Paper 1507-2014

SASReduce - An implementation of MapReduce in BASE/SAS®

David Moors, Whitehound Limited, UK

ABSTRACT

This paper will explain how a simple processing framework created by Google, and more recently popularised by the Open Source technology Hadoop, can be replicated using cornerstone SAS technologies such as BASE/SAS®, SAS/MACRO® and SAS/CONNECT®.

The paper will explain how, out-of-the-box, the SAS DATASTEP® can replicate the 'Map' function, and we'll discover how well established SAS Procedures can be used to create 'Reducer' like functionality. We'll also see how Parallel Processing data across multiple SAS Sessions using MP/CONNECT® can replicate MapReduce's approach to data processing.

MAPREDUCE DEFINITION

MapReduce is a programming model for processing parallelisable jobs across large dataset using a large number of nodes (computers). The concept was made popular, and later patented, by Google, and is based on the MAP and REDUCE functions found in the functional programming language LISP.

Key-value pairs form the basic structure for MapReduce tasks. The MAP procedure takes a data domain, or type, and returns a list of pairs. In the example demonstrated later in this paper the data type is a server log from NASA, containing IP addresses that have requested data from the NASA website in July 1995.

Figure 1 illustrates the basic form of a MAP task.

```
Map(k1,v1) → list(k2,v2)
```

Figure 1. MAP Input and Output

MAP tasks iterate over a dataset, extracting the defined 'key' (**k**), and a value (**v**) is assigned to the key, in some cases this is a numeric value. The output from MAP tasks are lists containing key/value pairs which may or may not be passed to a Reducer task.

The REDUCE function collects the answers (lists) from the Map() tasks and combines the results to form the output of the MapReduce task. Figure 2 below shows the basic form of a REDUCE function.

```
Reduce(k2, list (v2)) → list(v2)
```

Figure 2. REDUCE Input and Output

When looking to understand a programming model it is usually best to write some pseudo code so as to see what the code would look like without the need to interpret the syntax of a particular programming language.

Figure 3 shows a commonly used piece of pseudo code for a MAP task. This code is taken from the seminal MapReduce paper by *Dean* and *Ghemawat.*

```
map(String key, String value)
 // key: document name
 // value: document contents
  for each word w in value
    EmitIntermediate(w, "1")
```

Figure 3. Map Task pseudo code

Looking at the pseudo code for the MAP task in Figure 3, we can see that a loop (**for each**) is used to process all the data on each line of the input file. The EmitIntermediate in MapReduce outputs a word (w) and an associated value, in this case '1'.

How would we go about replicating an iterative MAP task similar to this in our favourite technology of choice, SAS? Well, luckily for us, the clever folk at SAS had the foresight to include this implied looping facility out-of-the-box in the DATA step.

The ILDS (Implied Loop of the DATA step) makes the process of writing a MAP task trivial. A full description of the

ILDS is beyond the scope of this paper; fortunately Ian Whitlock has already written a comprehensive SUGI paper converting this topic. A link to this paper can be found in the 'References' section.

The SAS DATA step code to write a simple MAP task can be seen in Figure 4 below.

```
/* MAP */
filename srvlog 'S:\sgf_paper_files\NASA_access_log_Jul95 LRECL=1500;
data work.Mapped;
    infile srvlog;
    input ip:$30;
    val=1;
run;
```

Figure 4. MAP Task using SAS DATA step

In this step a FILENAME statement is first used to associate an external file, in this case the NASA server log, to a SAS file reference. A SAS DATA step reads the external file and the INPUT statement assigns the first 30 characters of each row in the file to the variable IP. A numeric value of '1' is then assigned to the variable VAL. This process is repeated for each row in the file courtesy of the ILDS.

In a similar fashion, if we analyse the REDUCE task shown in the pseudo code in Figure 5 below, this shows a summing up of the data values (**result += Parseint(v)**) by passing the values from the MAP Task and then looping through the intermediate 'map' output.

```
reduce(String key, Iterator values):
    // key: word
    // values: a list of counts
    int result = 0;
    for each v in values:
        result += ParseInt(v) //in this case 'v' is always a 1;
    Emit(AsString(result));
```

Figure 5. REDUCE Task pseudo code

To replicate this type of processing in SAS the SQL procedure provides the out-of-the-box functionality needed to accomplish this task. Example code can be seen in Figure 6.

In this case, as the data is now available in a dataset (table), the SQL procedure will scan through all the records in the table, group the data by IP addresses, and provide a summary (count) of all the IP addresses that have been grouped.

```
/* REDUCE */
proc sql noprint;
    create table work.Reduced as
    select ip, sum(val) as count
    from work.Mapper
    group by ip
    order by count desc;
quit;
```

Figure 6. REDUCE Task using PROC SQL

And there we have it, MapReduce in SAS made simple! Everyday tasks such as importing a file and summarising the values, that most people take for granted, can be viewed as a simple MapReduce process.

