Paper 081-2013

Need for Speed - Boost Performance in Data Processing with SAS/Access® Interface to Oracle

Svein Erik Vrålstad, Knowit Decision Oslo, Norway

ABSTRACT

Big Data is engulfing us. The expectations of users increase, and analytics is getting more and more advanced. Timely data and fast results have never had greater value. In data warehousing, analytics, data integration, and reporting, there is an ever-growing need for speed. When operating in environments where performance is of importance, it is of great value to fully understand the interaction between the different components of the environment. Hence, the importance of in-database execution is accelerating. To know when to let SAS process data, and when to use Oracle to perform the task is then of great value. This paper explores ways to achieve substantial gains in performance when processing (read, transform, calculate, and write) data in an effective manner.

The audience for this paper is system developers and architects working with SAS in collaboration with data held in Oracle, where performance for at least one of the reasons stated above is of importance. Some of the tips will also prove to have value when working with SAS on other databases, for example when utilizing SAS Connect. The paper will also prove to be of importance to DBA´s with responsibilities in environments where SAS is one of several platforms consuming resources of his data warehouse.

INTRODUCTION

There are often a number of different data source formats that SAS developers have to handle. SAS has provided us with SAS/ACCESS software to make this challenge manageable. This paper will look into some options that will prove to be of value when connecting to database management systems (DBMS). The SAS/ACCESS options available depend on the SAS version installed and the DBMS to connect to. What is described here has been tested on a combination of SAS 9.2 and Oracle V10g Enterprise Edition. Most hints are also applicable on other DBMS, but you should always study the guidelines for your system and do your own investigations to get the most out of the SAS/ACCESS software.

This paper is not a complete guide to the world of SAS/ACCESS, but it will give some insights on the issue.

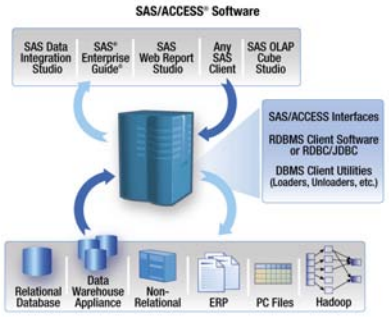

Figure 1 - SAS/ACCESS® Software

CONNECTING TO THE DBMS

SAS/ACCESS is the software to use to connect to various DBMS. SAS/ACCESS provides two main alternative methods to connect: SAS/ACCESS Libname and the SAS Pass-Through Facility.

SAS/ACCESS LIBNAME

The libname statements to use are quite familiar to the SAS developer. Here is an example connecting to an Oracle server.

```
libname ora_lib oracle user=user password=pwd path=server <libname-options>;
```

With a libname statement like this you could use your ordinary SAS toolbox of DATA Step and PROCs almost as if you were working with ordinary SAS DATA sets. SAS/ACCESS translates the commands into SQL which the DBMS can process. This translation is done with various degrees of success as we will see later. Keep in mind the SAS version used here, as every new release might be still better in handling the necessary translations.

SAS SQL PASS-THROUGH FACILITY

The Pass-Through Facility is another way to establish a connection to the external database. The syntax is slightly different, it looks almost like the LIBNAME statement and PROC SQL combined.

```
proc sql;
    connect to oracle (user=user password=pwd path=server <pass-through options>);
    create table any_table as
           select * from connection to oracle (
```

```
                    select * from ora_table);
    disconnect from oracle;
quit;
```

or for non-query sql

```
proc sql;
    connect to oracle (user=user password=pwd path=server <pass-throug options>);
          execute (delete from ora_table) by oracle;
    disconnect from oracle;
quit;
```

Pass-Through SQL is sent as-is directly to the DBMS, thus requiring DMBS native SQL.

WHICH ONE TO CHOOSE?

Which method to choose depends much on the circumstance and the previous experience of the developer. However, there are, in my opinion, some pros and cons with these approaches. In short:

Use SQL Pass-through when

- there is a need for DBMS specific SQL
- you need total control of the SQL processed by the DBMS
- SAS/ACCESS does not translate SAS steps to efficient SQL

Use the SAS/ACCESS libname when

- the code should be easy to create and maintained by developers more used to SAS syntax
- the programs should be able to connect to various DBMS with only minor modifications

MODS FOR MAXIMUM PERFORMANCE

In the optimization process there are many dimensions to consider. Some main performance boosters could be:

- Minimize the volume of data being transferred between SAS and the DMBS
- Fast transmission of necessary data
- Optimizing the DBMS processing

KEEP TRANSFERRED DATA AT A MINIMUM

- Get as few records as possible by fetching minimized subsets of data
- Limit the number of columns, either in the select statement of the PROC SQL or by use of data set keep/drop options
- Make the DBMS do the processing, in-database execution

Make the DBMS do the processing

For efficiency it is often optimal to leave calculations and especially subsetting to the DBMS. This might sound pretty straightforward; in reality it proves not always to be that simple.

