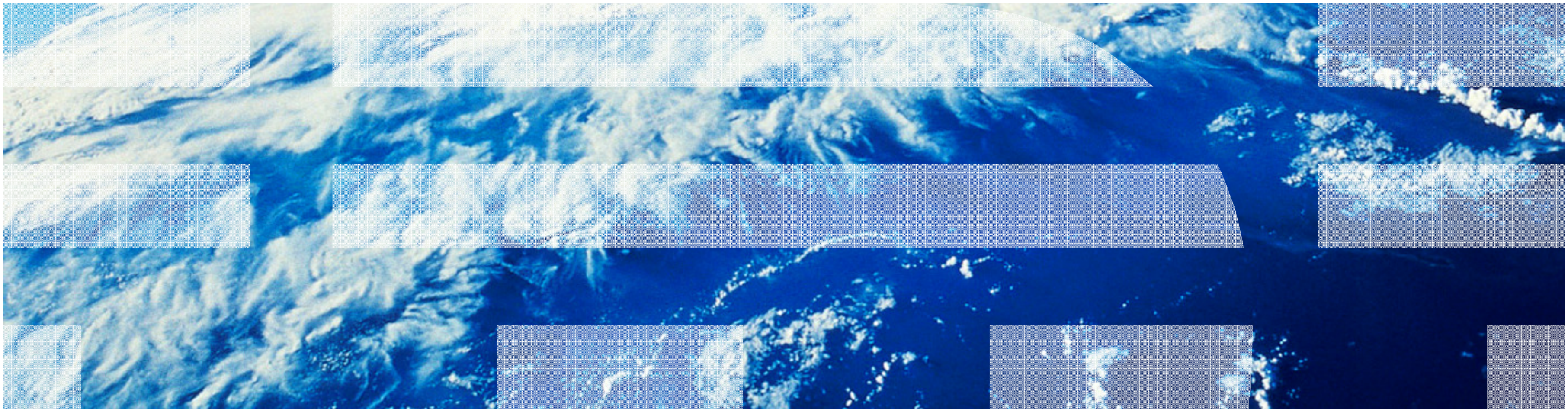
Welcome to a Smarter Planet
People want it. We can do it.

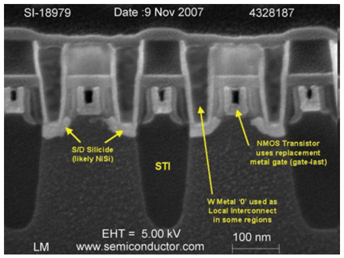
Programming Techniques for Optimizing SAS® Throughput

Steve Ruegsegger
steve.rueg@gmail.com

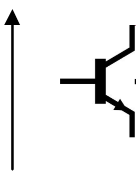


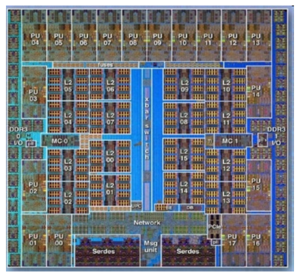
Introduction

- **Analysis environment**: Large datasets, large number of users, big servers

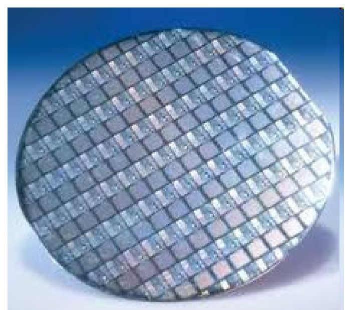


10'sM transistors on a chip/die
Measured multiple times





100's die on a wafer



25w/lot

Single transistor
~100 measurements
(Vt, Idd, Rs, Cap, Ioff,…)

100's WSD

- "Active" data warehouse ~ 10's TB
- Typical datasets ~ 1M to 10M's obs x 100's cols
- Typically 100's users @ 1000's SAS jobs / day

→ Must keep running smoothly

Optimization Strategy

- For our setup, this education has focused on a single strategy:

Reduce disk I/O

- Why?