Note: The above example does not provide an exact 'like for like' comparison of code, and aims to demonstrate the concepts of creating key/value pairs in SAS and looping through raw input. Appendix 1 provides a 'like-for-like' example of the SAS process shown above, using PYTHON and the MapReduce streaming API.

The MapReduce example above only describes the basic programming model. What makes MapReduce such a tantalising prospect for data processing is not only the simplistic framework, but the ability to split larger files into smaller segments and parallel process the file segments across multiple nodes or machines.

MapReduce was designed by Google to address their needs to index the entire internet in order to provide a better way to search for data. In order to do this they had to devise a software architecture to handle the vast sizes of files required to provide search terms people would use.

As Google hasn't open sourced their code for MapReduce and the GFS (Google File System) they instead published academic papers on these topics, which eventually led to the open source technology Hadoop being developed.

Hadoop, like the Google Infrastructure, incorporates a number of technologies in order to securely and optimally process large data files in parallel. This includes:

- Distributed File System.
- Batch processing of files
- Scheduling software
- Data replication, Fault tolerance, etc

SASREDUCE

SASReduce aims to provide some of the functionality provided by a MapReduce-like framework, in order parallel process raw data files on multiple SAS sessions on a SMP machine.

The components that make up SASReduce are as follows:

- **SASReduceFramework.sas**
- **Map.sas**
- **Reduce.sas**

The 'SASReduceFramework' aims to provide a level of abstraction for the developer and incorporates the following functionality.

- Parameter based Batch Processing
- File splitter
- Queue based scheduler
- Parallel processing

These will be described in more detail in later sections of this paper.

The Map.sas and Reduce.sas are user-written code called by the SASReduce framework and will also be described later on in the paper.

At the time of writing the SASReduce framework is unable to provide the following functionality found in other mainstream MapReduce frameworks:

- Data replication
- Fault tolerance
- Shared-nothing architecture

These topics may be addressed in a follow-up paper on SASReduce 2.0.

PARAMETER BASED BATCH PROCESSING:

MapReduce was designed to be a batch-oriented approach to data processing due to large file sizes the framework needs to process, and in homage to this, SASReduce has been written to allow processing to be carried out in batch. **Note**: With a few small changes the framework can process the code interactively, if desired.

To call the SASReduce job in batch from the command line (in Windows), code similar to that seen in Figure 7 can be used:

```
"C:\Program Files\SAS\SASfoundation\9.3\SAS.exe" -sysin
S:\SGF_Paper_Files\code\MRcode\SASReduceFramework.sas -sysparm
'fpath=S:\SGF_Paper_Files\inputs\, fname=NASA_access_log_Jul95,
outpath=S:\SGF_Paper_Files\output\, outfile=NASA_MR,
code_path=S:\SGF_Paper_Files\code\MRCode\, blocksize=64';
```

Figure 7. Batch submission code for SASReduce

In order to facilitate the creation of macro parameters required by the SASReduce framework program, the SYSPARM option is used to pass (via batch call) any additional text needed. The SYSPARM option is only able to pass a single string, however, by separating out the required parameters with a comma ‘,’ additional information can be gathered.

When the string from the batch call is passed to the framework a SAS DATA step reads the string from the SYPARM option, and breaks the string into multiple parameters. These values are then placed into macro variables using a CALL SYMPUT statement.

The framework code which interprets the SYSPARM string and creates multiple macro variables can be seen in Figure 8 below;

```
data _null_;
  length  sysparm express param value $ 200;
  sysparm = symget('sysparm');
  do i=1 to 50 until(express = '');
      express = left(scan(sysparm, i, ','));
      param = left(upcase(scan(express, 1, '=')));
      value = left(scan(express, 2, '='));
      valid = not verify(substr(param,1,1),
                'ABCDEFGHIJKLMNOPQRSTUVWXYZ_') and
              not verify(trim(param),
                'ABCDEFGHIJKLMNOPQRSTUVWXYZ_0123456789') and
              length(param) <=8; /* Ensure valid V8 macrovar name */
      if valid then call symput(param, trim(left(value)));
  end;
run;
```

Figure 8. Extract macro variables from SYSPARM string

Based on the values passed in the SYSPARM option, global macro variables are created by the framework. Table 1 below, shows the macro variables extracted by the DATA step seen in Figure 8 using the code used in Figure 7.

SCOPE	NAME	VALUE
GLOBAL	fpath	S:\SGF_Paper_Files\inputs\
GLOBAL	fname	NASA_access_log_Jul95
GLOBAL	outpath	S:\SGF_Paper_Files\output\
GLOBAL	outfile	NASA_MR
GLOBAL	Code_path	S:\SGF_Paper_Files\code\MRCode\
GLOBAL	blocksize	64

Table 1. Framework macro variables; Name, Values and Scope.

