OPTIMIZING SAS® ACCESS TO AN ORACLE DATABASE IN A LARGE DATA WAREHOUSE

Clare Somerville, Team Comtex, Wellington, New Zealand
Clive Cooper, Department of Social Welfare, Wellington, New Zealand

ABSTRACT

SAS/ACCESS® to Oracle® provides facilities to treat an Oracle database as if it's a series of SAS data sets. Experience in a data warehouse environment has shown that there are a number of issues that need to be addressed to maximize some of the options available for performance improvement to update and query access.

One of the major components of the New Zealand Department of Social Welfare's data warehouse is the replication of an 80 gigabyte database from Unisys™ DMS II into Oracle on a UNIX platform.

Analysts use SAS on this warehouse for query access. But it has also been used to create the Oracle database, and to maintain the database replica in step with the Unisys system.

Because of the extensive quantities of data that are being dealt with here, it has been necessary to undertake some testing in order to optimize the SAS access to Oracle. This paper presents some of the findings from this work.

The paper describes different methods of accessing Oracle with SAS, the use of syntax to control what gets passed through to Oracle, and the results of running the same tests against SAS and Oracle.

BACKGROUND

Environment

The data warehouse is based on a UNIX platform: an HP9000/K570, recently upgraded from a K460, with six 64 bit PA8200 processors. There are 633 gigabytes of available disk space provided by twelve disk arrays operating in RAID5, and two gigabytes of RAM. The platform operating system is HP-UX version 10.20.

The database software is Oracle, version 7.3.2.3. SAS version 6.12 is used on the warehouse platform and on the warehouse users desktop. The desktop is currently running Windows NT 4.0.

The Oracle Database Replica

The Oracle database is a replica of the operational system SWIFTT. The SWIFTT system is a DMS II database running on a Quad Unisys A series. It currently consists of 216 different tables, stored in around 80 gigabytes.

To create the Oracle database, an extract of all the SWIFTT data was taken. Since the extract, daily logs of all database changes have been accumulated in flat files consisting of around 500 megabytes per day.

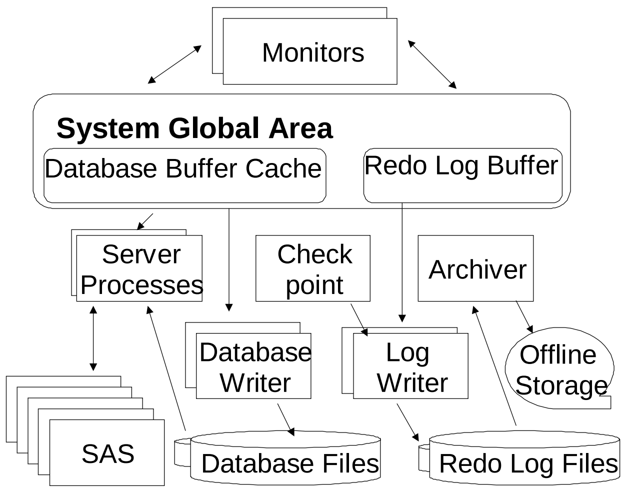
The initial extract data was read into Oracle. The daily logs, which are sent to the data warehouse electronically, are used to update the Oracle database with all the database changes. Though the warehouse database is a replica, it differs from the SWIFTT database in that a transaction history is maintained. In SWIFTT, data is overwritten; in the warehouse replica all database changes are maintained as logical deletes, with the changed data inserted. This provides the time dimension required by analysts.

PERFORMANCE ISSUES

Our SUGI 22 paper included a number of the broader issues of performance of the SAS/Oracle interface. The three aspects of performance to be considered here include: the SAS system domain, the Oracle domain, and the gateway -- SAS/ACCESS Interface to Oracle -- which joins them.

Monitoring performance on both sides of the gateway is a complex process. First there is the issue of measuring CPU consumption itself. The SAS system CPU consumption can be isolated to each SAS job by setting the fullstimer option.

It's not so simple for the Oracle environment. The diagram below shows the main Oracle programs that are running to support the database. Each of the Oracle programs contributes something to the process:



If more than one SAS program is accessing Oracle, the question of separating out the resource consumption becomes impractical. Taking the global totals of consumption can provide an approximation of the consumption by SAS and Oracle over a given period. On our system this provides a split of the CPU time of 58.5% for SAS and 41.5% to Oracle.

When using the SAS/ACCESS gateway the CPU is consumed in part by SAS and Oracle checking the status of the gateway. The UNIX sockets mechanism used involves the exchange of messages between SAS and Oracle. For example, SAS will ask Oracle for data and Oracle retrieves the data and sends it to SAS. The number of times this exchange takes place depends on a number of factors, some of which are discussed in this paper. Running on the K460, there were between 200 and 300 messages per second between SAS and Oracle for each SAS session accessing Oracle.