1. It's the slowest. All other HW components – CPUs, memory, backplane, even gigabit ethernet network connection – are incredibly fast *compared to* the disk drives.
2. SAS is incredibly disk-IO intensive. It accesses every dataset from disk, *not* memory.
   – If you understand the PDV, then you know that each observation is read from the disk. So, each data and proc has very intensive disk IO.
   – Now, multiply that by large datasets and large number of users and jobs and you get a disk bottleneck.

Program Data Vector (PDV)

- A "register" or 1-D array (vector) with an element for each variable type:
    1. input vars
    2. output vars
    3. temporary (internal) vars

PDV	*Inputs*	*Outputs*	*Temps*

- 4 Steps in "The DATA Step Loop" for each obs (row)
    1. Set all vars (cols) in PDV to nulls (init)
    2. Input values from *next row* of input dataset into PDV (read)
    3. Run statements (execute)
    4. Output PDV as next row of output dataset (write)

- (+) It's how SAS handles HUGE datasets
    - "one row at a time"
    - Not a memory hog
    - Nearly infinitely scalable
    - Robust. Speed is not first priority.
    - Memory is more expensive than HD

- (-) All my SAS commands can only operate on one row
    - *(You can use lag() or retain to deal with this.)*

PDV in DATA step

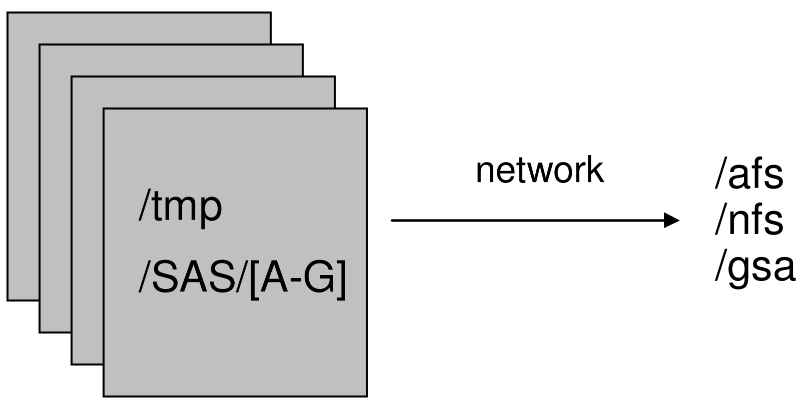
```
data out1;
     input a b c $;  * c is default length $8.;
     x=2*a; y=b+3; z=substr(c,1,3)||"A";  * step 3A;
     a=0;   b=1;   c="nikon";  * step 3B;
cards;
2 3 X96V
5 6 X92V
;
```

	input vars			calc/output vars		
	a	b	c	x	y	z
1. init:	.	.	.	.	.	.
2. read:	2	3	"X96V"	.	.	.
3A. execute:	2	3	"X96V"	4	6	"X96A"
3B.	0	1	"nikon"	4	6	"X96A"