Once the Global macro variables have been set, they are available throughout the remainder of the SASReduce framework processing cycle.

FILESPLITTER:

To facilitate the parallel processing of raw files, similar to that of MapReduce or Hadoop, the SASReduce framework needs to provide the functionality to split a large raw file into smaller files. In the HDFS or GFS this task is done when the data is read into the file system, and the split sizes for each chunk are based on the minimum blocksize specified in a configuration file.

As SAS is unable to split a raw file without first ‘reading in’ the larger raw file (which defeats the purpose of what SASReduce is trying to achieve) an alternative approach was required.

Depending on the operating system being used (Windows or Unix/Linux - sorry Mainframe folk, I haven't covered this) two options are available to users.

On Unix/Linux the Operating System **SPLIT** command can be used. An example of the syntax can be seen in Figure 9 below:

```
split --bytes 64M Largefilename Smallfilename Prefix
```

Figure 9. Unix/Linux 'Split' command

In the example syntax for the Unix/Linux SPLIT command, above, the blocksize has been set to '64M' meaning that all the file splits will be 64Mb in size. This value can be set by developers using a parameter &BLOCKSIZE defined in the SYSPARM option on the batch call to SASReduce.

Users of Microsoft Windows sadly don't have a SPLIT command available to them like Unix/Linux users, since Windows natively doesn't provide this functionality. However, there are various free 3rd party programs that can provide this functionality. The Open Source FILESPLIT tool by DryDeadFish proves to be more than adequate for the task.

An example of the syntax provided by the FILESPLIT tool for Windows can be seen in figure 10 below:

```
filesplit -s Filepath/Filename 65536
```

Figure 10. Windows 'FILESPLIT' command using a 3rd Party tool

The syntax for the FILESPLIT tool differs from Unix/Linux SPLIT command in that it only needs to know the name of the input file as the tool will generate file splits (chunks) with the name of the original file appended with an iterative numerical suffix. An example of this can be seen in Figure 11.

The syntax to specify the BLOCKSIZE using the 3rd party FILESPLIT tool differs to the Unix/Linux split command. Here, the BLOCKSIZE is specified in bytes rather than megabytes, so the SASReduce framework needs to make a calculation to convert the BLOCKSIZE (assigned to the macro variable &BLOCKSIZE by developers in the SYSPARM option) from megabytes to bytes.

Note: Syntax will differ between 3rd party file splitting tools, so it is worth checking the documentation. It is also advisable to check with your systems administrator before installing any 3rd party software on a server.

Whilst SAS is unable to natively supply a command to split a raw file, it can programmatically allow the issuing of command line tools via the SYSTASK command. Figure 10 below, demonstrates the issuing of the commands in the framework. In this case the framework checks the host operating system using the automatic macro variable &SYSSCP**,** then issues the correct OS command.

```
%if &sysscp = 'WIN' %then %do;
      systask command "filesplit -s &fpath&fname 65536"
           taskname=splitfile status=splitstat wait;
    waitfor _all_ splitfile;
%end;
%else %do;
   systask command "split --bytes=64M &fpath&fname &fpath&fname chunk"
           taskname=splitfile status=splitstat wait;
   waitfor _all_ splitfile;
%end;
```

Figure 11. Determine OS type using &SYSSCP macro, then call appropriate SPLIT command

When this process has run successfully the original file and also the multiple splits, based on the BLOCKSIZE specified, will be available in the directory specified. Figure 12 below, shows an example of the directory listing after file splitting has taken place.

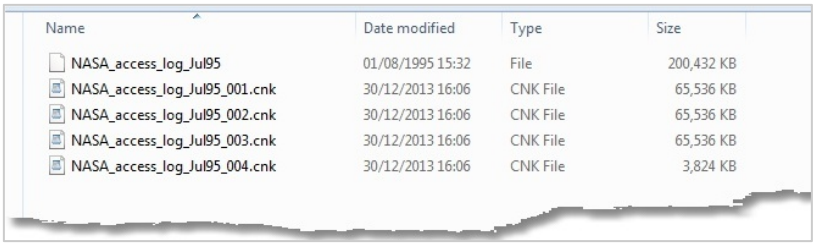


Figure 12. Directory listing showing the original file and the split files.