Choosing SQL pass-through, you will have total control of the SQL the DBMS will process. The main drawback is that you have to use the specific DBMS-compatible SQL. If you want to write ordinary SAS DATA Step/PROC SQL and SAS functions there are some challenges to be aware of.

Be aware of the SAS functions

SAS/ACCESS will always try to translate any SAS PROCs and SAS DATA Steps to native DBMS language. Often SAS is doing a good job, but it's not perfect. If SAS maps correctly to a DBMS and the SQL are totally processed by the DBMS, it is called *Implicit SQL pass-through*. Use of certain SAS functions will prevent efficient pass-through and force SAS to forget all about subsetting. Instead SAS will fetch **all** records from the DBMS, and next do the processing of the SAS functions locally in SAS on the entire complete data set. This could have huge and undesired negative impact on performance.

How to avoid local processing in SAS

To always avoid SAS functions that SAS/ACCESS can't transform is a good rule of thumb. Using those special SAS facilities, the recommended solution is to return only the desired subset of data to the SAS server in the first place,

and then do the processing in SAS in a later step.

Which functions is SAS/ACCESS able to transform to corresponding DBMS SQL? The SAS/ACCESS library of compatible functions is expanded continuously with every new SAS version. The number of functions available for Implicit SQL Pass-Through depends on the version of SAS and the DBMS. SAS offers a list of functions that are supported by different DBMS.

Please look at the chapter "Passing Functions to the DBMS Using PROC SQL" in SAS/ACCESS® 9.2 for Relational Databases: Reference, Fourth Edition for SAS 9.2 implementation:

http://support.sas.com/documentation/cdl/en/acreldb/63647/HTML/default/viewer.htm#a001630468.htm

Reveal the secrets of SAS/DBMS communication

Being updated on the list of "admissible" SAS functions makes the optimizing process a lot easier. But sometimes it proves to be of significant value to actually know what SAS is doing. Normally it is of little help studying the ordinary log to be assured that the SQL sent to the DBMS is efficient. The SAS macro language offers some handy options to enhance the logging in order to reveal the effect of the macro facility processing. There exists a similar option for SAS/ACCESS.

The SAS option SASTRACE is here to shed light on what kind of pass-through is actually performed. With this option you could learn from the log what SQL statements SAS actually has sent into the DBMS for processing. My guess is that you will be surprised once in a while.

The SASTRACE option

Depending of the use of settings, SASTRACE can give different output to the log. This paper will only look at one useful feature:

```
OPTIONS SASTRACE=',,,d' SASTRACELOC=SASLOG NOSTSUFFIX;

SASTRACE = ',,,d' = Print SQL statements sent to the DBMS
SASTRACELOC = Set the log as destination for the output
NOSTSUFFIX = Suppress less important information
```

To stop the enhanced output to the log:

```
OPTIONS SASTRACE = OFF
```

SASTRACE also has other options like ',,,sd' (or s alone) which will give additional timing information to the log. I recommend the SAS support pages for a full documentation.

SASTRACE examples

Compare output from different connection methods

The table below shows some simple SASTRACE outputs to the log.

Ex 1.

Pass-through SQL Facility

PROC SQL

```
proc sql;
connect to oracle(user=usr
password=pwd path=serv);
create table cust as
  select * from connection
to oracle(
    select * from
customers);

ORACLE_1: Prepared: on
connection 1
select * from customers

ORACLE_2: Executed: on
connection 1
SELECT statement  ORACLE_1
```

SAS/ACCESS Libname

PROC SQL

```
proc sql;
create table cust as
select * from
ora_sas.customers;

ORACLE_1: Prepared: on
connection 1
SELECT * FROM customers

ORACLE_2: Executed: on
connection 1
SELECT statement  ORACLE_1
```

DATA STEP

```
ORACLE_1: Prepared: on
connection 1
SELECT * FROM customers
data cust;
  set ora_sas.customers;
run;

ORACLE_2: Executed: on
connection 1
SELECT statement  ORACLE_1
```

Ex 2.

