "TIMEZONE='Central'"
40                  "TIMEZONE='Eastern'"
41                  "TIMEZONE='Hawaii'"
42                  "TIMEZONE='Mountain'"
43                  "TIMEZONE='Pacific'" ));
44           where zip_class eq " ";
45         run;

GREENPL_8: Executed: on connection 0 22192 1427351164 no_name 0 DATASTEP
```

```
SELECT  "zip", "y", "x", "zip_class", "city", "state", "statecode",
"statename", "county", "countynm", "msa", "areacode", "areacodes", "timezone",
"gmtoffset", "dst", "poname", "alias_city", "alias_cityn", "city2",
"statename2"  FROM gp_stg.ZIPCODE  WHERE  ( "zip_class" = ' ' OR  "zip_class"
IS  NULL  ) AND  TIMEZONE='Alaska' FOR READ ONLY  22193 1427351164 no_name 0
DATASTEP
GREENPL_9: Executed: on connection 0 22196 1427351164 no_name 0 DATASTEP
SELECT  "zip", "y", "x", "zip_class", "city", "state", "statecode",
"statename", "county", "countynm", "msa", "areacode", "areacodes", "timezone",
"gmtoffset", "dst", "poname", "alias_city", "alias_cityn", "city2",
"statename2"  FROM gp_stg.ZIPCODE  WHERE  ( "zip_class" = ' ' OR  "zip_class"
IS  NULL  ) AND  TIMEZONE='Atlantic' FOR READ ONLY  22197 1427351164 no_name 0
DATASTEP
GREENPL_10: Executed: on connection 0 22200 1427351164 no_name 0 DATASTEP
SELECT  "zip", "y", "x", "zip_class", "city", "state", "statecode",
"statename", "county", "countynm", "msa", "areacode", "areacodes", "timezone",
"gmtoffset", "dst", "poname", "alias_city", "alias_cityn", "city2",
"statename2"  FROM gp_stg.ZIPCODE  WHERE  ( "zip_class" = ' ' OR  "zip_class"
IS  NULL  ) AND  TIMEZONE='Central' FOR READ ONLY  22201 1427351164 no_name 0
DATASTEP
GREENPL_11: Executed: on connection 0 22204 1427351164 no_name 0 DATASTEP
SELECT  "zip", "y", "x", "zip_class", "city", "state", "statecode",
"statename", "county", "countynm", "msa", "areacode", "areacodes", "timezone",
"gmtoffset", "dst", "poname", "alias_city", "alias_cityn", "city2",
"statename2"  FROM gp_stg.ZIPCODE  WHERE  ( "zip_class" = ' ' OR  "zip_class"
IS  NULL  ) AND  TIMEZONE='Eastern' FOR READ ONLY  22205 1427351164 no_name 0
DATASTEP
GREENPL_12: Executed: on connection 0 22208 1427351164 no_name 0 DATASTEP
SELECT  "zip", "y", "x", "zip_class", "city", "state", "statecode",
"statename", "county", "countynm", "msa", "areacode", "areacodes", "timezone",
"gmtoffset", "dst", "poname", "alias_city", "alias_cityn", "city2",
"statename2"  FROM gp_stg.ZIPCODE  WHERE  ( "zip_class" = ' ' OR  "zip_class"
IS  NULL  ) AND  TIMEZONE='Hawaii' FOR READ ONLY  22209 1427351164 no_name 0
DATASTEP
GREENPL_13: Executed: on connection 0 22212 1427351164 no_name 0 DATASTEP
SELECT  "zip", "y", "x", "zip_class", "city", "state", "statecode",
"statename", "county", "countynm", "msa", "areacode", "areacodes", "timezone",
"gmtoffset", "dst", "poname", "alias_city", "alias_cityn", "city2",
"statename2"  FROM gp_stg.ZIPCODE  WHERE  ( "zip_class" = ' ' OR  "zip_class"
IS  NULL  ) AND  TIMEZONE='Mountain' FOR READ ONLY  22213 1427351164 no_name 0
DATASTEP
GREENPL_14: Executed: on connection 0 22216 1427351164 no_name 0 DATASTEP
SELECT  "zip", "y", "x", "zip_class", "city", "state", "statecode",
"statename", "county", "countynm", "msa", "areacode", "areacodes", "timezone",
"gmtoffset", "dst", "poname", "alias_city", "alias_cityn", "city2",
"statename2"  FROM gp_stg.ZIPCODE  WHERE  ( "zip_class" = ' ' OR  "zip_class"
IS  NULL  ) AND  TIMEZONE='Pacific' FOR READ ONLY  22217 1427351164 no_name 0
DATASTEP
GREENPLUM:  Thread 5 contains 68 obs. 22219 1427351165 no_name 0 DATASTEP
GREENPLUM:  Thread 1 contains 61 obs. 22220 1427351165 no_name 0 DATASTEP
GREENPLUM:  Thread 7 contains 2720 obs. 22221 1427351165 no_name 0 DATASTEP
GREENPLUM:  Thread 6 contains 2128 obs. 22222 1427351165 no_name 0 DATASTEP
GREENPLUM:  Thread 4 contains 13187 obs. 22223 1427351165 no_name 0 DATASTEP
GREENPLUM:  Thread 3 contains 11516 obs. 22224 1427351165 no_name 0 DATASTEP
GREENPLUM:  Thread 2 contains 127 obs. 22225 1427351165 no_name 0 DATASTEP
GREENPLUM:  Threaded read enabled. Number of threads created: 7 22226
1427351165 no_name 0 DATASTEP
NOTE: There were 29807 observations read from the data set GPDB.ZIPCODE.
      WHERE zip_class=' ';
NOTE: DATA statement used (Total process time):
      real time           0.77 seconds
      user cpu time       0.20 seconds
```

Output 6. Partial SAS Log, with unique partition values

Again, although the read operation was spread across more threads (in this case, seven), these threads were not uniformly distributed.

The last of these examples attempts to rationalize partitions into three of (approximately) equal size