Once the large file has been split, a quick check by opening the file, can confirm the files have been split and the 'chunk' file, denoted with a 'CNK' filetype (in Windows), has been populated with data. The results of opening one of the 'chunk' files can be seen in Output 1, below:

```
199.72.81.55 - - [01/Jul/1995:00:00:01 -0400] "GET
/history/apollo/ HTTP/1.0" 200 6245
unicomp6.unicomp.net - - [01/Jul/1995:00:00:06 -0400] "GET
/shuttle/countdown/ HTTP/1.0" 200 3985
199.120.110.21 - - [01/Jul/1995:00:00:09 -0400] "GET
/shuttle/missions/sts-73/mission-sts-73.html HTTP/1.0" 200 4085
burger.letters.com - - [01/Jul/1995:00:00:11 -0400] "GET
/shuttle/countdown/liftoff.html HTTP/1.0" 304 0
199.120.110.21 - - [01/Jul/1995:00:00:11 -0400] "GET
/shuttle/missions/sts-73/sts-73-patch-small.gif HTTP/1.0" 200
4179
burger.letters.com - - [01/Jul/1995:00:00:12 -0400] "GET
/images/NASA-logosmall.gif HTTP/1.0" 304 0
burger.letters.com - - [01/Jul/1995:00:00:12 -0400] "GET
/shuttle/countdown/video/livevideo.gif HTTP/1.0" 200 0
205.212.115.106 - - [01/Jul/1995:00:00:12 -0400] "GET
/shuttle/countdown/countdown.html HTTP/1.0" 200 3985
d104.aa.net - - [01/Jul/1995:00:00:13 -0400] "GET
/shuttle/countdown/ HTTP/1.0" 200 3985
129.94.144.152 - - [01/Jul/1995:00:00:13 -0400] "GET / HTTP/1.0"
200 7074
unicomp6.unicomp.net - - [01/Jul/1995:00:00:14 -0400] "GET
/shuttle/countdown/count.gif HTTP/1.0" 200 40310
unicomp6.unicomp.net - - [01/Jul/1995:00:00:14 -0400] "GET
/images/NASA-logosmall.gif HTTP/1.0" 200 786
unicomp6.unicomp.net - - [01/Jul/1995:00:00:14 -0400] "GET
/images/KSC-logosmall.gif HTTP/1.0" 200 1204
d104.aa.net - - [01/Jul/1995:00:00:15 -0400] "GET
/shuttle/countdown/count.gif HTTP/1.0" 200 40310
```

Output 1. Web-log data from split data, showing IP addresses

Note: Developers are not required to check the file, rather this is only being described to clarify the file splitting process works, or for debugging purposes.

Once the 'larger' file has been split into multiple smaller chunks, the final part of the FILESPLITTER macro is to create a dataset containing the contents of the directory. The dataset created will be used later on in the framework to provide the number of MAP Tasks that need to be parallel processed.

The code used to obtain the directory listing is shown in Figure 13.

```
filename dirlist pipe "dir ""&fpath"" /b";
data work.dirlist;
  infile dirlist missover pad;
  input filename $255.;
  if filename =:"&fname._";
run;
```

Figure 13. Framework code used to create a directory listing containing the split files

The code uses the PIPE command to obtain the directory listing (DIR command) from the path specified in the batch parameters (SYSPARM). The output from the DIR command is then read into a SAS dataset. An IF statement with the LIKE operator '=:' is used to filter the results from the variable FILENAME, ensuring the results are 'like' those held in the macro variable &FNAME extracted from the batch SYSPARM option.

The dataset created by running the directory listing can be seen in Figure 14 below:

	filename
1	NASA_access_log_Jul95
2	NASA_access_log_Jul95_001.cnk
3	NASA_access_log_Jul95_002.cnk
4	NASA_access_log_Jul95_003.cnk
5	NASA_access_log_Jul95_004.cnk

Figure 14. Output Dataset created using directory listing code

PARALLEL PROCESSING:

The heart of the SASReduce framework is the processing of the multiple file chunks in parallel. Figure 15 provides a visual representation of how data is processed by the framework.

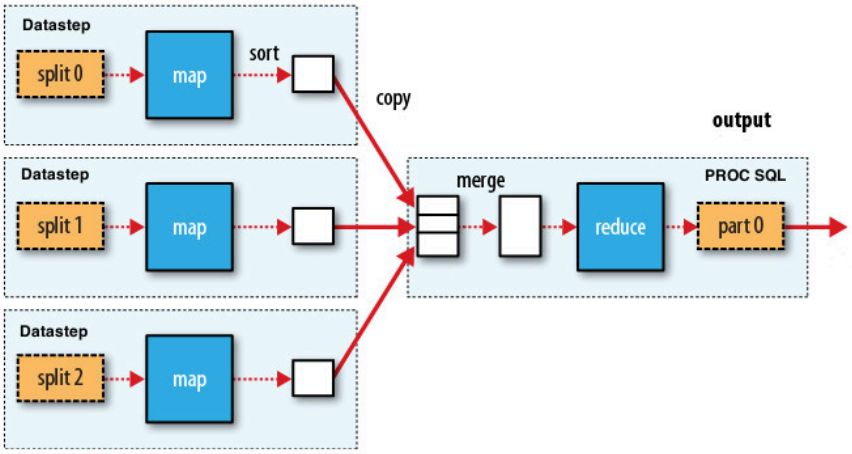


Figure 15. SASReduce Parallel processing architecture. Adapted from a diagram by Tom White for the O'Reilly publication 'Hadoop, the Definitive Guide'.