Pass-through SQL Facility

PROC SQL

```
proc sql;
  connect to
oracle(user=ora_sas
password=ora_pwd
path=ora_serv);
  create table cust as
  select * from connection
to oracle(select cust_id,
cust_name from customers);
ORACLE_1: Prepared: on
connection 1
select cust_id, cust_name
from customers
ORACLE_2: Executed: on
connection 1
SELECT statement  ORACLE_1
```

SAS/ACCESS Libname

PROC SQL

```
proc sql;
  create table cust as
  select cust_id, cust_name
from ora_sas.customers;
ORACLE_1: Prepared: on
connection 1
SELECT * FROM customers
ORACLE_2: Prepared: on
connection 1
SELECT  "CUST_ID",
"CUST_NAME" FROM customers
ORACLE_3: Executed: on
connection 1
SELECT statement  ORACLE_2
```

DATA STEP

```
ORACLE_1: Prepared: on
connection 1
SELECT * FROM customers
data cust;
  set
ora_sas.customers(keep=cust
_id cust_name);
run;
ORACLE_2: Prepared: on
connection 1
SELECT  "CUST_ID",
"CUST_NAME" FROM customers
ORACLE_3: Executed: on
connection 1
SELECT statement  ORACLE_2
```

Quite similar SQL produced by all methods. So let's leave the Pass-Through Facility for now and compare the effect of a couple of queries using SAS/ACCESS.

Using SAS functions

In these examples we have used a quite small Oracle table with approx. one million records.

The TRIM/STRIP functions

```
ORACLE_1: Prepared: on connection 0
SELECT * FROM ORA.CUSTOMERS

19 data customers;
20  set ora.customers;
21  where trim(cust_id)='1234';
22 run;

ORACLE_2: Prepared: on connection 0
SELECT  "CUST_ID", "CUST_NAME", "ADDRESS",
"POST_CODE", "TOWN", "CONTACT", "CUST_DATE"
FROM ORA.CUSTOMERS

ORACLE_3: Executed: on connection 0
SELECT statement ORACLE_2

NOTE: There were 1 observations read from
the data set ORA.CUSTOMERS.
WHERE TRIM(cust_id)= '1234';
NOTE: The data set WORK.CUSTOMERS has 1
observations and 7 variables.
NOTE: DATA statement used (Total process
time):
      real time          43.94 seconds
      cpu time           4.41 seconds
```

```
ORACLE_1: Prepared: on connection 0
SELECT * FROM ORA.CUSTOMERS

19 data customers;
20  set ora.customers;
21  where strip(cust_id)='1234';
22 run;

ORACLE_2: Prepared: on connection 0
SELECT  "CUST_ID", "CUST_NAME", "ADDRESS",
"POST_CODE", "TOWN", "CONTACT", "CUST_DATE"
FROM ORA.CUSTOMERS
WHERE  ( TRIM("CUST_ID") = '964052220' )

ORACLE_3: Executed: on connection 0
SELECT statement ORACLE_2

NOTE: There were 1 observations read from
the data set ORA.CUSTOMERS.
      WHERE STRIP(cust_id)= '1234';
NOTE: The data set WORK.CUSTOMERS has 1
observations and 7 variables.
NOTE: DATA statement used (Total process
time):
      real time          0.37 seconds
      cpu time           0.00 seconds
```

You probably would expect the trim function to be easily translated into the Oracle counterpart. But the log tells us a different story. The WHERE clause is not sent to Oracle and the entire data set is pushed to SAS, leaving the processing of the function to SAS. According to the right hand log above, replacing TRIM with the STRIP function is much more efficient. Maybe surprisingly, this time SAS choose to translate the strip to the trim function in the SQL transmitted to Oracle.

