What a View!
Matthew Cohen, Wharton Research Data Services

ABSTRACT
This paper will present practical information about when to use SAS® views. Should you create a view or a temporary dataset when it will be read multiple times? Is it OK to join two views or to join two datasets into a view? Does it matter if PROC SQL or the DATA step is used? The answers might surprise you. I will focus on real-world results on multiple platforms rather than theoretical answers.

INTRODUCTION
A view in SAS, as in most databases, is a set of instructions. These instructions read from one or more underlying tables, join them together, subset them, or perform any number of calculations. The only things stored on disk are variable descriptors such as data types and lengths, along with the code that created the view. In recent versions of SAS, this code is retrievable using a DESCRIBE statement in a DATA step or in the SQL Procedure.

This paper will cover only native views (DATA step views, SQL views). The alternative, interface views are created with SAS/Access and work with 3rd party data storage products. It will also focus on the first two of the many benefits of views:

**Save disk space by avoiding intermediate datasets**
**Reduce run time**
Separate data storage from data presentation; shield users from architecture changes
Distribute prepared information to non-programmers, hiding complex joins
Ensure that analyses and reports are run on the most recent data
Security – restrict access to a subset of a table

DATA STEP VS SQL VIEWS
SAS supports two different kinds of native views, DATA step and SQL views. The main feature of the DATA step view is that it allows all the usual DATA step programming, including the ability to read raw data directly to a view. Only the DATA statement itself has to change:

```sas
data new;
  set mylib.base;
```

Becomes:

```sas
data new /view=new;
  set mylib.base;
```

A couple other features to note: libnames in views are fixed. If the 'base' dataset is moved to a different library or libref 'mylib' is redefined, the view will not update automatically. Also, global statements such as FILENAME and LIBNAME included in a view definition are ignored. These generally don't belong inside a DATA step anyway.

SQL views can't read raw data. However, they can update the underlying data, which DATA step views cannot. SQL views can also be subset with a where clause before being loaded into memory, while DATA step view must be entirely loaded into memory and then subset.

TESTING
To test the performance of Data views and SQL views compared to datasets, I make 4 variations of a simple task: create a new variable from two existing variables, then run a report on the new variable with the MEANS procedure. The variations are made by joining a second, third, and fourth table before running the report.

The main data table, SECD, is 6GB and has daily security information, from which I calculate the market cap of each company using the closing price and shares outstanding. The first variation joins to FUNDA, a 5GB dataset, to get the historical SIC code. The SIC code is added to the CLASS statement in PROC MEANS to analyze market caps by
industry. The second variation joins to FUNDA and then COMPANY, a 100MB dataset to get location of the company headquarters. Location becomes the second CLASS variable in PROC MEANS. The final variation joins a fourth table, FUNDA_FNCD, a 10GB dataset to determine if there are footnotes to the financial statements. Each table adds one variable to the CLASS statement in PROC MEANS to help analyze the data. Normally, I would only keep the necessary variables from each new table, but the point of this test is to stress the system a little.

This is the code used in the first variation.

```sas
data table1;
  set v.secd_6g;
  mktcap = prccd * cshoc;
run;

proc means data=table1;
  var mktcap;
run;
```

A subsequent set of tests add a second report to each variation, run with the REPORT procedure. Running a second report in addition to PROC MEANS, forces the views to be generated a second time. In other words, the original data must be read twice. This particular where clause keeps only one company out of more than 8,000, screening out over 99% of the 6 GB dataset.

```sas
proc report data=table1;
  where gvkey="006066";
run;
```

Although these are very simplistic from a research point of view, the point is to test reading from a single table or joining 2, 3, or 4 tables and then running a report. The four variations are run again, replacing the datasets with Data views, then again with SQL views. The same tests are run on a very powerful Linux 64-bit server (4.3 TB of free disk space, 50+ GB memory) and a mid-range Windows 7 32-bit desktop (230 GB of free disk space, 4 GB memory, and a dual core 3 GHz processor).

It is very difficult to accurately time the programs because of other tasks that might be competing for resources and because of caching done by SAS, the operating system, or other components. To minimize these discrepancies, I ran each program three times and in different orders, keeping the median time. I also restarted my desktop between each run. All programs are submitted from the command line to avoid SAS storing the datasets in memory. On Unix this is accomplished with the command

```
sas table1.sas
```

And on Windows with

```
"C:\Program Files\SAS\SASFoundation\9.2\sas.exe" -sysin table1.sas
```

**EXPECTED PROBLEMS**

Each program consists of just a few lines of code, and was very easy to write. The only problems occurred with SQL views. Each table has a company identifier, Gvkey, in common that is used to do the join. PROC REPORT also uses Gvkey in its where clause. When running the program, I received the error “The pushed-down WHERE clause made an ambiguous reference to one or more of the view's variables.” It appears that a SQL view looks at the entire view definition and execution together, including all the joins used to create it and the reports in which it runs. Since it analyzed PROC REPORT before performing the join, it found a variable with the same name in two different tables.