As discussed earlier, the SASReduce framework first requires a larger raw file to be broken up into individual smaller chunks (splits) in order to be processed by a number of SAS sessions, in parallel. The SAS MAP code, discussed later on in the paper, will read in the file splits in parallel and then sort the data based on a key variable provided by the user for MAP tasks.

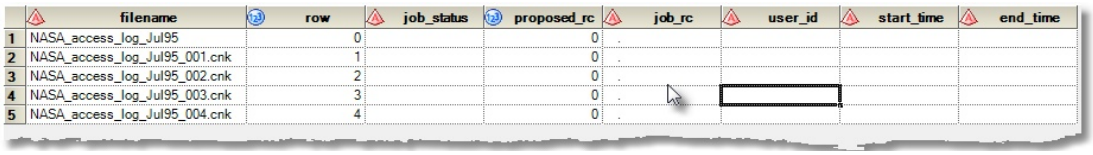
Data from the remote SAS sessions will then be passed back to the master SAS session (node) where the individual mapped datasets will be merged back into one dataset. The REDUCE task will then be called to provide a summarisation of the data and produce the desired results specified by the user/developer.

In order to provide some form of co-ordination for the file splits being processed in parallel, a SAS dataset is used as a control table for MAP Tasks.

The 'directory listing' dataset previously seen in Figure 14 is used as the basis to create the control table. The dataset contains the number of splits/chunks required for processing in parallel and the name of the chunks.

Along with the variable FILENAME, additional variables are created in the control table to provide both the framework and developers with information as to how the remote SAS sessions are progressing.

Figure 16 shows an example Control Table before job execution.



	filename	row	job_status	proposed_rc	job_rc	user_id	start_time	end_time
1	NASA_access_log_Jul95	0		0	.			
2	NASA_access_log_Jul95_001.cnk	1		0	.			
3	NASA_access_log_Jul95_002.cnk	2		0	.			
4	NASA_access_log_Jul95_003.cnk	3		0	.			
5	NASA_access_log_Jul95_004.cnk	4		0	.			

Figure 16. Example framework control table, pre run.

The next stage of the SASReduce framework is the LOOP_MAPPERS macro segment. This section of code checks to see the number of file splits needed to be processed in parallel by running a PROC SQL statement against the control table. An extract of the code can be seen in Figure 17 below:

```
proc sql noprint;
    select count(*) into :to_process from &control_table;
quit;
```

Figure 17. PROC SQL code to insert the number of splits into a macro variable

To determine the maximum concurrent sessions that can be run at any point in time the framework requires a value to be set in a macro variable. Developers can configure this value, however by default, it is set to be the maximum number of CPU's on the machine (determined using the automatic macro variable &SYSNCPU), minus one. One is subtracted from this value as the current 'master' SAS process is already occupying one CPU thread. The code to set the maximum concurrent sessions can be seen in Figure 18 below:

```
%let max_concurrent=%eval(&sysncpu -1); /* value based on number of cores
on SMP machine, minus one for the current master process */
```

Figure 18. Code to create a macro variable holding the max number of concurrent sessions based on CPU count -1

The framework then provides logic to determine if there are any 'file splits' to process using the value contained in the macro variable &TO_PROCESS. If the value is greater than zero a nested DO-LOOP is used to run further iterations from 1 to N number of splits based on the value from the macro variable &TO_PROCESS.

As there may be more file splits than the number of CPU's (and hence concurrent SAS sessions) available, a DO-WHILE loop is used to limit the number of sessions in progress (macro variable &IN_PROGRESS) to the value specified earlier in the &MAX_CONCURRENT macro variable. File splits with an iteration number higher than the value in the &MAX_CONCURRENT macro variable enter a queuing system called by the **QUEUE_CODE** macro.

An example of the code described above can be seen in Figure 19, below:

```
%if &to_process>0 %then
    %do i=1 %to &to_process;

   /* if we are at maximum concurrent sessions allowed, then wait and recheck */
        %do %while(&in_progress = &max_concurrent);
            %put ********************************************;
            %put **           WAIT FOR MAPPER I=&I            **;
            %put ********************************************;

            %queue_code
        %end;
```

Figure 19. DO-LOOP processing used to determine how many MAP tasks can be run based on the max concurrent sessions.