To manage this exchange, each side of the socket must monitor its status to see when there is something there to read. This monitoring process consumes CPU but can be considered wasted time as no productive work is being done. The testing process, and the results discussed in this paper, are aimed in part at reducing this non-productive work. The time when Oracle is working and SAS is 'idle', also contributes to the overall Realtime (elapsed time) for the job.

The second issue is about where the processing takes place in any given program. Processing can move from completely in SAS, with no Oracle involved, to almost the reverse. The following table illustrates this issue:

Usage	SAS	Oracle
SAS/ACCESS descriptor/view:		
No `WHERE`	All processing	Full table scan
With `WHERE`		
--Valid for Oracle	Process selection	Selects from table based on index availability
--Not Valid for Oracle	All processing	Full table scan
`PROC SQL` using SAS/ACCESS descriptor	All processing	Full table scan
`PROC SQL` using `Pass-Through`	Minimal output processing	All processing

Valid and invalid `WHERE` statements are discussed later in the paper.

The gateway provided by SAS/ACCESS to Oracle has a variable size. This is set by the BUFFSIZE parameter – covered in the next section. In addition, the performance through the gateway can be improved by restricting the amount of data transferred between SAS and Oracle. This is done by using `KEEP` or `DROP` in the data step `SET` statement. This has a similar effect to using the option in a standard SAS program where it reduces the amount of data brought into the program during an I/O operation. If you are using `PROC SQL` against an access descriptor or view then SAS automatically restricts the data requested of Oracle to the variables used in the `PROC`. There is more on this topic later in the paper.

BUFFSIZE

The buffer in SAS/ACCESS can be thought of as providing the ability to vary the capacity of the gateway. The default value is 25 and the maximum is 32,767. The value chosen for the access descriptor can have an effect on the performance of the gateway. Our operation involves update and insert activity in maintaining the database, and also query access by the users. A number of tests were conducted altering the BUFFSIZE, with the objective of finding the optimum value for each of these actions.

BUFFSIZE FOR UPDATE

The following table shows the time taken to update one table with six months of daily logs, with varying BUFFSIZEs:

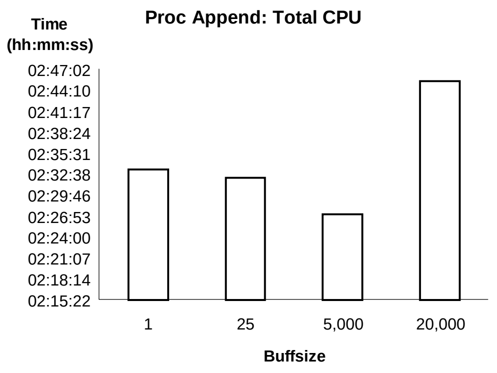
BUFFSIZE	Total CPU hh:mm:ss	Real Time hh:mm:ss
1	2:06:42	07:12:42
5	2:22:37	11:25:14
25	2:09:29	07:59:32
100	2:38:32	11:48:57
5,000	12:43:24	17:57:35

The BUFFSIZE chosen for update was one.

BUFFSIZE FOR INSERT

Tests were done on inserting (using `PROC APPEND`) a file of 3.5 million records into Oracle, using different BUFFSIZEs:

BUFFSIZE	Total CPU hh:mm:ss	Real Time hh:mm:ss
1	02:33:10	05:03:50
25	02:32:03	05:56:10
5,000	02:27:02	05:06:48
20,000	02:45:23	08:48:00

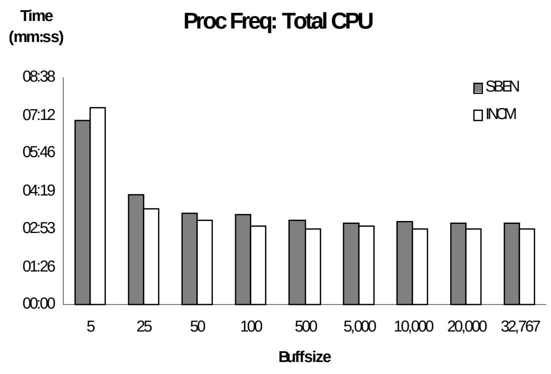


A BUFFSIZE of 5,000 produced the best CPU time.

BUFFSIZE FOR QUERY

A series of `PROC FREQs` were run varying the BUFFSIZE to check performance for query access. This was run on two files, each with around 5.8 million records: one with a record length of 103 bytes (INCM), and the other with a record length of 220 bytes (SBEN).

BUFFSIZE	INCM mm:ss	SBEN mm:ss
1	26:22	30:28
5	07:28	07:01
25	03:38	04:11
50	03:13	03:30
100	03:01	03:26
500	02:52	03:14
5,000	02:58	03:07
10,000	02:51	03:08
20,000	02:52	03:07
32,767	02:54	03:06



The results showed that as the BUFFSIZE increased, the CPU decreased, but that this levelled out around a BUFFSIZE of 5,000. The record length had little effect on the result in these cases.