The DATEPART function

```
ORACLE_1: Prepared: on connection 0
SELECT * FROM ORA.CUSTOMERS
19 data customers;
20  set ora.customers;
21  where datepart(cust_date)='01JUN2012'd;
22 run;

ORACLE_2: Prepared: on connection 0
SELECT  "CUST_ID", "CUST_NAME", "ADDRESS",
"POST_CODE", "TOWN", "CONTACT", "CUST_DATE"
FROM ORA.CUSTOMERS

ORACLE_3: Executed: on connection 0
SELECT statement  ORACLE_2



NOTE: There were 43 observations read from
the data set ORA.CUSTOMERS.
WHERE DATEPART(cust_date)='01JUN2012'D;
NOTE: The data set WORK.CUSTOMERS has 43
observations and 7 variables.
NOTE: DATA statement used (Total process
time):
      real time           41.53 seconds
      cpu time            4.39 seconds
```

```
ORACLE_1: Prepared: on connection 0
SELECT * FROM ORA.CUSTOMERS
19 data customers;
20  set ora.customers;
21  where cust_date='01JUN2012:00:00:00'dt;
22 run;

ORACLE_2: Prepared: on connection 0
SELECT  "CUST_ID", "CUST_NAME", "ADDRESS",
"POST_CODE", "TOWN", "CONTACT", "CUST_DATE"
FROM ORA.CUSTOMERS
WHERE  ("CUST_DATE" =TO_DATE('
01JUN2012:00:00:00','DDMONYYYY:HH24:MI:SS','
NLS_DATE_LANGUAGE=American') )
ORACLE_3: Executed: on connection 0
SELECT statement  ORACLE_2

NOTE: There were 43 observations read from
the data set ORA.CUSTOMERS.
WHERE cust_date='  01JUN2012:00:00:00'DT;
NOTE: The data set WORK.CUSTOMERS has 43
observations and 7 variables.
NOTE: DATA statement used (Total process
time):
      real time           0.69 seconds
      cpu time            0.00 seconds
```

Use of the SAS DATEPART function leads to a similar decrease in performance. The logs explain why; even this time the WHERE statement is never transmitted to the Oracle Server. One should always proceed with great care when using SAS date- and time-functions when working with a DBMS.

Remove the parts created by the SASTRACE option and the log output would be almost equal, leaving the developer with very little explanation of the substantial differences in processing time. Working with SAS/ACCESS LIBNAME and large quantities of data, it is important to do thorough tests for optimal handling of data.

We have learned that SASTRACE is a very useful option. Should we then keep this option on always? I would say no since the side effects could be quite annoying.

Be aware of the SASTRACE pitfalls

Now a few words on the SASTRACE and non-query SQL's. With standard SAS buffer handling (more about this later) a delete or update query on an Oracle-table will output about three lines to the log for every record being processed. Inserts will request a little less space, with the same output for every 10 records. This means huge logs if large tables are to be processed. Below are a couple of examples, using a plain DATA Step. Applied on an Oracle table a DATA Step will first make a SQL CREATE TABLE followed by an INSERT INTO.

This first part is shared by all runs:

```
ORACLE_1: Prepared: on connection 0
SELECT * FROM customers
19         data ora_sas.customers_new;
20           set ora_sas.customers(obs=xx);
21         run;
ORACLE_2: Prepared: on connection 1
SELECT * FROM ora_sas.customers_new
NOTE: SAS variable labels, formats, and lengths are not written to DBMS tables.
ORACLE_3: Executed: on connection 2
CREATE TABLE customers2(CUST_ID VARCHAR2 (20),CUST_NAME VARCHAR2 (200),ADDRESS VARCHAR2
(60),POST_CODE VARCHAR2 (5),TOWN VARCHAR2 (50), CONTACT VARCHAR2 (50),CUST_DATE DATE)
ORACLE_4: Executed: on connection 0
SELECT statement  ORACLE_1
ORACLE_5: Prepared: on connection 2
INSERT  INTO ora_sas.customers_new
(CUST_ID,CUST_NAME,ADDRESS,POST_CODE,TOWN,CONTACT,CUST_DATE) VALUES
(:CUST_ID,:CUST_NAME,:ADDRESS,:POST_CODE,:TOWN,:CONTACT,TO_DATE(:CUST_DATE,'DDMONYYYY:HH24:MI:
SS','NLS_DATE_LANGUAGE=American'))
```

set ora_sas.customers(**obs=10**)

```
ORACLE_6: Executed: on connection 2
INSERT statement  ORACLE_5
NOTE: There were 10 observations read from the data set ora_sas.customers.
ORACLE:  *-*-*-*-*-*-* COMMIT *-*-*-*-*-*-*
```

```
NOTE: The data set ora_sas.customers_new has 10 observations and 7 variables.
ORACLE:  *-*-*-*-*-*-* COMMIT *-*-*-*-*-*-*
```