4. write

Optimized HW setup

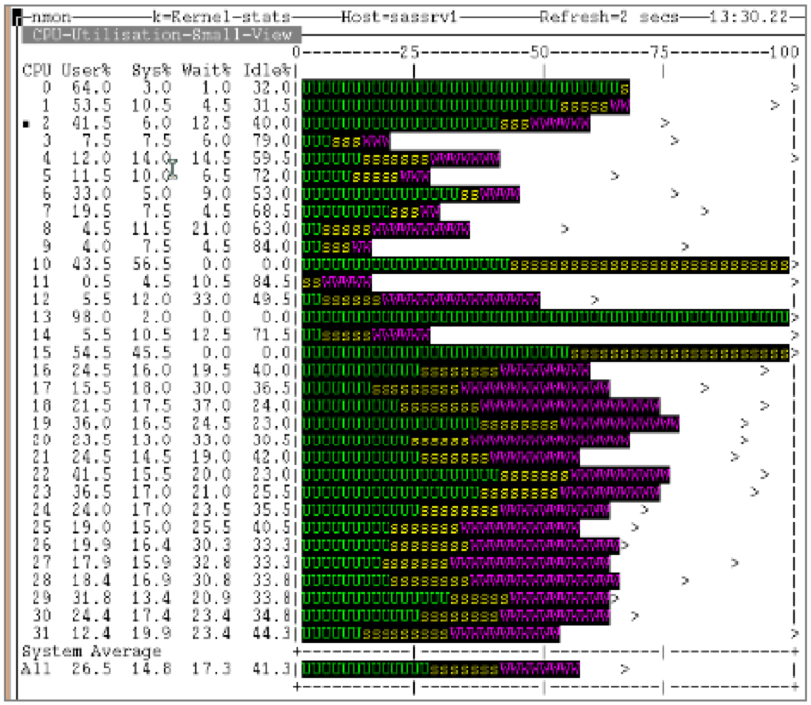
- We have a pool of 4 SAS servers
  - Each one: AIX, 32 CPUs, ~200GB RAM, > 2TB dedicated work space
- Notes on optimal HW configuration:
  - Do <u>NOT</u> use `/tmp` as SAS `work` space! *(=death)*
  - Similarly, do <u>not</u> use network space as SAS `work` space.
  - Get plenty of <u>dedicated</u> space for SAS `work` space
    - Huge
    - Fast
    - Striped – we have 5 HDs has 1 virtual drive
    - Spread users around on multiple, distributed, parallel copies of `work` space



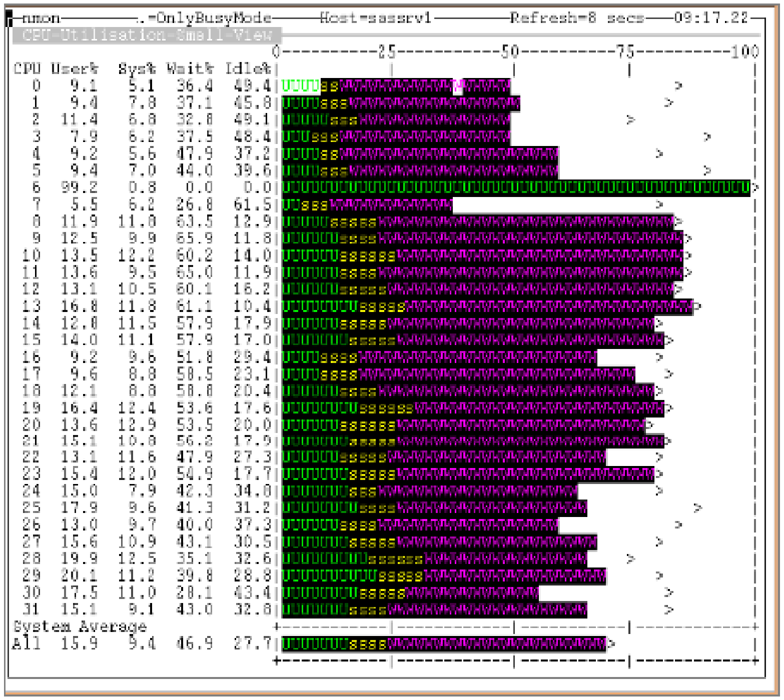
- Even with all this HW optimization ,we can still bog-down a server
  - 100's CPUs, TB RAM, maxed racks of dedicated hard drives

Problem demonstrated with increased Wait states

normal



diskIO bound



- Green U = 'system working'
- Yellow s = 'system overhead'
- Red W = *Waiting for IO*

"nmon"