ACCESS METHODS

There are two ways in which an Oracle database can be accessed using SAS. These include:

- SAS/ACCESS descriptors and views
- PROC SQL Pass-Through using PL/SQL

Access descriptors describe the Oracle database to the SAS system. By using descriptors, it is possible to access the Oracle database as if it's a SAS data set. Descriptors give us the advantage of using standard SAS code.

For example, the following data step accesses an Oracle database and creates a temporary copy of it as a SAS data set:

```
data temp ;
  set oracle.hca ;
run ;
```

In this example, `oracle` is the library name and `hca` is the access descriptor for the table of the same name.

SQL can also be used with the descriptors in `PROC SQL`. The equivalent code written in SAS ANSII SQL is:

```
proc sql ;
  create table temp as
    select * from oracle.hca ;
```

With `SQL Pass-Through` you can pass SQL statements direct to the Oracle database. This is Oracle PL/SQL:

```
proc sql ;
  connect to oracle as testone
    (user=xxxxxx password=xxxxxx) ;
  create table temp as
    select * from connection to testone
      (select * from hca) ;
  disconnect from testone ;
```

There are pluses and minuses with these methods. One of the key advantages with using descriptors is that the code tends to be easier and shorter, using all the power of the SAS language and its `PROCs`. For this reason, much of the work on this project has been done using descriptors. However, tests of SQL have also been made to identify the most efficient processing approach.

SQL CODE PASSED TO ORACLE

Some of the performance issues in accessing Oracle concern what gets passed through to Oracle, and what is processed by SAS. So when using SAS/ACCESS to Oracle there are two areas open for efficiency improvement: in SAS, and in Oracle. In the latter case it is necessary to give SAS the appropriate code to pass the best translation through to Oracle. If all else fails, Oracle PL/SQL may be used directly via the `Pass-Through` facility.

When using descriptors, a simple SAS statement gets passed through to Oracle as an SQL statement. The first example above of creating a temporary data set will cause SAS to generate the following SQL:

```
SELECT
"COMMENTS","CONDTIND","MAINTS",
. . . <naming all the variables in the table>
. . .
FROM "HCA"
```

KEEP STATEMENT

If a (KEEP=*VARNAME*) statement is added to the DATA statement, or a KEEP *VARNAME* statement is added on the data step, the select criteria remains the same: SAS does not restrict the retrieval from Oracle to the variables in the KEEP. To force the SELECT statement to include only the fields required, the KEEP must be put on the SET statement. For example:

```
data temp ;
  set oracle.hca (keep=swn) ;
run ;
```

This results in the following SQL:

```
SELECT
"SWN",
. . .
FROM "HCA"
```

A test creating a temporary SAS data set from a file of 5.7 million records took 1.5 minute CPU time with a KEEP statement for one variable on the SET. This KEEP statement was passed to Oracle.

The same test with a KEEP on the DATA statement, which was not passed to Oracle, took almost 12 minutes.

The following table summarises the placement of the KEEP statement and where it is processed:

KEEP in a	Processed By
DATA statement	SAS
SET statement	Oracle
DATA step	SAS

If the filter to restrict the amount of data going into the data set is done by SAS, then there is more data going through the gateway between Oracle and SAS.

IF/WHERE STATEMENTS

Suppose we want to include some selection criteria by adding an IF statement:

```
data temp ;
  set oracle.hca ;
  if servstus = “3” ;
run ;
```

This results in exactly the same generated SQL as in the example above with no IF statement: the IF statement is processed by SAS and not passed through to Oracle.

If we substitute a WHERE statement in place of the IF statement, the SAS code:

```
data temp ;
  set oracle.hca (where=(servstus=“3”)) ;
run ;
```

results in the following SQL, with the WHERE statement processed on the Oracle side.

```
SELECT
"SERVSTATUS","COMMENTS","CONDTIND",
...
FROM "HCA"  WHERE  ( "SERVSTUS" = '3' )
```

The following table shows when the WHERE statement is filtered by SAS or Oracle:

WHERE in a	Usually Processed By
DATA statement	SAS
SET statement	Oracle
DATA step	Oracle

Depending on the circumstances, using a WHERE statement which is passed to Oracle can result in significant CPU improvement over using an IF.