set ora_sas.customers(**obs=100**)

```
ORACLE_6: Executed: on connection 2
INSERT statement  ORACLE_5
…
…
ORACLE_15: Executed: on connection 2                ← 10 Executes
INSERT statement  ORACLE_5

NOTE: There were 100 observations read from the data set ora_sas.customers.
ORACLE:  *-*-*-*-*-*-* COMMIT *-*-*-*-*-*-*
NOTE: The data set ora_sas.customers_new has 100 observations and 7 variables.
ORACLE:  *-*-*-*-*-*-* COMMIT *-*-*-*-*-*-*
```

set ora_sas.customers(**obs=10000**)

```
ORACLE_6: Executed: on connection 2
INSERT statement  ORACLE_5
…
…
ORACLE:  *-*-*-*-*-*-* COMMIT *-*-*-*-*-*-*.        ← Multiple commits
…
…
ORACLE_1005: Executed: on connection 2              ← 1000 Executes
INSERT statement  ORACLE_5

ORACLE:  *-*-*-*-*-*-* COMMIT *-*-*-*-*-*-*.        ← Multiple commits

NOTE: There were 10000 observations read from the data set ora_sas.customers.
ORACLE:  *-*-*-*-*-*-* COMMIT *-*-*-*-*-*-*
NOTE: The data set ora_sas.customers_new has 10000 observations and 7 variables.
ORACLE:  *-*-*-*-*-*-* COMMIT *-*-*-*-*-*-*
```

The last log has more than 4000 lines. We will get an EXECUTE output for every 10 records, and a COMMIT output for every 1000. It's easy to imagine the kind of log if there were millions or billions of records. Running on 1 million records created a 450 000 line log, with a size about 9 MB. Deleting the same amount of DBMS data with an ordinary PROC SQL statement would mean an enormous log file, since delete and update produces one EXECUTE output to the log for every single record deleted.

Conclusion: Always limit the number of records to process using OBS= (INOBS in PROC SQL) when developing or debugging with SASTRACE on.

FAST TRANSMITTING OF DATA BETWEEN SAS AND THE DBMS

Even if you succeed in limiting the amount of data that has to be transferred, there will still often be an abundance of data to be shuffled between the systems. SAS has some options that are quite useful to create a more efficient data flow.

Buffering

SAS/ACCESS

Buffering means that data is stored in memory and transferred in blocks from one system to another. The default numbers of records being transferred simultaneously vary depending on the type of process:

- Read - 250 records
- Insert - 10 records
- Update/Delete - 1 record

Most often there are quite a lot to gain manipulating the default settings. Quite a few SAS documents states that the maximum buffer size is 32 767, but I recall reading somewhere that the limit is 2,147,483,648 records. Fact is that

> "SAS allows the maximum number of rows that the DBMS allows. The optimal value for this option varies with factors such as network type and available memory. You might need to experiment with different values in order to determine the best value for your site."
>
> From SAS/ACCESS® 9.2 for Relational Databases: Reference, Fourth Edition,

http://support.sas.com/documentation/cdl/en/acreldb/63647/HTML/default/viewer.htm#a001342321.htm

Nevertheless, using the highest value possible is not always, if ever, the best option. One will always have to do some testing in the relevant environment to find the optimal value.

Using the buffers

The buffer size might be manipulated on both the LIBNAME and the DATA Step/PROC SQL level. One or more buffer settings could be changed.

```
libname ora_lib oracle user=username password=password path=server_name
readbuff=10000 insertbuff=10000 updatebuff=10000;

set ora_lib.ora_table(readbuff=10000);

select * from  ora_sas.customers(readbuff=10000);
```

Using insertbuff (default buffer size=10)**:**

```
libname ora_sas oracle user=user_name
password=pwd path=server;
```

```
NOTE: Appending work_table to
ora_lib.ora_table
NOTE: There were 3190735 observations read
from the data set work_table
NOTE: 3190735 observations added.
NOTE: The data set ora_lib.ora_table has .
observations and 14 variables.
NOTE: PROCEDURE APPEND used (Total process
time):
      real time           6:46.93
      cpu time            31.85 seconds
```

```
libname ora_sas oracle user=user_name
password=pwd path=server
insertbuff=32767;
```

```
NOTE: Appending work_table to
ora_lib.ora_table
NOTE: There were 3190735 observations read
from the data set work_table
NOTE: 3190735 observations added.
NOTE: The data set ora_lib.ora_table has .
observations and 14 variables.
NOTE: PROCEDURE APPEND used (Total process
time):
      real time           1:17.14
      cpu time            29.93 seconds
```

