Techniques for Reading DB2 Data
Steven Feder, Federal Reserve Board, Washington, DC

ABSTRACT

Of the ways to get results from DB2® using SAS®, which work best in which situations? We compare PROC SQL, with and without explicit pass-through, data step code, and procedures. We test merging subsetted observations from multiple tables and simple summarizations. We also look at comparisons on the Windows® and z/OS® platforms.

INTRODUCTION

More than a few papers have been written about reading DB2 data, and SAS Institute Inc. provides ample documentation online. Please see the Reference section for lists of papers and links. SAS provides three basic techniques for reading data: PROC SQL, with and without explicit pass-through, and the data step. In addition, SAS PROCs also can read data directly from the database. In this paper we compare the essential techniques, reviewing some of the many tips on which to use, and explore further the differences in efficiency.

GENERALLY ASSUMED GUIDELINES FOR EFFICIENCY

A very short and limited list of guidelines for efficiency would be:

- Only retrieve the columns needed.
- Write code that has the database perform merges/joins and summarizations and functions.
- Retrieve data from the server only after summarization.
- Retrieve the minimum subset of data needed.
- Code functions that the database can evaluate.

For review of the details of these and many related guidelines, see the publications selected in the reference section. This paper will focus mainly on comparisons of the basic techniques, keeping this short list in mind.

PLATFORM

Benchmark testing of code was done on two platforms. Initial testing was done on the SAS for Windows platform using version 9.2. The SAS/Access to ODBC® engine was used to read the data on this platform. It was expected that comparison testing reading the same DB2 server on the SAS for z/OS platform using our production version 9.13 would produce results similar, at least in the direction of the various comparisons. This was in fact not always the case. We also tested with the SAS for z/OS version in development at our site, 9.2, with results similar to the 9.13 tests.

METHOD

For this paper various coding techniques that produce the same data results were tested in comparison to each other. To reduce the effects of external factors such as database server and network load, the coding techniques were run serially in a macro loop 11 times. (See Appendix 1 for an example of this macro loop in a full program.) This means that each set of runs of a variety of techniques was subject to similar conditions. Averaged over the entire series of runs, one anticipates that this should effectively eliminate the effect of variations in server performance. The log of the code is then read with a macro using simple INPUT statements. To identify which loop and section of code the log is recording, Macro %PUT statements write flag lines to the log. (See Appendix 2 for the log reading macro.) The first run is discarded to exclude whatever initialization time might be involved. A PROC UNIVARIATE takes the mean of the ten results for each type of code. Running the loop ten times reasonably ensures that variations in the load or performance of the server will not distort the results. To examine what SQL code the DB2 database is executing, this option is set:

    OPTIONS sastrace=',,d' sastraceloc=saslog nостsuffix;
BASIC CODE ON WINDOWS

The basic code reads an id variable and a date from DB2, plus seven numeric variables to simulate low but significant "real-world" production demands. All of the code tested in this paper is simplified but based on real situations. The code reads the data three ways, using implicit PROC SQL, referred to subsequently as PROC SQL, PROC SQL with Explicit Pass-through, referred to subsequently as PASS-THROUGH, and a Data Step:

PROC SQL

```sql
proc sql;
create table call2 as
select r5.id_rssd, r5.dt, r5.d_dt,r5.RCON7204,r5.RCON7205,r5.RCON7206,
    r6.RCON3385,r6.RCON3386,
    r7.RCON3771,r7.RCON5363
from infdr.cuv_rcri05_nc as r5,
    infdr.cuv_rcri06_nc as r6,
    infdr.cuv_rcri07_nc as r7
where r5.id_rssd=r6.id_rssd and r5.id_rssd=r7.id_rssd and
    r5.dt=r6.dt and r5.dt=r7.dt and
    r5.dt>=19990630 and r5.dt<=20090630
order by id_rssd,dt;
quit;
```