```
49          data _null_;
50            set GPDB.zipcode(DBSLICE=(
51                  "TIMEZONE in ('Eastern','Atlantic')"
52                  "TIMEZONE in ('Central','Mountain')"
53                  "TIMEZONE in ('Pacific','Alaska','Hawaii')"));
54            where zip_class eq " ";
55          run;

GREENPL_16: Executed: on connection 0 22232 1427351165 no_name 0 DATASTEP
SELECT  "zip", "y", "x", "zip_class", "city", "state", "statecode",
"statename", "county", "countynm", "msa", "areacode", "areacodes", "timezone",
"gmtoffset", "dst", "poname", "alias_city", "alias_cityn", "city2",
"statename2"  FROM gp_stg.ZIPCODE  WHERE  ( "zip_class" = ' ' OR  "zip_class"
IS  NULL  ) AND  TIMEZONE in ('Eastern','Atlantic') FOR READ ONLY  22233
1427351165
no_name 0 DATASTEP
GREENPL_17: Executed: on connection 0 22236 1427351165 no_name 0 DATASTEP
SELECT  "zip", "y", "x", "zip_class", "city", "state", "statecode",
"statename", "county", "countynm", "msa", "areacode", "areacodes", "timezone",
"gmtoffset", "dst", "poname", "alias_city", "alias_cityn", "city2",
"statename2"  FROM gp_stg.ZIPCODE  WHERE  ( "zip_class" = ' ' OR  "zip_class"
IS  NULL  ) AND  TIMEZONE in ('Central','Mountain') FOR READ ONLY  22237
1427351165
no_name 0 DATASTEP
GREENPL_18: Executed: on connection 0 22240 1427351165 no_name 0 DATASTEP
SELECT  "zip", "y", "x", "zip_class", "city", "state", "statecode",
"statename", "county", "countynm", "msa", "areacode", "areacodes", "timezone",
"gmtoffset", "dst", "poname", "alias_city", "alias_cityn", "city2",
"statename2"  FROM gp_stg.ZIPCODE  WHERE  ( "zip_class" = ' ' OR  "zip_class"
IS  NULL  ) AND  TIMEZONE in ('Pacific','Alaska','Hawaii') FOR READ ONLY  22241