If the SAS session number is less than or equal to the number of maximum concurrent sessions (seen in Figure 19 above) and the macro variable &KILL_DONE is set to 'N' (indicating ~~the~~ that all jobs haven't finished or have been prematurely ended) the framework will then proceed to access the control table and insert the file split name (FILENAME) and the proposed return code (proposed_job_RC) into the macro variables &FNAME2 and &RC respectively.

The correct records from the control table are selected based on the value loop iteration number (&i). An example of the code described above can be seen in Figure 20, below:

```
/* must be below maximum number of sessions allowed to get here */
%if &kill_done = N %then
    %do;
        %ds_lock(&lib..&control_table);
           proc sql noprint;
               select filename,
                      proposed_job_rc
               into :fname2,
                    :rc from &control_table
               where row=&i;
           quit;
       %ds_unlock
```

Figure 20. SAS code to extract data from the control table and populate macro variables based on the framework iteration number.

The framework will then issue a SAS/CONNECT SIGNON command to establish a connection with a separate SAS session. The remote SAS session status (CMACVAR) is placed into a macro variable, which is checked by the QUEUE_CODE macro to obtain the number of concurrent sessions. The CONNECTIONWAIT option is set to NO to allow each SAS session to run asynchronously and then automatically terminate when the ENDRSUBMIT statement is run.

An example of the SIGNON command issued by the framework can be seen in Figure 21 below:

```
signon s&i cmacvar=s&i._status connectwait=no sascmd="!sascmd";
```

Figure 21. Example SAS/CONNECT SIGNON code.

Once the SIGNON to the remote SAS session has been established, the framework is then ready to run the user written MAP code in an asynchronous manner. **Note:** the use of the CREMOTE (CONNECTIONREMOTE) option to specify the ID for the remote SAS session to run the code in.

To replicate the data transfer mechanism of a MapReduce-type process, the framework makes use of the INHERITLIB option on the RSUBMIT statement. INHERITLIB libraries defined in the client (master) SAS session can be inherited by a remote SAS server session. In this case, the WORK library from the master SAS session is being inherited by the spawned SAS sessions, and all datasets being created in the WORK library on the remote sessions will be available in the WORK library on the master session. Figure 22 on page 10 provides an example of the code described above.

It is worth noting that when running on a SMP machine a single SAS library could be used to store all the datasets being created on each spawned SAS session. However this would lead to all 'mapped' datasets being stored permanently on the box until they are removed. A scenario where this might be desirable would be debugging a process. However, the aim of the SASReduce framework was to provide similar functionality to that of a MapReduce-like processing environment such systems as Hadoop. In those environments the intermediate data from MAP tasks are not stored permanently, so the SAS WORK library is an ideal solution.

```
/* REMOTE SUBMIT CODE */
rsubmit cremote=s&i cpersist=no inheritlib=(work=swork);
    /* assign libnames */
    libname &lib &libpath;
    %macro map;
        %put ********************************;
        %put task running in parallel         ;
        %put ********************************;

        %let Time = %sysfunc(time(),time8.0);
        /* Insert task start time into Table */
        proc sql;
            update &lib..&control_table
            set start_time = "&Time" where row = %eval(&i-1);
        quit;

        filename srvlog "&fpath&fname._00&i..cnk" LRECL=1500;
        option obs=max;

        /* MAP */

        /* Call the user written 'Map' code */
        %include "&codepath.Map.sas";

        /* Sort step (below) could be removed if developer/user happy to
             let PROC SQL sort the data in the 'reducer' phase. */
        proc sort data = &syslast;
             by &key;
         run;

        %sysrput &status_var=&proposed_job_rc;
    %mend map;
    %map;
endrsubmit;
```

Figure 22. Remote submit and MAP macro code.

Figure 22 above, also illustrates the use of inserting data into the control table to show when a MAP task has started, along with using the parameters from the SYSPARM option from the batch submission to build a generic filename statement used in the MAP.SAS code described later on in the paper.

The next section of the LOOP_MAPPERS macro checks the macro variable &KILL_DONE to obtain a value. If the value remains N, a DO_UNTIL loop is used to call the QUEUE_CODE macro. This process will be repeated until the value of the macro variable &IN_PROGRESS is set to **0** (when all the MAP tasks have finished). An example of the code described above can be seen in Figure 23:

```
/* finally wait for the mappers to finish */
   %if &kill_done=N %then
       %do %until(&in_progress=0);
            %put **********************;
            %put WAIT FOR FINAL I=&I;
            %put **********************;

            %queue_code;
       %end;
%mend loop_mappers;
%loop_mappers;
```

Figure 23. DO-LOOP code checking for Kill Code or Progress counter.