PROC SQL WITH EXPLICIT PASS-THROUGH

```sql
proc sql;
connect to odbc(dsn=M1DB2P uid=m1shf00 pwd=&pwd);
create table call1 as
select * from connection to odbc
    (select r5.id_rssd, r5.dt, r5.d_dt,r5.RCON7204,r5.RCON7205,r5.RCON7206,
    r6.RCON3385,r6.RCON3386,
    r7.RCON3771,r7.RCON5363
from fdrp.cuv_rcri05_nc as r5,
    fdrp.cuv_rcri06_nc as r6,
    fdrp.cuv_rcri07_nc as r7
where r5.id_rssd=r6.id_rssd and r5.id_rssd=r7.id_rssd and
    r5.dt=r6.dt and r5.dt=r7.dt and
    r5.dt>=19990630 and r5.dt<=20090630)
order by id_rssd,dt;
disconnect from odbc;
quit;
```

DATA STEP

```sql
data call3;
    merge infdr.cuv_rcri05_nc (keep=id_rssd dt d_dt RCON7204 RCON7205 RCON7206)
        infdr.cuv_rcri06_nc (keep=id_rssd dt d_dt RCON3385 RCON3386)
        infdr.cuv_rcri07_nc (keep=id_rssd dt d_dt RCON3771 RCON5363)
    ;
    by id_rssd dt;
    where dt>=19990630 and dt<=20090630;
run;
```
BASIC WINDOWS RESULTS

At this point the casual reader interested in knowing the best approaches to reading data may want to skip to the conclusions section. The results turn out at times to be confusing and counter-intuitive, and overall, more complicated than expected.

All times presented are in seconds and are rounded to the whole second. The first run of the code was done without the subsetting for date in the code above and produced these results averaged over the ten runs in the loop:

<table>
<thead>
<tr>
<th>Code</th>
<th>Real Time</th>
<th>CPU Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>PROC SQL</td>
<td>51</td>
<td>1</td>
</tr>
<tr>
<td>PASS-THROUGH</td>
<td>38</td>
<td>1</td>
</tr>
<tr>
<td>DATA STEP</td>
<td>51</td>
<td>1</td>
</tr>
</tbody>
</table>

The conclusions based on this first set of runs seem simple enough. PROC SQL and Data Step perform the same within rounding error, and PASS-THROUGH is obviously faster.

However, we went on to testing the code exactly as it is printed above, with the subsetting, and obtained these results on two full runs of the loops:

**Run 1**

<table>
<thead>
<tr>
<th>Code</th>
<th>Real Time</th>
<th>CPU Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>PASS-THROUGH</td>
<td>40</td>
<td>1</td>
</tr>
<tr>
<td>DATA STEP</td>
<td>60</td>
<td>1</td>
</tr>
<tr>
<td>PROC SQL</td>
<td>53</td>
<td>1</td>
</tr>
</tbody>
</table>

**Run 2**

<table>
<thead>
<tr>
<th>Code</th>
<th>Real Time</th>
<th>CPU Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>PASS-THROUGH</td>
<td>33</td>
<td>1</td>
</tr>
<tr>
<td>DATA STEP</td>
<td>38</td>
<td>1</td>
</tr>
<tr>
<td>PROC SQL</td>
<td>37</td>
<td>1</td>
</tr>
</tbody>
</table>

The results with subsetting are substantially different from the previous run without subsetting. The direction of the results for the two runs with subsetting is the same, but the magnitude of the differences is substantially different. The performance ratios differ as follows:

**Run 1**

<table>
<thead>
<tr>
<th>Groups</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATA STEP/PASS-THROUGH</td>
<td>1.5</td>
</tr>
<tr>
<td>PROC SQL/PASS-THROUGH</td>
<td>1.3</td>
</tr>
<tr>
<td>DATA STEP/PROC SQL</td>
<td>1.1</td>
</tr>
</tbody>
</table>
Run 2

<table>
<thead>
<tr>
<th>Groups</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATA STEP/PASS-THROUGH</td>
<td>1.2</td>
</tr>
<tr>
<td>PROC SQL/PASS-THROUGH</td>
<td>1.1</td>
</tr>
<tr>
<td>DATA STEP/PROC SQL</td>
<td>1.0</td>
</tr>
</tbody>
</table>

We present the data for these multiple runs with disparate results to demonstrate that simple, obvious results may mislead. How many runs is enough? Would statistical testing of the differences be needed to draw reliable conclusions? Given the limits of writing this paper and the real-world situations it simulates, we decided against pursuing statistical testing and ended the testing series arbitrarily. We concluded that, for this type of simple merge example, when using SAS/ACCESS to ODBC on SAS for Windows, PASS-THROUGH runs faster. The limitations of this conclusion will be important in the next section when the same testing is done on z/OS.