Problem demonstrated in 100% utilized disks

normal

Topas Monitor for host: sassrv1 Interval: 5 Tue Mar 4

Disk	Busy%	KBPS	TPS	KB-R	ART	MRT	KB-W	AWT	MWT
hdisk16	87.4	740.[illegible]	185.0	0.0	0.0	0.0	740.0	7.3	211.8
hdisk1	82.6	1.[illegible]K	273.2	0.0	0.0	0.0	1.6K	5.3	224.5
hdisk12	67.8	10.[illegible]K	243.4	10.3K	6.4	214.4	244.0	9.0	56.8
hdisk63	67.2	8.[illegible]K	186.2	0.0	0.0	11.8	8.7K	5.8	150.8
hdisk62	65.4	8.[illegible]K	172.6	0.0	0.0	17.8	8.3K	5.9	155.4
hdisk60	63.2	8.[illegible]K	170.6	0.0	0.0	13.1	8.1K	5.6	191.6
hdisk14	62.0	9.2K	191.6	9.1K	7.1	224.4	99.2	8.7	128.8
hdisk15	61.0	9.2K	177.0	9.1K	7.0	234.1	118.4	6.9	35.7
hdisk10	60.6	9.6K	199.0	9.5K	6.7	182.5	110.4	8.0	52.7
hdisk61	60.0	7.9K	165.2	0.0	0.0	27.1	7.9K	5.5	131.4
hdisk13	59.6	10.2K	189.2	10.1K	6.5	235.0	173.6	8.2	36.7
hdisk59	58.2	8.2K	173.4	0.0	0.0	18.6	8.2K	5.2	167.4
hdisk11	56.2	9.7K	181.0	9.6K	6.5	236.3	127.2	8.7	148.8
hdisk58	44.6	503.2	125.8	0.0	0.0	0.0	503.2	3.6	120.7
hdisk26	34.0	11.8K	218.6	11.0K	2.1	24.2	824.0	4.9	29.0
hdisk25	33.6	11.7K	213.8	10.9K	2.4	133.6	816.8	5.0	72.9
hdisk29	33.4	11.9K	229.2	11.0K	2.0	218.8	888.8	4.8	28.6

diskIO bound

Topas Monitor for host: sassrv1 Interval: 2 Tue Mar

Disk	Busy%	KBPS	TPS	KB-R	ART	MRT	KB-W	AWT	MW
hdisk40	100.1	18[illegible]4K	402.4	0.0	0.0	0.0	18.4K	7.5	43.
hdisk43	100.1	25[illegible]4K	490.4	0.0	0.0	0.0	25.4K	6.1	38.
hdisk42	100.1	17[illegible]0K	378.1	0.0	0.0	0.0	17.0K	7.9	37.
hdisk41	100.1	25[illegible]2K	481.8	0.0	0.0	0.0	25.2K	6.2	29.
hdisk38	100.1	21[illegible]3K	465.1	0.0	0.0	0.0	21.3K	6.4	32.
hdisk1	88.0	1[illegible]9K	245.7	10.1	6.1	12.6	1.8K	5.5	24.
hdisk10	83.4	13.3K	267.4	13.0K	8.2	50.9	244.7	11.3	29.
hdisk12	80.4	12.9K	256.3	12.8K	7.7	85.7	12.1	8.4	29.
hdisk11	79.4	13.5K	247.7	13.2K	7.6	31.5	279.1	8.6	41.
hdisk14	78.9	13.2K	256.8	13.1K	7.5	66.1	20.2	3.8	50.
hdisk16	75.3	701.7	175.4	0.0	0.0	0.0	701.7	5.3	36.
hdisk15	73.8	13.3K	234.6	13.2K	7.4	64.8	52.6	8.0	19.
hdisk13	72.3	12.9K	227.5	12.9K	6.9	33.8	10.1	7.6	27.
hdisk56	72.3	6.9K	205.2	2.5K	6.6	40.6	4.4K	7.4	70.
hdisk46	67.2	9.1K	236.1	7.9K	5.6	39.9	1.2K	9.8	41.
hdisk45	61.7	9.0K	202.2	7.9K	6.0	28.0	1.1K	6.7	27.
hdisk39	60.2	25.8K	470.1	4.0	10.7	10.7	25.8K	3.9	37.
hdisk58	59.7	667.3	166.8	0.0	0.0	0.0	667.3	3.6	9.
hdisk47	58.1	9.1K	190.1	8.5K	6.3	25.5	675.4	7.9	33.
hdisk49	56.6	9.4K	191.1	8.2K	5.8	27.0	1.2K	7.4	25.
hdisk50	56.1	9.3K	199.7	8.2K	5.7	37.3	1.1K	6.7	20.
hdisk48	54.6	9.1K	212.3	8.4K	5.4	35.2	663.3	6.6	24.
hdisk52	53.1	5.1K	171.4	1.8K	6.0	23.8	3.4K	6.8	44.
hdisk55	52.6	4.2K	173.4	2.4K	6.5	30.4	1.7K	6.5	42.
hdisk57	50.0	4.3K	181.5	2.5K	6.2	31.4	1.7K	6.6	25.
hdisk44	49.5	572.3	143.1	0.0	0.0	0.0	572.3	3.7	26.
hdisk53	48.5	4.0K	166.8	2.2K	6.5	44.2	1.8K	6.1	32.
hdisk54	46.5	3.8K	153.2	2.1K	6.4	24.5	1.7K	5.9	49.
hdisk9	44.5	666.3	88.0	240.6	5.5	30.3	425.7	12.3	31.