Using updatebuff (default buffer size=1)**:**

```
libname ora_sas oracle user=user_name
password=pwd path=server;
```

```
NOTE: 3190735 rows were deleted from
ora_lib.ora_table
NOTE: PROCEDURE SQL used (Total process
time):
      real time           37:36.11
      cpu time            1:44.44
```

```
libname ora_sas oracle user=user_name
password=pwd path=server
updatebuff=32767;
```

```
NOTE: 3190735 rows were deleted from
ora_lib.ora_table
NOTE: PROCEDURE SQL used (Total process
time):
      real time           1:56.78
      cpu time            12.53 seconds
```

Updatebuff has huge effect on performance because SAS normally will fetch the whole data set to get the records going to be deleted, and then delete one row in the DBMS at a time.

NOTE: You could also use the **DBIDIRECTEXEC** or a standard SQL Pass-through to avoid slow deletion.

DBCOMMIT

Commit-handling is another issue to consider. Performing a commit means that data is permanently written to the DBMS. The DBCOMMIT option controls how many rows will be processed between the commits. The default setting is 1000 when creating a table and inserting data in a single step (DATA Step/PROC SQL CREATE TABLE). Creating a table with 1 million records, 1000 commits will be performed. In steps doing inserts, updates and deletes, the default is 0, which means the commit will be done only once after the step is completed.

The option might be set on the libname level or and in DATA/PROC SQL steps.

```
libname ora_lib oracle user=user password=pwd path=server dbcommit=0;

data ora_lib.ora_table_new(dbcommit=0);
```

One might be tempted to always set the value to 0. But it should be used with caution, as this could have a negative impact on performance on the Oracle side. As usual, it should be thoroughly tested before being implemented.

SQL Pass-Through

Likewise the SQL Pass-Through facility offers a read buffer that might improve performance.

```
connect to oracle (user=user password=pwd path=server buffsize=1000);
```

OPTIONS DBIDIRECTEXEC

Fast transmission between SAS and the DBMS is a good thing. But it´s even better if you could avoid moving data at all. With DBIDIRECTEXEC, delete and create table queries will be passed directly to the DBMS for processing which is much more efficient. In SAS 9.2 this does not apply to DATA Steps.

```
options NODBIDIRECTEXEC;
```

```
proc sql;
 create table ora_sas.customers_new as
 select * from ora_sas.customers;
quit;

NOTE: SAS variable labels, formats, and lengths are not written to DBMS tables.
NOTE: Table ora_sas.customers_new created, with 1082070 rows and 7 columns.
NOTE: PROCEDURE SQL used (Total process time):
      real time           8:41.71
      cpu time            12.90 seconds
```

```
options DBIDIRECTEXEC;
```

```
NOTE: PROCEDURE SQL used (Total process time):
      real time           7.20 seconds
      cpu time            0.01 seconds
```

Other SAS performance boosters

Bulkloading

Bulkloading are said to be the fastest way to transmit large amounts of data to the DBMS. In Oracle, bulkloading means taking advantage of the Oracle's SQL* loader. It might be a little more complicated to use than the buffer options, and do not always give better performance than fine-tuning the buffers through iterative testing of different settings. There are lots of good papers discussing this option, among those are *Methods of Storing SAS® Data into Oracle Tables* and *Five Ways to Speed Up Your Data Loading Using SAS/ACCESS® for Relational Databases*.

```
data ora_lib.ora_table(BULKLOAD=YES BL_DELETE_DATAFILE=YES);
```

Slicing

Setting the DBSLICE options you could specify the numbers of threaded reads and the number of connections to use. Adding the 't'-part to the SASTRACE will output threading information to the log.

```
options sastrace=',,t,d' sastraceloc=saslog nostsuffix;
```

DBSLICEPARM

This option uses table partitions if available, or the modulus function to create separate parallel queries to the table. It is available at multiple levels from configuration files to the LIBNAME and data set options.

```
option dbsliceparm=(all,4);

libname ora_lib oracle user=user password=pwd path=server dbsliceparm=(all,4);

set ora_lib.ora_table(dbsliceparm=(all,4));

select * from  ora_sas.customers(dbsliceparm=(all,4));
```