First, however, we want to explain the results, particularly the apparent superiority of PASS-THROUGH to PROC SQL, given that the SAS code seems to be doing the exact same thing.

The SASTrace command noted above produced the following statements in the Windows SAS log. The statements the server ran are bolded:

**PASS-THROUGH:**

ODBC_1: PREPARED: ON CONNECTION 1
ODBC: COMMIT PERFORMED ON CONNECTION 1.

ODBC_2: EXECUTED: ON CONNECTION 1
PREPARED STATEMENT ODBC_1
NOTE: TABLE WORK.CALL1 CREATED, WITH 340372 ROWS AND 10 COLUMNS.

**SQL:**

ODBC: AUTOCOMMIT TURNED ON FOR CONNECTION ID 0
ODBC_3: PREPARED: ON CONNECTION 0
SELECT * FROM FDRP.CUV_RCRI05_NC FOR READ ONLY
ODBC: COMMIT PERFORMED ON CONNECTION 0.
ODBC_4: PREPARED: ON CONNECTION 0
SELECT * FROM FDRP.CUV_RCRI06_NC FOR READ ONLY
ODBC: COMMIT PERFORMED ON CONNECTION 0.
ODBC_5: PREPARED: ON CONNECTION 0
SELECT * FROM FDRP.CUV_RCRI07_NC FOR READ ONLY
ODBC: COMMIT PERFORMED ON CONNECTION 0.
ODBC_6: PREPARED: ON CONNECTION 0
ODBC: COMMIT PERFORMED ON CONNECTION 0.
ODBC_7: EXECUTED: ON CONNECTION 0
PREPARED STATEMENT ODBC_6
ACCESS ENGINE: SQL STATEMENT WAS PASSED TO THE DBMS FOR FETCHING DATA.
Data Step:

- ODBC_8: PREPARED: ON CONNECTION 0
  SELECT * FROM FDRP.CUV_RCRIO5_NC FOR READ ONLY
  ODBC: COMMIT PERFORMED ON CONNECTION 0.

- ODBC_9: PREPARED: ON CONNECTION 0
  SELECT * FROM FDRP.CUV_RCRIO6_NC FOR READ ONLY
  ODBC: COMMIT PERFORMED ON CONNECTION 0.

- ODBC_10: PREPARED: ON CONNECTION 0
  SELECT * FROM FDRP.CUV_RCRIO7_NC FOR READ ONLY
  ODBC: COMMIT PERFORMED ON CONNECTION 0.

- ODBC_11: PREPARED: ON CONNECTION 0
  SELECT "DT", "ID_RSSD", "D_DT", "RCON7204", "RCON7205", "RCON7206" FROM FDRP.CUV_RCRIO5_NC WHERE ( "DT" BETWEEN 19990630 AND 20090630 ) ORDER BY "ID_RSSD", "DT" FOR READ ONLY
  ODBC: COMMIT PERFORMED ON CONNECTION 0.

- ODBC_12: EXECUTED: ON CONNECTION 0
  PREPARED STATEMENT ODBC_11

- ODBC_13: PREPARED: ON CONNECTION 0
  SELECT "DT", "ID_RSSD", "D_DT", "RCON3385", "RCON3386" FROM FDRP.CUV_RCRIO6_NC WHERE ( "DT" BETWEEN 19990630 AND 20090630 ) ORDER BY "ID_RSSD", "DT" FOR READ ONLY
  ODBC: COMMIT PERFORMED ON CONNECTION 0.

- ODBC_14: EXECUTED: ON CONNECTION 0
  PREPARED STATEMENT ODBC_13

- ODBC_15: PREPARED: ON CONNECTION 0
  SELECT "DT", "ID_RSSD", "D_DT", "RCON3771", "RCON5363" FROM FDRP.CUV_RCRIO7_NC WHERE ( "DT" BETWEEN 19990630 AND 20090630 ) ORDER BY "ID_RSSD", "DT" FOR READ ONLY
  ODBC: COMMIT PERFORMED ON CONNECTION 0.