5 Disks at 100% busy!

"topas"

These 5 disks are 1 striped logical drive

Demonstration of these techniques

- Obviously, past gains is not a guarantee of future performance.
- But, I ran one 'typical' SAS job with and without these techniques and averaged a 5x improvement in real-time throughput.

Date time server	Original code		Optimized code		Throughput increase
3/21 ~3pm Server1	real user sys	1h6m20.76s 0m32.24s 1m14.47s	real user sys	9m5.72s 0m8.78s 0m11.16s	6x
3/22 ~9am Server1	real user sys	54m48.77s 0m29.20s 0m59.51s	real user sys	7m30.77s 0m8.42s 0m8.79s	8x
3/22 ~10am Server3	real user sys	3m18.73s 0m55.52s 0m45.96s	real user sys	1m24.08s 0m16.89s 0m8.14s	3x

TECHNIQUE #1 - `SASFILE`

- Undocumented command, but introduced in v8.2. Documented in v9.
- Puts a dataset in memory
- Pros:
  - – Removes Disk IO
  - – Memory is fast compared to disks reads
- Cons:
  - – Cannot edit the dataset – including `PROC SORT`
  - – Must have enough free RAM.
    Doesn't make sense if computer starts memory "swapping"
- Typical use:
  - – Once a large dataset is finally prepped, and all the (editing) writing is done, and the dataset is then to be used for many reads for analysis (typically `PROC`s) – then, if enough free memory, use `sasfile`.

TECHNIQUE #1 - SASFILE

```
* editting;
data largeds;
     merge ds1 ds2;
     by id;
     run;

proc sort data=largeds;
     by descending date customer; run;

sasfile largeds open;   * put ds in memory!;
                        * no more editting;

*analysis;
proc gplot data=largeds;        * read from mem;

proc freq data=largeds;         * read from mem;

proc univariate data=largeds;   * read from mem;

sasfile largeds close;
```

No disk IO!

TECHNIQUE #2 -- INDICES

- An *index* is a separate file with keys that tells SAS how to order the main dataset, yet the main dataset is **not** re-written in that new order.
  - Example = a phonebook → address/phone-number lookup based on last name
- **By using indices, all the sorting is done "up-front"**
- Pros:
  - Allows us to use `sasfile`'s in multiple `BY` statements without the reorder of a `PROC SORT`.
  - We could eliminate many future `PROC SORT`'s (and disk IO) used for BY statements
- Cons:
  - Increased disk space
  - Doesn't save any Disk IO if only use that index once (the overhead didn't ROI in that scenario)

TECHNIQUE #2 -- INDICES

```
%put * making multiple indices *;
proc datasets library=work nolist;
  modify zscore;
    index create i1=(sasbox dttm);
    index create i2=(sasbox hour);
    index create i3=(date hour);
    index create zby;
  run;
  quit;

sasfile zscore open;   * put in memory & no more editting;

proc means; by date hour;  * i3;

proc gplot; by sasbox date; * i2;

proc freq; by sasbox dttm; * i1;
```

Without sasfile and indeces, I would have had to re-sort (read & write) and then re-read for the proc!