%QUEUE_CODE:

The purpose of the QUEUE_CODE macro within the framework is to provide a mechanism for queuing MAP tasks when the number of required MAP tasks exceeds the maximum concurrent SAS sessions specified by the framework. This works on a First-In-First-Out (FIFO) basis, whereby the status of the variable &&S&J._STATUS (returned from a remote SAS session using the CMACVAR option) is checked. Depending on status code, the variable JOB_STATUS within the control table is updated with values that are later read from the control table by the framework code.

Figure 24 provides an excerpt from the QUEUE_CODE macro demonstrating the checking of the status variable, and then the updating of the control table.

The framework also makes use of a macro DS_LOCK which locks the control table dataset whilst it is being updated, and then releases the lock when the updating has taken place. This ensures only one MAP task can update the table at a time, and also ensures no errors are encountered that may stop tasks from running.

```
%if &&s&j._status = 2 %then
    %do;
        %dset_lock(&lib..&control_table);
        %let Time = %sysfunc(time(),time8.0);
        proc sql;
            update &control_table
            set job_status = "I",
                end_time = "&Time"
            where row = &j-1;
        quit
        %dset_unlock
    %end;
%if &&s&j._rc=9999 and &kill_done ne Y %then
    %do;
        %let i=&to_process;
        %let Time = %sysfunc(time(),time8.0);
        killtask _all_;
        %dset_lock(&lib..&control_table);
        proc sql noprint;
            update &control_table
            set job_status = "K",
                job_rc=9999,
                end_time = "&Time"
            where job_status in ('','I');
        quit;
        %dset_unlock
        %let kill_done=Y;
    %end;
```

Figure 24. Extract from QUEUE_CODE macro

Table 2 below shows the values that can be populated in the variable JOB_STATUS in the control table via the framework.

VALUE	DESCRIPTION
C	Completed
F	Failed
I	In Progress
K	Killed

Table 2. Possible Job Status codes.

The final part of the QUEUE_CODE macro uses PROC SQL to count the number of rows in the control table where the variable JOB_STATUS has been set to **I** (In Progress). The results from the query are then inserted into the macro variable &IN_PROGRESS, which is checked by the LOOP_MAPPERS macro during its iteration cycle. This process will be repeated until all the observations in the control table have been populated, indicating all MAP Tasks have finished.

Figure 25, below, provides an example of how the control table is populated by the framework whilst processing is taking place.

	filename	row	job_status	proposed_rc	job_rc	user_id	start_time	end_time
1	NASA_access_log_Jul95_001.cnk	1	C	0	8	David Moors	13:55:40	13:55:45
2	NASA_access_log_Jul95_002.cnk	2	C	0	8	David Moors	13:55:41	13:55:47
3	NASA_access_log_Jul95_003.cnk	3	C	0	8	David Moors	13:55:43	13:55:48
4	NASA_access_log_Jul95_004.cnk	4	I	0		David Moors	13:55:42	

Figure 25. Control Table being populated during SASReduce processing

%MAP MACRO:

The purpose of the MAP macro is for the framework to provide a layer of abstraction, whereby all the necessary MAP related code is automated, and developers only have concentrate on writing the MAP.SAS code.

The macro code can be seen in Figure 22 on page 10, above, and consists of the creation of a generic FILENAME statement used to retrieve the specified file split using the LOOP_MAPPER macro iteration loop number. Using the macro variables &FNAME and &FPATH defined in the SYSPARM option on the batch process call, the macro sets the file name and file path for the external file.

A %INCLUDE statement is then issued, calling the MAP.SAS custom code written by the developer. The code can be seen in Figure 29 on page 14.

Finally a PROC SORT is performed on the data to sort the mapped data by the '**Key'** variable defined by the developer in the MAP.SAS user-written code.

When the MAP Tasks have run successfully the output will be available in the WORK library on the master node via the use of the INHERITLIB option on the RSUBMIT statement. Figure 26 below is an example of the temporary datasets created during the MAP process:

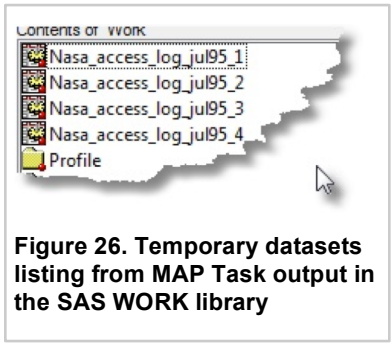


Figure 26. Temporary datasets listing from MAP Task output in the SAS WORK library

An example of a SAS VIEW created by the MAP Tasks process can be seen in Figure 27 below: **Note**: This is for illustrative purposes only, as the SAS VIEWS created are intermediate files and are deleted from the system once the framework has completed successfully.