- ODBC_16: EXECUTED: ON CONNECTION 0
  PREPARED STATEMENT ODBC_15

  NOTE: THERE WERE 340372 OBSERVATIONS READ FROM THE DATA SET INFDR.CUV_RCRIO5_NC.
  WHERE (DT>=19990630 AND DT<=20090630);
  NOTE: THERE WERE 340372 OBSERVATIONS READ FROM THE DATA SET INFDR.CUV_RCRIO6_NC.
  WHERE (DT>=19990630 AND DT<=20090630);
  NOTE: THERE WERE 340372 OBSERVATIONS READ FROM THE DATA SET INFDR.CUV_RCRIO7_NC.
  WHERE (DT>=19990630 AND DT<=20090630);

The PASS-THROUGH code by definition runs exactly what we send: one SELECT joining the tables. The PROC SQL step, however, runs three SELECT statements, appearing first to read the data, and then runs the one expected SELECT statement. The Data Step runs three SELECT statements, appearing first also to read the data, and then runs three SELECT statements appearing to subset the data.

We note that, according to these log statements, the SQL and Data Step codes generate a BETWEEN statement from the user-coded “>=” and “<=" operators which the PASS-THROUGH code executes as is. This seems like it could be important, but a test shows that in this case it does not make a noticeable difference. In this case, the BETWEEN operator is not a cause for concern.

To explore further what causes the difference in performance we switched the SASTRACE option to "s":

```sql
OPTIONS sastrace=',,,' sastraceloc=saslog nostsuffix;
```

This option produced a log with the times produced by the SQL statements the server runs:
PASS-THROUGH:

Summary Statistics for ODBC are:
Total row fetch seconds were: 28.501159
Total SQL execution seconds were: 0.085435
Total SQL prepare seconds were: 0.000108
Total SQL describe seconds were: 0.005895
Total seconds used by the ODBC ACCESS engine were 29.590420

NOTE: PROCEDURE SQL used (Total process time):
real time 29.60 seconds
cpu time 0.95 seconds

PROC SQL:

Summary Statistics for ODBC are:
Total SQL prepare seconds were: 0.000062
Total SQL describe seconds were: 0.110370
Total seconds used by the ODBC ACCESS engine were 0.358855

Summary Statistics for ODBC are:
Total SQL prepare seconds were: 0.000046
Total SQL describe seconds were: 0.088049
Total seconds used by the ODBC ACCESS engine were 0.246200

Summary Statistics for ODBC are:
Total SQL prepare seconds were: 0.000028
Total SQL describe seconds were: 0.073808
Total seconds used by the ODBC ACCESS engine were 0.155023

NOTE: Table WORK.CALL2 created, with 340372 rows and 10 columns.

Summary Statistics for ODBC are:
Total row fetch seconds were: 6.173920
Total SQL execution seconds were: 38.256525
Total SQL prepare seconds were: 0.000141
Total SQL describe seconds were: 0.075130
Total seconds used by the ODBC ACCESS engine were 44.727542

NOTE: PROCEDURE SQL used (Total process time):
real time 44.99 seconds
cpu time 0.39 seconds

Data Step:

NOTE: There were 340372 observations read from the data set INFDR.CUV_CRI05_NC.
WHERE (dt>=19990630 and dt<=20090630);

Summary Statistics for ODBC are:
Total row fetch seconds were: 6.356960
Total SQL execution seconds were: 6.180503
Total SQL prepare seconds were: 0.000146
Total SQL describe seconds were: 0.033642
Total seconds used by the ODBC ACCESS engine were 38.425079
NOTE: There were 340372 observations read from the data set INFDR.CUV_RCRI06_NC. WHERE (dt>=19990630 and dt<=20090630);

Summary Statistics for ODBC are:
Total row fetch seconds were: 5.347939
Total SQL execution seconds were: 8.024472
Total SQL prepare seconds were: 0.000080
Total SQL describe seconds were: 0.048544
Total seconds used by the ODBC ACCESS engine were 38.418223