Now, with this new method, I already have my sort in the indeces and everything is in memory. No disk IO!

TECHNIQUE #3 -- VIEWS

- A `view` is like a "formula"
- It's like a "virtual" dataset
- SAS does <u>not</u> actually create a new dataset
  – No disk IO is used to define the `view`
- SAS simply "remembers the formula" for how to create the dataset in memory *when needed*
- Using a `view` is useful for removing the read/write of creating a new dataset.
  – Rather than using a DATA block to make a *small* change to a *large* dataset only to re-read the data for a subsequent PROC, a `view` can be used to simply record that change and then make that change when the view is used.

TECHNIQUE #3 -- VIEWS

```
proc sort data=zscore; by tech;

sasfile zscore open;   * put in mem;

/* need to add one var from merge*/
proc sql;
  create view zview as
  select a.*,  (a.var - b.mean)/b.std as z
  from zdata a
  inner join tmp b on a.tech = b.tech
  ;
  quit;

/* Now the VIEW calculates z from
   zdata (which is already in
   memory via sasdata) */

proc means data=zview noprint;
     by tech;
```

No disk IO to define this view. But I don't have to read & write a large ds – just to re-read it for the means below.

And the large DS is in memory, so guess what, No Disk IO!

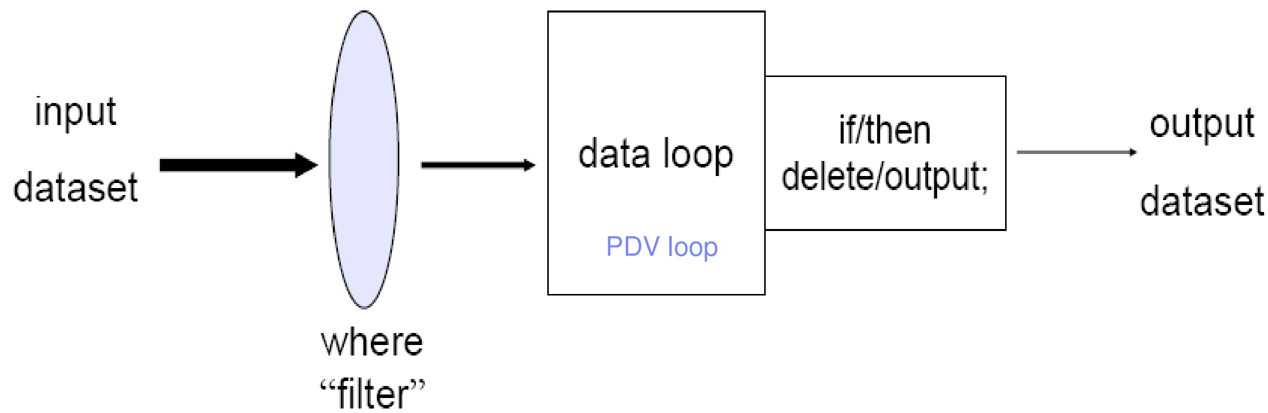
Combining these to form a strategy

- These 3 techniques were grouped together first because they form a strategy
  1. Pull, edit, manipulate your large dataset
  2. Create indices
  3. Put into memory with sasfile (must have enough free memory)
  4. Use views and dataset options as needed for changes
  5. Do all analysis
- During the analysis:
  – The 'base' dataset is already in memory from a sasfile command
  – It already appropriate indices for `by` processing
  – Any changes are from a view which is memory driven and not disk IO driven

  – The entire analysis is done from memory and only the result is written to disk. Now that's fast!