MISSING/NULL VALUES

A common requirement of our users is to select records with a MAINTS field value not equal to “D”, to exclude logically deleted records. We put the standard SAS code to select these records into a WHERE statement:

```
data temp ;
  set oracle.hca
    (where=(maints ne “D”)) ;
run ;
```

which produced the following output:

```
SELECT
"MAINTS","COMMENTS","CONDTIND",
...
FROM "HCA"
WHERE  ( "MAINTS" <> 'D' )
NOTE:  The  data  set  WORK.TEMP  has  7200
observations and 30 variables.
```

SAS has passed the WHERE clause to Oracle to process, and Oracle has returned 7,200 records, but the correct number of records which should be returned from this query is 34,045. Oracle has returned the records which have a MAINTS field value which is not “D”, but has excluded the records with a blank or missing MAINTS field. Oracle does not treat “missing” values the same as SAS.

If we modify the SAS code accordingly:

```
data temp ;
  set oracle.hca
    (where=(maints = “” or maints ne “D”)) ;
run ;
```

this still results in 7,200 records. The SQL generated by SAS shows that part of the selection statement has been ignored. SAS knows that Oracle does not understand = “” or =. so SAS does not pass it on. As Oracle does not return the needed data, SAS cannot process the missing selection itself. There is no error code returned, simply the wrong answer.

To correct this, it is necessary to use the Oracle syntax IS NULL:

```
data temp ;
  set oracle.hca
    (where=(maints is null or
            maints ne “D”)) ;
run ;
```

which results in:

```
SELECT
"MAINTS","COMMENTS","CONDTIND",
...
FROM "HCA"
WHERE  ( ( "MAINTS" IS NULL )
  OR ( "MAINTS" <> 'D' )  )

NOTE:  The  data  set  WORK.TEMP  has  34045
observations and 30 variables.
```

The WHERE statement has now been correctly passed through to and processed by Oracle, and the right number of rows obtained.

DATE SELECTION

Another user requirement is to select records based on a range of dates. The following code:

```
data temp ;
  set oracle.hca
    (where=(filedate le “02feb97”d and
    (repdate is null or
     repdate gt “02feb97”d))) ;
run ;
```

generates this SQL:

```
SELECT
"FILEDATE","REPDATE","COMMENTS","CONDTIND",
...
FROM "HCA"
```

The WHERE statement on the date fields is not passed through to Oracle, so the selection is performed by SAS. We would expect this statement to be passed through to Oracle, so we logged it with SAS support who came back with an answer: to include the time with the date:

```
data temp ;
  set oracle.hca
    (where=(filedate le “02feb97:12:00:00”dt
     and (repdate is null
     or repdate gt “02feb97:12:00:00”dt))) ;
run ;
```

This statement now gets passed through to and processed by Oracle, with substantially decreased processing time.

If an `IS NULL` statement is used in a WHERE statement on dates, it is processed correctly by SAS even though it is not passed through to Oracle. If the `IS NULL` is used in an `IF` statement, then it results in a SAS syntax error. The following table summarises the use of `IS NULL` and `=.` on different statements:

Statement	IS NULL	= .
IF	Processed by SAS: produces a SAS error	Processed by SAS
WHERE on Dates	Processed by SAS	Processed by SAS
WHERE on Other Fields	Processed by SAS	Processed by Oracle: produces the wrong answer

Data is often extracted from Oracle using the FILEDATE and REPDATE selection, but often there are other selection criteria which may or may not include other date fields.

In this example the FILEDATE/REPDATE selection is included in the WHERE statement, and the other selection criteria is in an `IF` statement:

```
data incmormd ;
  set oracle.incmmd
    (where=
    (filedate le "01nov97:12:00:00"dt and
    (repdate is null or
     repdate gt "01nov97:12:00:00"dt))) ;
  if (serv not in ('180' '181') and
    incst='1' and
    effdt le "01nov97"d) ;
run;
```

```
DEBUG:  (cur) SQL statement processed is:
SELECT "FILEDATE","REPDATE","FREQUENCY
   <naming all the variables>
FROM "INCM"
WHERE (MAINTS IS NULL OR MAINTS ^= 'D')
AND ( ( "FILEDATE" <= TO_DATE('1997-11-01-
12.00.00','yyyy-mm-dd-hh24.mi.ss') )
AND ( ( "REPDATE" IS NULL )
OR ( "REPDATE" >TO_DATE('1997-11-01-
12.00.00','yyyy-mm-dd-hh24.mi.ss') )  ) )

NOTE: The data set WORK.INCMORMD has 1365998
observations and 6 variables.
NOTE: DATA statement used:
      real time           6:44.21
      user cpu time       5:49.50
      system cpu time     33.86 seconds
      memory              1.35 M
```