NOTE: There were 340372 observations read from the data set INFDR.CUV_RCRI07_NC. WHERE (dt>=19990630 and dt<=20090630);

Summary Statistics for ODBC are:
Total row fetch seconds were: 5.755618
Total SQL execution seconds were: 6.107545
Total SQL prepare seconds were: 0.000080
Total SQL describe seconds were: 0.055179
Total seconds used by the ODBC ACCESS engine were 38.412954

NOTE: The data set WORK.CALL3 has 340372 observations and 10 variables.
NOTE: DATA statement used (Total process time):
    real time 38.43 seconds
    cpu time 1.18 seconds

For the PASS-THROUGH code, the server spends very little time in “execution,” and the join is completed during the “fetch” time. For the SQL code, the server spends most of the time in “execution” and relatively little in “fetch.” It is not exactly clear why this is the case, but it seems that the SQL code retrieves rows before merging, whereas the PASS-THROUGH merges as it fetches. To fully explain the statements in the log, a conversation or two with a DB2 administrator or SAS developer would be required. It seems safe to say, however, that the PASS-THROUGH code runs faster than the PROC SQL code because the server is doing something significantly different. The Data Step spends a similar amount of time both fetching and executing. Is the server fetching rows, then subsetting?

To add another option for how to produce the join that has been tested, the SQL code can also be written this way:

```sql
proc sql;
create table call2 as
select r5.id_rssd, r5.dt, r5.d_dt,
    r5.RCON7204, r5.RCON7205, r5.RCON7206,
    r6.RCON3385, r6.RCON3386,
    r7.RCON3771, r7.RCON5363
from infdr.cuv_rcri05_nc as r5 join
    infdr.cuv_rcri06_nc as r6 on
        (r5.id_rssd=r6.id_rssd and r5.dt=r6.dt)
    infdr.cuv_rcri07_nc as r7 on
        (r5.id_rssd=r7.id_rssd and r5.dt=r7.dt)
where r5.id_rssd=r6.id_rssd and r5.id_rssd=r7.id_rssd and
    r5.dt=r6.dt and r5.dt=r7.dt and
    r5.dt>=19990630 and r5.dt<=20090630
order by id_rssd,dt;
quit;
```

Comparisons of the SQL statement logs of this code to the previous logs show exactly the same server statements generated, and the comparative times are indistinguishable. So, for this simple example, we can choose a technique based solely on which style of SQL we prefer to write.
We also ran comparison tests of the code with and without the SASTRACE option set, just to make sure that extra logging was not producing any sort of artifact. The logging did not appear to make any difference.

We note that it is possible also to set both the SASTRACE options at once, though the volume of the log may be prohibitive for typical use:

```
options sastrace=',, ds' sastraceloc=saslog nostsuffix;
```

**BASIC z/OS RESULTS**

We ran the same series of benchmark tests of the code (with subsetting for date) already shown for SAS for Windows on z/OS twice, using our site’s current production version, 9.13, and the development version, 9.2.

The results of three runs (of ten iterations of the series of code techniques) show results different in direction from the results on Windows:

**Version 9.13**

<table>
<thead>
<tr>
<th>Code</th>
<th>Elapsed Time</th>
<th>CPU Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>PASS-THROUGH</td>
<td>93</td>
<td>72</td>
</tr>
<tr>
<td>DATA</td>
<td>136</td>
<td>124</td>
</tr>
<tr>
<td>SQL</td>
<td>63</td>
<td>57</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Ratios</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>DATA STEP/PROC SQL</td>
<td>2.2</td>
<td>2.2</td>
</tr>
<tr>
<td>PASS-THROUGH/PROC SQL</td>
<td>1.5</td>
<td>1.3</td>
</tr>
</tbody>
</table>

**Version 9.2**

<table>
<thead>
<tr>
<th>Code</th>
<th>Elapsed Time</th>
<th>CPU Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>PASS-THROUGH</td>
<td>135</td>
<td>97</td>
</tr>
<tr>
<td>DATA</td>
<td>171</td>
<td>138</td>
</tr>
<tr>
<td>SQL</td>
<td>94</td>
<td>74</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Ratios</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>DATA STEP/PROC SQL</td>
<td>1.8</td>
<td>1.9</td>
</tr>
<tr>
<td>PASS-THROUGH/PROC SQL</td>
<td>1.4</td>
<td>1.3</td>
</tr>
</tbody>
</table>