Taking advantage of a larger number of threads could save some time as these numbers undoubtedly states:

dbsliceparm not set 116.36 seconds
dbsliceparm=(all, 2) 58.97 seconds
dbsliceparm=(all, 4) 40.30 seconds
dbsliceparm=(all, 8) 31.28 seconds

dbsliceparm=(all, 8):

```
ORACLE_1: Prepared: on connection 0
SELECT * FROM ora_table
ORACLE:  DBSLICEPARM option set and 8 threads were requested
ORACLE:  No application input on number of threads.
ORACLE_2: Prepared: on connection 0
SELECT   "CUST_ID", "CUST_NAME", …. FROM CUSTOMERS  WHERE
ABS(MOD("CUSTOMER_ID", 8))=0
ORACLE: *** XOT CONNECTION: 1 ***
ORACLE: SQL executed on XOT CONNECTION 1: SELECT   "CUST_ID", "CUST_NAME", …. FROM CUSTOMERS
WHERE  ABS(MOD("CUSTOMER_ID", 8))=0
ORACLE: *** XOT CONNECTION: 8 ***
ORACLE: SQL executed on XOT CONNECTION 1: SELECT   "CUST_ID", "CUST_NAME", …. FROM CUSTOMERS
WHERE  ABS(MOD("CUSTOMER_ID", 8))=7
ORACLE:  Thread 7 contains 668364 obs.
ORACLE:  Thread 2 contains 881503 obs.
ORACLE:  Thread 1 contains 882723 obs.
ORACLE:  Thread 8 contains 668334 obs.
ORACLE:  Thread 6 contains 1024146 obs.
ORACLE:  Thread 3 contains 747029 obs.
ORACLE:  Thread 5 contains 1050116 obs.
ORACLE:  Thread 4 contains 757157 obs.
ORACLE:  Threaded read enabled. Number of threads created: 8
NOTE: Table WORK.CUSTOMERS created, with 6679372 rows and 20 columns.
```

DBSLICE

With this option you might specify the where-clause to perform a threaded read like

```
select * from customers where(dbslice="upper(gender)='M'" "upper(gender)='F'"))
```

Using this you have to be very cautious and be sure all possible subsets are included. The above would exclude all customers where gender is not set.

STREAMLINING THE DBMS PROCESSING

Some Oracle optimizing possibilities

- Partitioning
- Updated statistics
- Optimizer
- Hints and Explain plans

Some of the topics covered here will generally be used by the DBA to tune the DBMS environment. But at least using hints could be a valuable and easily applied tool for the SAS developer.

Using Oracle hints

In short, using hints means that you force an override on the explain plan that the Oracle optimizer has found to be the most efficient. In addition, you are allowed to make Oracle do the processing in parallel. There could be many reasons why Oracle chooses a less optimal way like non-optimal use of index or nested loops. Statistics not being updated could be one reason. To learn more about this, I recommend the Oracle documentation.

Oracle hints should always be placed directly after the Select word in the query. The syntax is

```
/*+hint(s)*/ (the + part is crucial)
```

Note that neither Oracle nor SAS will return any error messages if the syntax in between the /**/ is incorrect, it simply will be ignored.

To be a hint wizard is not necessary, there are a number of low hanging fruits in this field; Even without digging deep into the secrets of Oracle, a number of simple use of hints could make a big difference on the speed of processing.

Here are some examples:

- **Select /*+full(a)*/ from table a** - full table scan, no index
- **Select /*+no_index(a)*/ from table a** - do not use index
- **Select /*+parallel(a, n)*/ from table a** - applying parallel processing. n=number of parallels. NOTE! Setting a high number of parallels might have negative impact on performance
- **Select /*+ordered*/ from table a** - Means that the SQL will be processed in the order specified in the join. Might sometimes be effective to prevent Cartesian products
- **Select /*+use_hash(a)*/ from table a** - To avoid nested loops

In the following examples, I the SAS syntax using hints is shown. As parallels usually give a performance boost when processing large amounts of data, we will concentrate on parallels in the examples. Be aware that using many parallels might have a negative impact on performance and consumes processor resources in a shared environment. Being too greedy will not make you the DBA's best friend!