TECHNIQUE #4 -- `WHERE`

- The WHERE qualifier can be thought of as an "input filter"
  – *Only* the observations which meet the WHERE qualification are 'allowed through' during a read to populate the PDV
  – The requesting PROC or DATA command acts as if it saw a 'virtual data subset.'
- Much different than `if/then` or subsetting if
  – `where` does't even population the PDV, so large portions of the dataset might not be read
  – The subsetting-if does populate the PDV, so the entire row is read in
- Using a WHERE is usually <u>much better</u> than writing a *new* smaller dataset!



TECHNIQUE #4 -- WHERE

```
%let ndays = 30;
%let where = date ge today() - &ndays;

* edit zscore;
* index server in zscore;

sasfile open zscore;  * put in mem;

proc gplot data=zscore;
     title1 "normalized performance";
     by server;       * indexed;
     where &where;
     plot z * dttm = customer;
     run;
quit;
```

Dataset in memory from sasfile. By var indexed. Where statement skipping complete rows and not even reading into the PDV.

TECHNIQUE #5 -- RENAMING A VARIABLE

- This technique is making sure the programmers knows some particular SAS syntax.
- Some novice programmers will use a `DATA` step simply to rename a variable before a merge (in the subsequent `DATA` step)
- Method 1 is to use dataset option `rename=`
  - Rename is only for that one block
  - Format: `(rename=(old=new))`
- Method 2 is to use `proc datasets`
  - Permanently renames a variable
  - Changes the dataset headers
  - Does **not** read/write the entire dataset

```
* 2) rename perm in dataset;
proc datasets nolist;
 modify chip;
 rename lot_id = testlot
        wafer_id=wafer
        chip_id=chipid;
  run;
quit;
```

TECHNIQUE #5 -- RENAMING A VARIABLE

<table>
<tr><th>Original code</th><th>Optimized code</th></tr>
<tr><td><pre>
/* make new ds inline2 with lotid
   col to match col in kerf for
   merge */

data inline2;
     set inline;
     lotid = lot_id;
     run;

data merged;
     merge kerf inline2;
     by lotid;
     run;
</pre></td><td><pre>
data merged;
     merge kerf
           inline(rename=(lot_id=lotid));
     by lotid;
     run;
</pre></td></tr>
</table>

3 reads & 2 writes

2 reads & 1 write
(0 reads from disk if kerf & inline are
already in memory from sasfile

TECHNIQUE #6 – COMBINE BLOCKS AS MUSH AS POSSIBLE

- This technique is about thinking where redundant reads and writes can be combined.
- A classic example is a copy from library to `work` followed by a required `proc sort`.
- Rather than read/write for the copy, then read/write for the sort…
  - Read from the lib, sort, and write to work all from `proc sort` with `out=` argument.

<table>
<tr><th>Original code</th><th>Optimized code</th></tr>
<tr><td><pre>proc datasets;
     copy in=archive out=work;
     select bigds;
     run;
     quit;
proc sort data=bigds;
     by test_dt;
     run;</pre></td><td><pre>proc sort data=archive.bigds
             out=work.bigds;
     by test_dt;
     run;</pre></td></tr>
</table>

2 reads & 2 writes

1 read & 1 write

TECHNIQUE #7 – `PROC APPEND`

- This technique is about knowing the procs available
- A novice might append to large dataset using `DATA` block
- However, `proc append` will <u>not</u> rewrite the "big" ds
  - This saves a large read *and* write
- If cols don't match, an error is produced
  - Option "`force`" will override, but I try not to use that option

Original code	Optimized code

`data bigds;`
`  set bigds update;`
`  run;`
	`proc append base=bigds data=append;`
	`run;`

Reads and re-writes bigds! Only writes the smaller ds

Conclusions

- Single programming strategy for our large-server environment: reduce disk IO
  - Disks are slow hw & SAS is disk intensive
- 7 programming techniques were presented which removed redundant reads and writes
- Each technique has to be used in the correct setting
  - None are 'silver bullets' but tools to be understood.
- The first 3 techniques combine into one methodology:
  - Pull, edit, manipulate your large dataset
  - Create indices
  - Put into memory with sasfile (must have enough free memory)
  - Use views and dataset options as needed for changes
  - Do all analysis
- The paper has additional thoughts for optimization