Neither the Data views nor the tables encountered this problem because they ran one step at a time. By the time those programs reached PROC REPORT the join had been completed and only a single instance of Gvkey remained.

To resolve this problem, I attempted to use SQL syntax in PROC REPORT to refer to one of tables (where a.gvkey="006066"). This did not work because SQL syntax is not valid in other procedures. Next, I attempted
to drop Gvkey from one of the tables in PROC SQL (`create view table1 (drop=a.gvkey)`). This did not work because “SAS dataset options are not permitted for SQL views”. Finally, I renamed gvkey in one of the tables before the merge. This was far from an ideal solution, but it worked. It could also have been avoided by listing each variable in the select statement rather than writing select * . However, the tables in this experiment have several thousand variables, so it was not feasible.

**RESULTS**

<table>
<thead>
<tr>
<th>Linux</th>
<th>Tables</th>
<th>Reporting Procs</th>
<th>Tables Realtime</th>
<th>Data Views Realtime</th>
<th>SQL Views Realtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Means</td>
<td>4:39</td>
<td>1:34</td>
<td>1:46</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Means + Report</td>
<td>5:30</td>
<td>2:13</td>
<td>1:39</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Means</td>
<td>3:47</td>
<td>2:17</td>
<td>0:45</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Means + Report</td>
<td>4:31</td>
<td>3:27</td>
<td>1:02</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Means</td>
<td>20:56</td>
<td>2:36</td>
<td>1:52</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Means + Report</td>
<td>28:34</td>
<td>5:04</td>
<td>1:36</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Means</td>
<td>59:34</td>
<td>8:45</td>
<td>6:19</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Means + Report</td>
<td>88:32</td>
<td>28:33</td>
<td>6:34</td>
<td></td>
</tr>
</tbody>
</table>

Starting with a single table, Data and SQL views produced similar results when running a single report, and both view types were much faster than using tables. This was expected because of the IO saved by not writing to and then reading from an intermediate table.

As more tables were joined and the processing load increased, the SQL views performed increasingly better than the Data views, although the Data views still significantly outperformed the tables. In addition to real time, the CPU time of the SQL views was between ½ and 1/3rd that of the Data views and 1/6th of the tables. On the other hand, memory usage went as high as 6 GB, more than 10 times the Data views, which in turn were 2 times the tables.

It is interesting to note that SQL views were hardly affected by running a second report. This may be evidence of the where clause in use. As noted above, SQL views can take advantage of the where clause and load only the needed rows into memory.

![Real Time Graph](image)
On Windows, the results started the same as on Linux with both Data and SQL views achieving similar times and both outperforming tables. But as the load increased, the Data views quickly proved to be the best option.

The SQL views were proportionally less affected than the Data views by adding the second report, however the difference in the times for two or more tables were so great that it did not matter. Closer examination showed that all three used similar amounts of memory, generally less than 200 MB (perhaps constrained by system resources or system options).
The times in the Tables column are starred for the last two rows because the programs terminated at that point due to lack of resources – they filled the entire 230 GB of available disk space. Neither the Data nor SQL views encountered this problem. This is just a small reminder that time is only one element that should be minimized. Temporary, intermediate datasets in the work directory are often overlooked because they are automatically removed at the end of the program, however they do have a real cost.
CONCLUSION

Use views! They save time, disk space, and the IO that would have been used to read and write that disk space.

These tests show a significant performance improvement with views over tables even the data is read multiple times. They did not show any problems joining two views to create a third. From a run time perspective, use SQL views when running on a server. Sacrificing memory to achieve time, disk space, and IO savings is generally a good trade in a server environment. SAS is very often constrained by IO resources. Since you cannot avoid reading the original data or writing the final output, any savings that can be achieved in the middle will not only improve your performance, but will also help everyone on the server.

When memory is in short supply, use Data views. There are other considerations, of course, and some tasks are clearly better suited to the DATA step or PROC SQL. The goal of this paper is to clarify which tool to use when both are suitable to the task at hand.

Views are just one part of writing an efficient program. They should be used in addition to dropping unneeded variables, using where clauses and indexes properly, and avoiding unnecessary sorts.

ACKNOWLEDGEMENTS

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.

CONTACT INFORMATION

Matthew Cohen
IT Director
Wharton Research Data Services
cohenmw@wharton.upenn.edu
http://wrds.wharton.upenn.edu