On z/OS, PROC SQL is noticeably more efficient than PASS-THROUGH, and Data Step is much less efficient than PASS-THROUGH. Other things being equal, PROC SQL is the clear choice on z/OS. The log with SASTRACE shows the same pattern of server statements executed as on Windows. We will not speculate about why PROC SQL is the most efficient. More research would help, both in benchmarking and in discussion with experts.
SUMMARIZATION TECHNIQUES

Next we look at techniques for summarizing data, using standard SQL functions, data step code, and PROC UNIVARIATE:

PROC SQL

proc sql;
create table call3 as
select r5.id_rssd, avg(r5.RCON7205) as avg7205,
    sum(r5.RCON7205) as sum7205,
    count(r5.RCON7205) as cnt7205
from infdr.cuv_rcri05 as r5
group by id_rssd;
quit;

PROC SQL WITH EXPLICIT PASS-THROUGH

proc sql;
connect to odbc(dsn=M1DB2P uid=m1shf00 pwd=&pwd);
create table call3 as
(select r5.id_rssd, avg(r5.RCON7205) as avg7205,
    sum(r5.RCON7205) as sum7205,
    count(r5.RCON7205) as cnt7205
from fdrp.cuv_rcri05 as r5
group by id_rssd);
disconnect from odbc;
quit;

DATA STEP

data call4;
set infdr.cuv_rcri05(keep=id_rssd dt d_dt RCON7205);
by id_rssd dt;
retain avg7205 sum7205 cnt7205 0;
if first.id_rssd then do;
    sum7205=0;
    cnt7205=0;
end;
sum7205=sum7205+rcon7205;
cnt7205=cnt7205+1;
if last.id_rssd then do;
    avg7205=sum7205/cnt7205;
    output;
end;
drop rcon7205;
run;

PROC UNIVARIATE

proc univariate data=infdr.cuv_rcri05 noprint;
by id_rssd;
var rcon7205;
output out=call15 n=cnt7205 mean=avg7205 sum=sum7205;
run;

We also tested PROC MEANS, but the SQL generated is the same as for the PROC UNIVARIATE, and the results are similar.

WINDOWS SUMMARIZATION RESULTS

On Windows, based on ten runs of the serial comparison as before, PASS-THROUGH and SQL perform similarly on the summarization code. The PROC UNIVARIATE code performs less well, and the Data Step is far worse:

<table>
<thead>
<tr>
<th>Code</th>
<th>Real Time</th>
<th>CPU Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>PASS-THROUGH</td>
<td>44</td>
<td>0</td>
</tr>
<tr>
<td>DATA STEP</td>
<td>85</td>
<td>3</td>
</tr>
<tr>
<td>PROC SQL</td>
<td>45</td>
<td>0</td>
</tr>
<tr>
<td>UNIVARIATE</td>
<td>57</td>
<td>2</td>
</tr>
</tbody>
</table>

The SASTRACE portion of the log makes the difference clear:

PASS-THROUGH:
ODBC: AUTOCOMMIT is YES for connection 1
ODBC_1: Prepared: on connection 1
select r5.id_rssd, avg(r5.RCON7205) as avg7205,sum(r5.RCON7205) as sum7205, count(r5.RCON7205) as cnt7205 from fdrp.cuv_rcri05 as r5 where r5.dt>=20010101 and r5.dt<=20090630 group by id_rssd
ODBC: COMMIT performed on connection 1.