This data step takes over six minutes to run. Oracle returns the required number of records from the FILEDATE/REPDATE selection, and SAS processes the `IF` statement. This code could be modified to remove the `IF` statement by including all selection criteria in the WHERE statement:

```
data incmormd ;
  set oracle.incmmd
   (where=
   ((filedate le "01nov97:12:00:00"dt and
    (repdate is null or
    repdate gt "01nov97:12:00:00"dt)) and
    serv not in ('180','181') and
    incst='1' and
    effdt le "01nov97:12:00:00"dt)) ;
run;

DEBUG:  (cur) SQL statement processed is:
SELECT "FILEDATE","REPDATE","SERVCDE",
  <naming all the variables>
FROM "INCM"
WHERE (MAINTS IS NULL OR MAINTS ^= 'D')
AND ( ( "FILEDATE" <=TO_DATE('1997-11-01-
12.00.00','yyyy-mm-dd-hh24.mi.ss') )
AND ( ( "SERVCDE" NOT IN ( '180' , '181' ) ))
AND ( "INCMSTAT" = '1' )
AND ( "EFFECTDT" <= TO_DATE('1997-11-01-
12.00.00','yyyy-mm-dd-hh24.mi.ss') )
AND ( ( "REPDATE" IS NULL )
OR ( "REPDATE" >TO_DATE('1997-11-01-
12.00.00','yyyy-mm-dd-hh24.mi.ss') )  ) )


NOTE: The data set WORK.INCMORMD has 1364495
observations and 6 variables.
NOTE: DATA statement used:
      real time           1:48.42
      user cpu time       1:22.55
      system cpu time     12.75 seconds
      memory              1.35 M
```

The processing time is reduced but the correct number of records is not obtained – around 1,500 records have been lost. Further tests showed that the inclusion of date fields in WHERE statements passed through to Oracle can produce unreliable results. This has been logged with SAS support.

INDEXES

An index on an Oracle table will not necessarily return the rows to SAS in the order of that indexed field. For example, running this code on an Oracle table with an index on SWN:

```
    data temp1 ;
      set oracle.clint (keep=swn) ;
    run ;
```

will not produce the rows in the same order as:

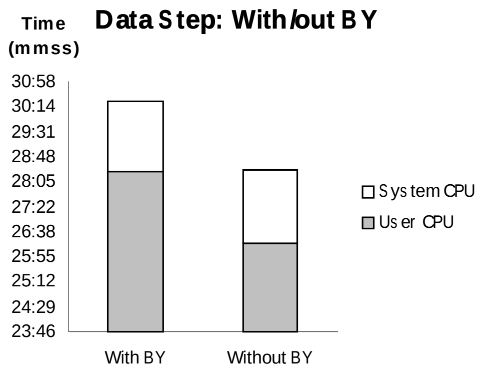
```
    data temp2 ;
      set oracle.clint (keep=swn) ;
      by swn ;
    run ;
```

At first glance, a print of the records in the data set appear to be sorted:

```
        OBS     SWN
          1    8420
          2    8471
          3    8535
          4    8607
          5    8991
```

But compared to the temp2 data set obtained with the BY statement, it is obvious they are not:

```
        OBS     SWN
          1      19
          2      35
          3      78
          4     190
          5     221
```



There is a clear overhead in terms of CPU for including the BY statement. The extra time results from the additional work done by Oracle in retrieving the records in index order.

UPDATING THE ORACLE DATABASE

The process of maintaining the Oracle database involves taking the daily log files containing all update changes, and applying them to the Oracle database.

The logs contain sequences of records which are identified as “add” or “delete” records. A “delete” record may be a stand alone “delete”, or it may be followed by an “add” -- this is equal to an “update”. In this case the “delete” record gives an image of the record as it was in the database, and the following “add” record contains the updated record.

An “add” record may be a new record to be added, or it may be an “update”. In addition to these valid “add” and “delete” records, the logs contain multiple “no-change” records which have to be screened out and removed.

Having identified the valid records to be included, a process was developed to update the database. To do this, it was necessary to match the “delete” records to the existing record in the database. The existing record would have a replacement date set to show that it was no longer the current record. The “add” records were then inserted into the database as new records, with a file date (FILEDATE field) showing the date the record was added, and a blank replacement date (REPDATE).

The initial method designed to update the Oracle database with the daily logs was to use the SAS MODIFY statement. Using the MODIFY statement, the _IORC_ variable can be checked to ascertain if a match of records has been found between the log file and the Oracle table. If a match is found, the “deleted” record can be replaced (a logical delete), and the “add” record inserted in the Oracle table.