Hints in DATA Step and PROC SQL

```
set ora_lib.ora_table (orhints='/*+hints(s)*/');

Select * from ora_lib.ora_table (orhints='/*+hints(s)*/');
```

Hints in SQL Pass_Through SQL

In SQL Pass-Through use the ordinary Oracle syntax. The option PRESERVE_COMMENTS are required. Otherwise SAS will interpret the hint as an ordinary comment that will not be passed on to Oracle, and hence not executed.

```
proc sql;
 connect to oracle (user=ora_sas password=ora_pwd path=ora_serv preserve_comments);
    Select * from connection to Oracle (
           Select /*+hint(s)*/ * from ora_table);
quit;
```

Special considerations when using hints in SQL Pass-Through in macros

When used in macros hints are treated as comments and will not be stored in the compiled macro. Wrap the hints with %str in order to preserve and pass them on to Oracle:

```
proc sql;
 connect to oracle (user=ora_sas password=ora_pwd path=ora_serv preserve_comments);
    Select * from connection to Oracle (
           Select %str(/)%str(*)+hint(s)%str(*)%str(/) * from ora_table);
quit;
```

The effect of Oracle hints

It is not always easy to get a complete picture of the effect of the hints. Third-party tools like the free Oracle SQL Developer will give you the opportunity to view the Explain plan and the cost of the query being implemented. If you do not have direct access to Oracle, you have to rely on the processing timing information from the SAS log. Adding **options fullstimer** or **sastrace=',,,s'** will open up an ocean of performance statistics. Always start with a subset of data using the SAS inobs/obs options to avoid running inefficient queries on large amounts of data. It is not an easy task to cancel a running Oracle query from SAS. Even after stopping the process in SAS, the query will continue to run on Oracle. Again, your DBA might not turn out very happy. Believe me, I have been there!

```
proc sql;
connect to oracle (user=user
password= pwd path=serv);
  create table usage as
  select * from connection to oracle
   (select
    * from sas_ora.usage a
    where number like '47%' and
    length(number )=10
    and period = '20120901');
quit;
```

```
NOTE: Table WORK.usage created, with
63473033 rows and 6 columns.
NOTE: PROCEDURE SQL used (Total process
time):
            real time          22:18.09
            cpu time            2:43.25
```

```
proc sql;
connect to oracle (user=user
password=pwd path=serv
preserve_comments);
  create table usage as
  select * from connection to oracle
   (select /*+full(a) parallel(a, 8)*/
    * from sas_ora.usage a
    where number like '47%'
    and length(number )=10
    and period = '20120901')
quit;
```

```
NOTE: Table WORK.usage created, with
63473033 rows and 6 columns.
NOTE: PROCEDURE SQL used (Total process
time):
     real time             6:22.49
     cpu time            2:48.07
```

Adding **buffsize=10000** we were able to further reduce the time spent**:**

```
     real time             4:10.09
```

Note! In the example just one day of data is fetched. Imagine what amount of time is saved when extracting data for the entire month!

CONCLUSION

Optimizing is an art. You will never be fully trained. But even for the beginner there are quite a few buttons to push to make the data flow faster. Your NEED FOR SPEED is substantially more tangible.

Amongst others, digging into this topic has revealed some facts:

- SAS offers plenty of optimization opportunities
- They are quite easy to implement
- Might give better performance with little effort
- Provide more efficient use of resources
- Buffer consumes memory
- Parallels consumes processor
- It will require some effort to find the optimal settings in the specific environment and circumstances.
- To be wiser, use available options to output useful information to the log
- And last, but not least; Always test the options on small subsets of data

DISCLAIMER: The contents of this paper are the work of the author. All opinions and recommendations in this document are his alone.

Comparative testing has been carried out in a shared environment and thus the results presented are not to be considered accurate.

REFERENCES

Levin, Louid. “Methods of Storing SAS® Data into Oracle Tables.” *Proceedings of SUGI 29,* May 2004. http://www2.sas.com/proceedings/sugi29/106-29.pdf

Liming, Douglas B. “Five Ways to Speed Up Your Data Loading Using SAS/ACCESS® for Relational Databases”. *Proceedings of the SAS® Global Forum 2010 Conference.* April 2011. http://support.sas.com/resources/papers/proceedings11/103-2011.pdf

Passing Functions to the DBMS Using PROC SQL
http://support.sas.com/documentation/cdl/en/acreldb/63647/HTML/default/viewer.htm#a001630468.htm

RECOMMENDED READING

"SAS/ACCESS® 9.2 for Relational Databases: Reference, Fourth Edition"
http://support.sas.com/documentation/cdl/en/acreldb/63647/HTML/default/viewer.htm#titlepage.htm

CONTACT INFORMATION

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

Name:	Svein Erik Vrålstad
Enterprise:	Knowit Decision Oslo, Norway
Address:	Lille Grensen 5
ZIP City:	0159 Oslo
Work Phone:	+47 480 70 54
E-mail:	sveinerik.vralstad@knowit.se
Linkedin:	http://www.linkedin.com/in/sveine