PROC SQL:
ODBC: AUTOCOMMIT turned ON for connection id 0
ODBC_3: Prepared: on connection 0
SELECT * FROM fdrp.CUV_RCRI05 FOR READ ONLY
ODBC: COMMIT performed on connection 0.
ODBC_4: Prepared: on connection 0
 select r5."ID_RSSD", AVG(r5."RCON7205") as avg7205, SUM(r5."RCON7205") as sum7205, COUNT(r5."RCON7205") as cnt7205 from fdrp.CUV_RCRI05 r5 where r5."DT" between 20010101 and 20090630 group by r5."ID_RSSD" FOR READ ONLY
ODBC: COMMIT performed on connection 0.
ODBC_5: Executed: on connection 0
Prepared statement ODBC_4

Data Step:
ODBC_6: Prepared: on connection 0
SELECT * FROM fdrp.CUV_RCRI05 FOR READ ONLY
ODBC: COMMIT performed on connection 0.
ODBC_7: Prepared: on connection 0
SELECT "DT", "ID_RSSD", "D_DT", "RCON7205", FROM fdrp.CUV_RCRI05 WHERE ( ( "DT" BETWEEN 20010101 AND 20090630 ) ) ORDER BY "ID_RSSD","DT" FOR READ ONLY
For both SQL techniques the database server summarizes the data. The Data Step and PROC UNIVARIATE both read all of the data, and SAS then summarizes.

**Z/OS SUMMARIZATION RESULTS**

On z/OS, to limit resources consumed for this paper, we ran the benchmark test code only once per version. The SQL statements generated are similar to those on Windows. One run is not enough to say with any certainty that there is any difference in the PASS-THROUGH and SQL performance. The Data Step and UNIVARIATE both perform much less well:

**Version 9.13**

<table>
<thead>
<tr>
<th>Code</th>
<th>Elapsed Time</th>
<th>CPU Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>PASS-THROUGH</td>
<td>65</td>
<td>51</td>
</tr>
<tr>
<td>DATA STEP</td>
<td>228</td>
<td>197</td>
</tr>
<tr>
<td>PROC SQL</td>
<td>57</td>
<td>51</td>
</tr>
<tr>
<td>UNIVARIATE</td>
<td>210</td>
<td>175</td>
</tr>
</tbody>
</table>

**Version 9.2**

<table>
<thead>
<tr>
<th>Code</th>
<th>Elapsed Time</th>
<th>CPU Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>PASS-THROUGH</td>
<td>64</td>
<td>53</td>
</tr>
<tr>
<td>DATA STEP</td>
<td>252</td>
<td>198</td>
</tr>
<tr>
<td>PROC SQL</td>
<td>67</td>
<td>53</td>
</tr>
<tr>
<td>UNIVARIATE</td>
<td>231</td>
<td>177</td>
</tr>
</tbody>
</table>

On both platforms either type of SQL is the clear choice for efficiency.
IN-DATABASE PROCESSING

SAS/access to DB2 can now use in-database processing for certain procedures, including PROC MEANS, but only on certain platforms. Those platforms do not include access through ODBC or on z/OS. This is why PROC MEANS reads all the data from the server before doing the summarization, just as in PROC UNIVARIATE.

DB2 MONITORING

We did not have the capability to monitor the server with the various tools DB2 provides. These tools would be useful for research, but most SAS users of DB2 probably do not have these tools either.

CONCLUSIONS

On our simple merge/join benchmark test on SAS for Windows, PROC SQL with explicit pass-through performed marginally better than implicit PROC SQL. Both performed better than Data Step code. On z/OS, implicit PROC SQL performed better than PROC SQL with explicit pass-through, with both again better than Data Step code. The disparity between platforms in the direction of results for the simple merge was unexpected and requires further explanation, beyond using SASTRACE. On our simple summarization test, both types of SQL performed better than Data Step, PROC UNIVARIATE, and PROC MEANS code, on both platforms. This was as expected. Variation in the test results, despite measures to reduce the effect of external factors, suggests a measure of circumspection in interpreting the results.

REFERENCES:


ACKNOWLEDGMENTS

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

Other brand and product names are registered trademarks or trademarks of their respective companies.