The following skeleton code for the update MODIFY does a REPLACE for the matched “delete” records, and an OUTPUT for the new “add “records.

```
    data oracle.table ;
      modify oracle.table
             logfile ;
      by ids repdate ;
      select (_iorc_) ;
    * Match located ;
        when %str((%sysrc(_sok))) do ;
          replace oracle.table ;
        end ;
    * Match not located ;
        when %str((%sysrc(_dsenmr))) do ;
          output oracle.table ;
        end ;
    * Error on anything else ;
        otherwise do ;
          put "** ERROR Unexpected error
            condition: _IORC_ = " _iorc_ ;
          abort return ;
        end ;
      end ;
    run ;
```

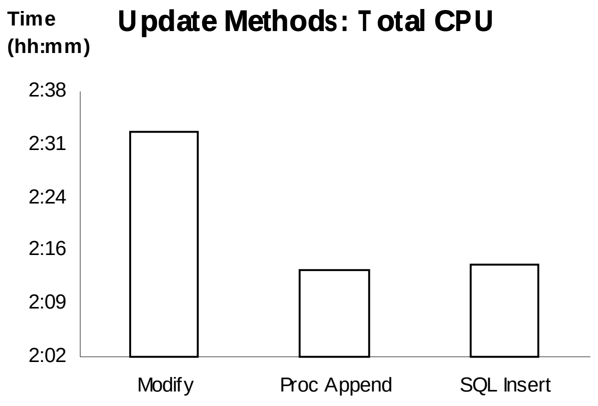
Extensive updating of the database was done using this method. But with the accumulation of a backlog of daily logs, and some very large tables, applying the updates covering a three month period to one table could take up to twelve days of continuous processing. Obviously, it was still necessary to speed up this process.

Altering the BUFFSIZE did not produce the required performance improvement. At SUGI 22 in San Diego we had discussions with SAS personnel responsible for writing the MODIFY statement. These discussions suggested that the slowest part of this UPDATE MODIFY process would be in the insertion of the “add” records, rather than in the replacing of the “delete” records. To test this, the “add” records were output to a temporary SAS data set and appended to the Oracle table in a separate PROC APPEND step, using a BUFFSIZE of 5,000:

```
data oracle.table
  adds ;
  modify oracle.table
         logfile ;
  by ids repdate ;
  select (_iorc_) ;
* Match located ;
    when %str((%sysrc(_sok))) do ;
      replace oracle.table ;
    end ;
* Match not located ;
    when %str((%sysrc(_dsenmr))) do ;
      output adds ;
    end ;
* Error on anything else ;
    otherwise do ;
      put "** ERROR Unexpected error
        condition: _IORC_ = " _iorc_ ;
      abort return ;
    end ;
  end ;
run ;

proc append  base = oracle.table
             data = adds ;
run ;
```

In addition to PROC APPEND, PROC SQL INSERT was tested for adding the new records:



Our tests showed that it was possible to improve the update process CPU time by about 13% using either PROC APPEND or SQL INSERT. There seemed to be little difference between using PROC APPEND and PROC SQL INSERT.

Another point in favour of the PROC APPEND method was what appeared to be a bug in the MODIFY process, whereby an “add” record which should be inserted would not be output into the Oracle table. This was an infrequent and random event, which we could not reliably duplicate.

PROC MEANS

Users query the Oracle database using SAS/ACCESS. A number of tests were carried out to try to optimize the query access to Oracle.

Many of these tests were conducted on the APPRG table. This table contains 5.7 million records, with 23 fields (record length of 215), and is 820 MB in size. The DBDEBUG option was set on.

PROC MEANS output was obtained from APPRG in a variety of different ways. The MEANS included a WHERE statement based on the FILEDATE and REPDATE fields. The following queries were run:

1. PROC MEANS on the Oracle database using descriptors, without an index, selecting on date
2. PROC MEANS on the Oracle database using descriptors, without an index, selecting on date/time
3. PROC MEANS on the Oracle database using descriptors, without an index, selecting on date/time, with a KEEP statement
4. PROC MEANS on the Oracle database using descriptors, with an index, selecting on date/time
5. PROC SQL on Oracle to produce the equivalent MEANS output
6. PROC SQL Pass-Through to produce a MEANS output
7. Oracle PL/SQL to produce a MEANS output

The initial PROC MEANS syntax used was:

```
proc means data = oracle.apprg
  (where=(filedate le "02feb97"d and
  (repdate is null or
   repdate gt "02feb97"d)))
   print missing ;
run ;
```

In this code there is no restriction of what gets passed back from Oracle. The addition of the date/time selection restricted the number of observations returned from Oracle, and the KEEP statement restricted it further. The MEANS procedure required far less coding than the equivalent SQL, which does have an implied KEEP:

```
proc sql ;
select
  count(actiondt), avg(actiondt),
  std(actiondt), min(actiondt),
  max(actiondt),
  count(actionti), avg(actionti),
  std(actionti),  min(actionti),
  max(actionti),
  . . . <repeated for all the variables>
  from oracle.apprg
    where (filedate <= '2feb97:12:00:00'dt)
      and
          (repdate is null or
           repdate > '2feb97:12:00:00'dt);
```

The SQL Pass-Through coding was yet more extensive, with the dates requiring more coding:

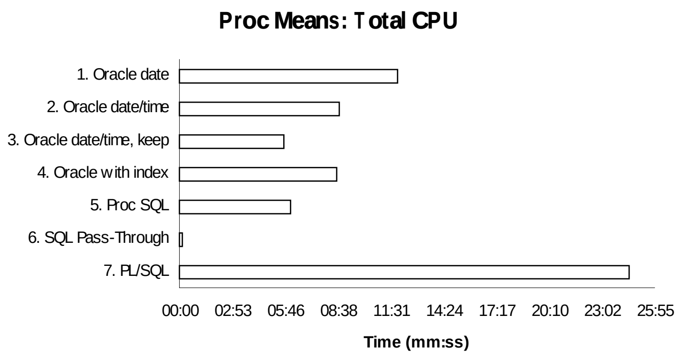
```
proc sql ;
 connect to oracle as testone (user=xxxxxx
password=xxxxxx) ;
 select * from connection to testone
(select
   count(actiondt),
   avg(to_number(to_char(actiondt,'j'))),
   stddev(to_number(to_char(actiondt,'j'))),
   min(actiondt), max(actiondt),
   count(actionti), avg(actionti),
   stddev(actionti),  min(actionti),
   max(actionti),
   . . . <repeated for all the variables>
   from oracle.apprg
   where (filedate <= round
     (to_date('2-feb-1997','dd-mon-yyyy'))
   and (repdate is null or
     repdate > round
    (to_date('2-feb-1997','dd-mon-yyyy')))));
   disconnect from testone ;
```

Not only is the SQL code more extensive, but the output from the PL/SQL code in particular is not neatly formatted as in the PROC MEANS output. This would require more work. And of course there are some statistics that PROC MEANS could produce for which there is no equivalent in SQL.

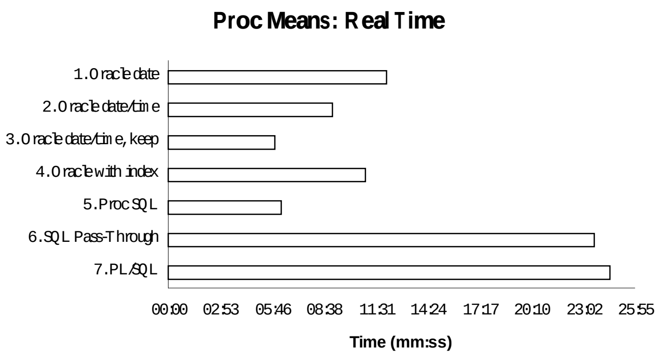
The timings produced for these tests were:

Query Type		Total CPU mm:ss	Real Time mm:ss
1.	Oracle without index, date	11:54	12:08
2.	Oracle without index, date/time	08:43	09:08
3.	Oracle without index, date/time, KEEP	05:42	05:55
4.	Oracle with index, date/time	08:36	10:59
5.	PROC SQL	06:03	06:18
6.	SQL Pass-Through	00.09	23:41
7.	PL/SQL	24:33	24:33

It was expected that by adding an index on the FILEDATE/REPDATE fields in Oracle that the processing times would decrease. This was not so – the processing time remained about the same.



The SQL Pass-Through looks like a good option until we look at the elapsed time and see that the SAS CPU time does not give an accurate indication of the processing time involved. All the processing is done in Oracle:



The PROC SQL took less CPU than the MEANS statement on Oracle because less data was passed back to SAS: only the fields specified in the SELECT statement would be passed through the gate separating Oracle and SAS. When a KEEP statement is added to the PROC MEANS, this time difference disappears. A VAR statement also limits the number of variables selected and therefore improves the CPU time.

PROC FREQ, MEANS AND SUMMARY

Tests were done performing three queries: a PROC FREQ, PROC MEANS and PROC SUMMARY. These simple queries used a WHERE statement on the two date fields. The following queries were tested:

1. Running the queries on Oracle using access descriptors
2. Creating a temporary SAS data set from Oracle and running the queries on the temporary data set
3. Creating a view from Oracle using descriptors, and running the queries on the view
4. As for 3. but using PROC SQL to create the view.

The results were as follows:

Query Method	Total CPU mm:ss	Real Time mm:ss
1. Oracle	12:12	18:16
2. SAS temp data set	13:40	14:11
3. SAS view	27:43	28:43
4. SQL SAS view	27:18	28:20

There is little difference between the two different methods of creating the view. Using the view adds substantially to both the CPU and the elapsed time when the view is used more than once, as SAS creates a temporary data set each time the view is used.

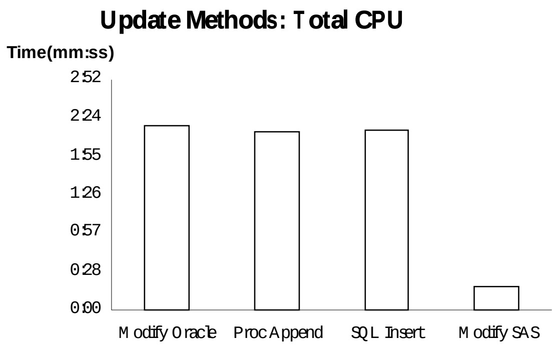
The time taken using the temporary SAS data set could be reduced by the addition of a KEEP statement, as could the Oracle processing.