GREENPLUM:  Thread 3 contains 2849 obs. 22243 1427351165 no_name 0 DATASTEP
GREENPLUM:  Thread 2 contains 13644 obs. 22244 1427351166 no_name 0 DATASTEP
GREENPLUM:  Thread 1 contains 13314 obs. 22245 1427351166 no_name 0 DATASTEP
GREENPLUM:  Threaded read enabled. Number of threads created: 3 22246
1427351166 no_name 0 DATASTEP
NOTE: There were 29807 observations read from the data set GPDB.ZIPCODE.
      WHERE zip_class=' ';
NOTE: DATA statement used (Total process time):
      real time           0.51 seconds
      user cpu time       0.21 seconds
```

Output 7. Partial SAS Log, with consolidated partition values

From these results, the following can be observed:

- In each case, the USER CPU TIME is (essentially) identical, as this is the total time taken by SAS to process the user code, regardless of how many threads are (or are not) used.
- However, by testing various DBSLICE options, and using **SASTRACE** to examine thread results, the REAL TIME to process the data can be reduced by spreading the workload over multiple simultaneous threads. *(For this reason, there may be instances when the REAL TIME could be faster than the USER CPU TIME!)*

It important to emphasise that in order to determine effective partition settings, the user must have a good-to-thorough knowledge not only of the data values in the table and their distribution, but also of the architecture connecting the SAS engine & the database. **SASTRACE** is an ideal way to test & examine different partition settings.

EXAMINING DATABASE TABLE CREATION

Some databases (such as Greenplum & Teradata) impose specific requirements on tables, such as having to define a primary index. If not specified, the first field of a created table is used as the Primary Index; if the first field happens to be a natural candidate for a Primary Index (for example, User ID) then the data will be stored in a (reasonably) distributed method within the database.

However, beware if the first field is NOT a particularly good candidate for a Primary Index, for example:

- Gender field in an Armed Forces table (most likely to skew towards males)
- Year field (followed by Month, Day, etc) in a "Results2014" table – common in a schema of yearly tables.

In these cases, a poorly-distributed table may result in poor table performance; for example, a multi-threaded read on a table with a skewed distribution may quickly return results from most threads, but then be delayed by a single poorly-performing thread.

SASTRACE will echo the table creation SQL code to the SAS log, including index creation.

Below is sample code to upload a SAS table into a Greenplum database schema. **SASTRACE** has been set to `',,,db'` to display a summary of all generated SQL code:

```
options sastrace=",,,db" sastraceloc=saslog;
data GPDB.zipcode2;
      set sashelp.zipcode(obs=100);
run;
```

Output 8. Sample SAS code to create a Greenplum table

Below is the resultant SAS log; relevant **SASTRACE** messages have been highlighted:

```
23          options sastrace=",,,db" sastraceloc=saslog;
24          data GPDB.zipcode2;
25            set sashelp.zipcode(obs=100);
26          run;