CONTACT INFORMATION

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

Steven Feder
Federal Reserve Board of Governors
20th & Constitution Ave, NW
Washington, DC 20551
202-452-3144
Steven.h.feder@frb.gov
Appendix 1 – Benchmark Test Macro Loop

OPTIONS sastrace=',,,d' sastraceloc=saslog nostream;
%let prog=MERGE;

proc printto log="I:\dev\nesug\2011\&prog._LOG.log" new;
run;

libname infdr odbc complete="dsn=M1DB2p;Uid=m1shf00;pwd=&pwd"
   schema=fdrp;

%macro all;
   %do i=1 %to 11;
   %put LOOP &i~;
   %put PROG MERGE~;
   %put CODE CONNECT~;
   /* PROG: SDB1b.sas */
   proc sql;
   connect to odbc(dsn=M1DB2p uid=m1shf00 pwd=&pwd);
   create table call1 as
   select * from connection to odbc
   (select r5.id_rssd, r5.dt, r5.d_dt,r5.RCON7204,r5.RCON7205,r5.RCON7206,
      r6.RCON3385,r6.RCON3386,
      r7.RCON3771,r7.RCON5363
      from fdrp.cuv_rcri05_nc as r5,
      fdrp.cuv_rcri06_nc as r6,
      fdrp.cuv_rcri07_nc as r7
      where r5.id_rssd=r6.id_rssd and r5.id_rssd=r7.id_rssd and
      r5.dt=r6.dt and r5.dt=r7.dt)
    order by id_rssd,dt;
   disconnect from odbc;
   quit;
   %put PROG MERGE~;
   %put CODE SQL~;
   /* PROG: SDB1a2c.2.sas */
   proc sql;
   create table call2 as
   select r5.id_rssd, r5.dt, r5.d_dt,r5.RCON7204,r5.RCON7205,r5.RCON7206,
      r6.RCON3385,r6.RCON3386,
      r7.RCON3771,r7.RCON5363
      from infdr.cuv_rcri05_nc as r5,
      infdr.cuv_rcri06_nc as r6,
      infdr.cuv_rcri07_nc as r7
      where r5.id_rssd=r6.id_rssd and r5.id_rssd=r7.id_rssd and
      r5.dt=r6.dt and r5.dt=r7.dt)
    order by id_rssd,dt;
   disconnect from odbc;
   quit;

%macro all;
%put PROG MERGE~;
%put CODE DATA~;

/* PROG: SDB1a2b.1.sas */
data call3;
    merge infdr.cuv_rcri05_nc (keep=id_rssd dt d_dt RCON7204 RCON7205 RCON7206)
        infdr.cuv_rcri06_nc (keep=id_rssd dt d_dt RCON3385 RCON3386)
        infdr.cuv_rcri07_nc (keep=id_rssd dt d_dt RCON3771 RCON5363)
    ;
    by id_rssd dt;
run;

%end;
%mend;
%all

proc printto log=log;
run;

%inc 'LogRead.sas';
%logread(prog=&prog)
APPENDIX 2 – LOG READING MACRO

%macro logread(prog=);

data t;
  infile "I:\dev\nesug\2011\&prog._LOG.log" missover pad;
  length prog $40;
  retain loop prog code realtime cputime;
  input c1 $40.;
  pl=index(c1,'LOOP');
  if pl>0 then do;
    loop=scan(substr(c1,pl+5),1,'~');
  end;
  p=index(c1,'PROG');
  if p>0 then do;
    prog=scan(substr(c1,p+5),1,'~');
  end;
  pc=index(c1,'CODE');
  if pc>0 then do;
    code=scan(substr(c1,pc+5),1,'~');
  end;
  if index(c1,'The SAS System used') then prog=' ';
  IF prog ne ' ' then do;
    prt=index(c1,'real time');
    if prt>0 then do;
      RealTimec=scan(substr(c1,prt+10),1,'s');
      realtime=input(trim(left(RealTimec)),stimer7.);
    end;
    pct=index(c1,'cpu time');
    if pct>0 then do;
      CPUTimec=scan(substr(c1,pct+10),1,'s');
      cputime=input(trim(left(CPUTimec)),stimer.);
      output;
    end;
  end;
run;

proc sort;
  by code;
run;

proc univariate data=t noprint;
  class code;
  var realtime cputime;
  where loop>'1';
  output out=t2 mean=realtime cputime;
run;
ods tagsets.excelxp file="I:\dev\nesug\2011\&prog..xls";
proc print;
run;
ods tagsets.excelxp close;
%mend logread;