Depending on the processing required, and the size of the data set, creating a temporary SAS data set for a series of queries may be quicker than running each query on the Oracle database.

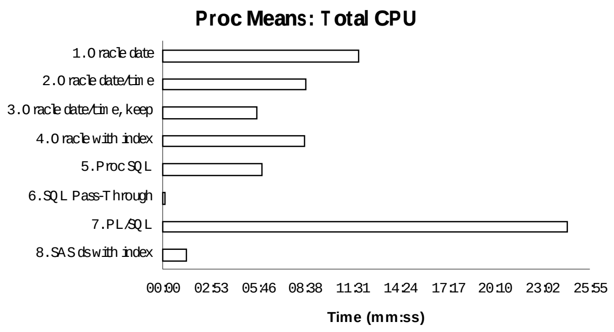
SAS DATA SET COMPARISON

During the period we were testing the SAS/ACCESS interface, we made a number of comparisons against the same table structures in SAS data sets. It is appropriate to cover these results here to show how the SAS system performs when SAS/ACCESS is not involved.

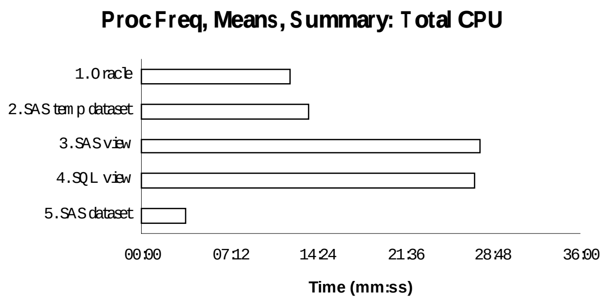
The update `MODIFY` was run over SAS data sets. All methods tested against Oracle took over two hours of CPU. Running against SAS the `MODIFY` took eighteen minutes.



The `PROC MEANS` on an indexed SAS data set produced a total CPU of one minute twenty-seven seconds. This time is comparable to the `SQL Pass-Through` CPU for the same `MEANS`, but it does not have the overhead of elapsed time – just over a minute compared to almost 24 minutes for the `Pass-Through`.



The series of simple queries was run over a SAS data set. The CPU time of three minutes forty seconds compared to the best time against Oracle of 12 minutes.



CONCLUSION

There are no simple solutions to improve the performance of the SAS/ACCESS to Oracle interface.

The tests described in this paper show a number of ways in which performance may be effected. The efficiencies achieved depend on the size of the gateway and what is passed through it.

- Test your environment for the best size of the BUFFSIZE parameter. Use different access descriptors for significantly different needs.
- Perform the data extract filtering on the most appropriate side of the gateway.
- Ensure the syntax used is translatable by SAS into the syntax used by the database being accessed, for example in the `WHERE` clause.
- Use `KEEP` or `DROP` on the `SET` statement to restrict the amount of data brought into the data step.
- Create temporary SAS data sets when the same subset of data is required for use in more than one `PROC` or `DATA` step.

Clearly, if the option is available to you, the best performance comes from using SAS data sets. That approach is significantly faster than accessing the same data through the SAS/ACCESS Interface to Oracle. If you are using the SAS/ACCESS interface, the tests show that you can not afford to program inefficiently: you must consider all possible efficiency opportunities.

Finally, carry out appropriate tests to ensure you are achieving the best performance in your specific environment.

References

SAS Institute Inc., *SAS/ACCESS® Interface to Oracle®: Usage and reference, Version 6, Second Edition*, Cary, NC: SAS Institute Inc., 1993. 261pp

SAS Institute Inc., *SAS/ACCESS® Software for Relational Databases:, Reference, Version 6, First Edition*, Cary, NC: SAS Institute Inc., 1994.

Oracle 7 Server Reference, Release 7.3, January 1996, Part No. A32589-1

Oracle 7 ServerSQL Reference, Release 7.3, February 1996, Part No. A32538-1

Paper 112, SUGI 22 Proceedings of the Twenty-Second Annual *SAS® Users Group International Conference,* San Diego, California March 16-19 1996

Paper 78, SUGI 21 Proceedings of the Twenty-First Annual *SAS® Users Group International Conference,* Chicago, Illinois March 10-13 1996

Author Contact:

Clare Somerville
Consultant
6 Raroa Crescent
Wellington 5
New Zealand
Phone 64 4 25 501 575
Fax: 64 4 916 3916
email clares@actrix.gen.nz
email clare.somerville@dsw.govt.nz