<< Some rows cut..>>

NOTE: SAS variable labels, formats, and lengths are not written to DBMS tables.
  1834 1427373451 no_name 0 DATASTEP
GREENPL_262: Executed: on connection 2 1835 1427373451 no_name 0 DATASTEP
CREATE TABLE gp_stg.ZIPCODE2 (ZIP DOUBLE PRECISION,Y NUMERIC(11,6),X
NUMERIC(11,6),ZIP_CLASS VARCHAR(1),CITY VARCHAR(35),STATE
DOUBLE PRECISION,STATECODE VARCHAR(2),STATENAME VARCHAR(25),COUNTY DOUBLE
PRECISION,COUNTYNM VARCHAR(25),MSA DOUBLE
PRECISION,AREACODE DOUBLE PRECISION,AREACODES VARCHAR(12),TIMEZONE
VARCHAR(9),GMTOFFSET DOUBLE PRECISION,DST VARCHAR(1),PONAME
VARCHAR(35),ALIAS_CITY VARCHAR(300),ALIAS_CITYN VARCHAR(300),CITY2
VARCHAR(35),STATENAME2 VARCHAR(25)) DISTRIBUTED RANDOMLY  1836
```

Output 9. Resultant (partial) log of Greenplum table creation code.

OTHER USES OF SASTRACE

The time allotted to present this paper does not allow for a more detailed exploration of **SASTRACE** functionality; in particular, when performing SQL joins of tables from different data sources, such as a query extracting data from a Greenplum table, an Oracle table and a SAS table. Again, the information sent to the SAS log by **SASTRACE** is highly beneficial in gaining a thorough understanding of relationship between all data sources in a query.

Another trap I have witnessed at previous sites is when organisations migrate their SAS data tables to their strategic data store (Teradata, Oracle, etc). The first step is often a "quick fix" to "lift-and-shift" the various SAS libraries to different schemas in a relational database, and re-define the libnames located in autoexec.sas (or metadata), but with little or no other change to the SAS code or the data structures & relationships.

As a result, code which previously ran efficiently on SAS data, still runs without errors on relational data, but with much poorer performance. This performance is often blamed on architecture, I/O bottlenecks, but a simple view of **SASTRACE** logs would reveal code which does not take into effect the “disconnect” between the SAS engine and the relational data.

The following output is a “pseudo-code” sample to add more fields to a SAS table:

```
libname corpdb BASE "/CorpData";
data corpdb.mynewtable;
       merge corpdp.myoldtable(in=In1) work.newdata(keep=Employee_ID NewStats);
       by Employee_ID;
       if In1;
run;
```

Output 10. Pseudo-code of a DATA STEP Merge of SAS tables.

The following output is similar “pseudo-code”, but to a Teradata table:

```
libname corpdb TERADATA /* Teradata options go here.. */;
data corpdb.mynewtable;
       merge corpdp.myoldtable(in=In1) work.newdata(keep=Employee_ID NewStats);
       by Employee_ID;
       if In1;
run;
```

Output 11. Pseudo-code of a DATA STEP Merge of Teradata & SAS tables.

In the example described above, the original SAS code updated the “NewTable” in 60-75 minutes. However, the migrated code run on a much more powerful database then took more than 8 hours to complete!

SASTRACE quickly revealed that individual rows are being drawn down from the relational database, merged, and then returned to the database one row at a time – an extremely inefficient process.

The solution applied was to modify the code to upload the NEWDATA table to the database using FASTLOAD techniques, and then perform an in-database SQL join. The new code ran in roughly 20 minutes – certainly the intended improvement on the original SAS code of 60-75 minutes, and infinitely better than the 8+ hours of the “first cut” of migrated code.

CONCLUSIONS

The transparency of SAS/ACCESS is tempting, but SAS developers need to be wary of how SAS/ACCESS executes user-written code on relational data.

SAS and relational databases can co-exist in a mutually beneficial relationship, offering the “best of both worlds” to SAS developers. However, more likely than not, there are additional rules, requirements & constraints imposed on relational tables which are not imposed on SAS data; ignoring them can adversely impact performance.

My common mantra to team members when running SAS queries on relational data is "**Know Your Database. Respect Your Database!**".

SASTRACE is an invaluable tool in maintaining a healthy relationship between your SAS code and your relational data.

REFERENCES & RECOMMENDED READING

SAS ACCESS 9.4 for Relational Databases – Reference

SAS ACCESS 9.4 Interface to PC Files – Reference

SAS Communities website https://communities.sas.com

CONTACT INFORMATION

I welcome your feedback and any questions. Please feel free to contact me at:

Name: **Andrew Howell**
Organization: **ANJ Solutions Pty Ltd**
Address: **PO Box 765, Macleod VIC 3085 AUSTRALIA**
Phone: **+ 61 407 898 513**
Email: **info@anjsolutions.com.au**
Twitter: **@AndrewAtANJ**
Skype: **AndrewAtANJ**
LinkedIn: **http://au.linkedin.com/in/howellandrew/**
SAS Communities: **https://communities.sas.com/people/AndrewHowell**

I also strongly encourage ongoing discussion in the SAS Communities website.