	ip	val
1	buffnet3.buffnet.net	1
2	ix-wh3-10.ix.netcom.com	1
3	ad04-002.compuserve.com	1
4	buffnet3.buffnet.net	1
5	a1p09.connect.net	1
6	a1p09.connect.net	1
7	134.71.168.43	1
8	134.71.168.43	1
9	134.71.168.43	1
10	134.71.168.43	1
11	134.71.168.43	1
12	134.71.168.43	1
13	ad04-002.compuserve.com	1
14	199.233.81.1	1
15	www-b3.proxy.aol.com	1
16	a10404.hcc.hawaii.edu	1

Figure 27. Temporary SAS View created by MAP Tasks

%REDUCE MACRO:

As with the MAP macro described above, the REDUCE macro aims to provide a level of abstraction, leaving the user to concentrate on writing the REDUCE.SAS code to obtain their desired output.

The REDUCE macro appends all the temporary datasets created by the MAP tasks using a macro DO-LOOP to cycle through the number of mappers. A dataset view is created in the WORK library to reduce unnecessary file footprint on the server. An example of the temporary SAS VIEW can be seen in Figure 28:

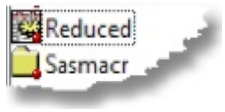


Figure 28. Temporary SAS View created by the REDUCE Task

The use of a temporary data structure is inline with that of a MapReduce-type processing framework e.g. Hadoop, whereby intermediate data is all temporary on disk The macro then calls the REDUCE.SAS user written code via a %INCLUDE statement. An example of the code can be seen in Figure 29 below:

```
%macro reduce;
    /*-------------------------------------------------------------
     create view to 'mapped' datasets - view used to reduce dataset
     footprint */
    -------------------------------------------------------------*/
    /* NOTE: the step below should not be altered.....           */
    /*-------------------------------------------------------------*/
    data work.reduced / view=work.reduced;
        set %do i = 1 %to &mappers;
                work.&fname._&i
            %end;
        ;
    run;

    /*-----------------------------------------------------------------------
      Step below should be changed by the user to provided desired outputs
    -----------------------------------------------------------------------*/
    %let Rlib = reduced;
    libname &Rlib "&outpath";

    /* Call the user written 'Reduce' code */
    %include "&codepath.Reduce.sas";
%mend reduce;
%reduce;
```

Figure 29. REDUCE Macro code

The SAS log, seen in Output 2 below, shows the combining of the temporary outputs of the Mappers (SAS VIEWS) to create the intermediate final REDUCE table specified by the developer.

```
NOTE: There were 612364 observations read from the data set WORK.NASA_ACCESS_LOG_JUL95_1.
NOTE: There were 617916 observations read from the data set WORK.NASA_ACCESS_LOG_JUL95_2.
NOTE: There were 619990 observations read from the data set WORK.NASA_ACCESS_LOG_JUL95_3.
NOTE: There were 36484 observations read from the data set WORK.NASA_ACCESS_LOG_JUL95_4.
NOTE: Table REDUCED.NASA_MR created, with 81960 rows and 2 columns.
```

Output 2. SAS Log from REDUCE Macro task

MAP.SAS – USER WRITTEN CODE:

The MAP.SAS code creates a SAS DATA step view using an INFILE statement to call the SRVLOG generic FILEREF described earlier.

Developers specify the 'Key' in a macro variable, which is the variable that forms part of a MapReduce key/value pair. The 'value' part of the key/value pair is again specified by the user, in this case the variable VAL containing the value **1**. The variable VAL will be summarised as part of the REDUCE process, described later in the paper.

The code has been written so that developers only have to be concerned about deciding what they require the key/values in the data to be. The framework takes care of the naming of datasets, based on the values passed in the SYSPARM option on the batch submission.

Figure 30 below, is an example of user-written MAP code.

```
/* using inherited libnames to send data from 'remote work' library
 to 'client work' library */
data swork.&fname._&i / view=swork.&fname._&i;
%** code block to amend starts here **;

/* specify 'key' variable */
    %let key = ip;

    infile srvlog;
    input &key.:$25.;
    val=1;
    /* NOTE: check the code below - probably incorrect */
    if index(&key.,'.') = 0 then delete;
%** code block to amend finished here **;
run;
```

Figure 30. User written MAP code

REDUCE.SAS – USER WRITTEN CODE:

The REDUCE.SAS program creates the final summarised output as requested by the developer. As with the MAP.SAS code described above, the framework handles the naming and location of the datasets, based on values passed in the SYSPARM batch option.

Developers are only required to specify the variables and calculations they require for the final output. In the case of Figure 31 below, this is a simple summarisation of the variable VAL along with grouping the variable IP.

The dataset view created earlier in the REDUCE macro section of the framework is dropped as good practice. However, this is not necessarily required as all intermediate files in the SASReduce framework are temporary datasets.

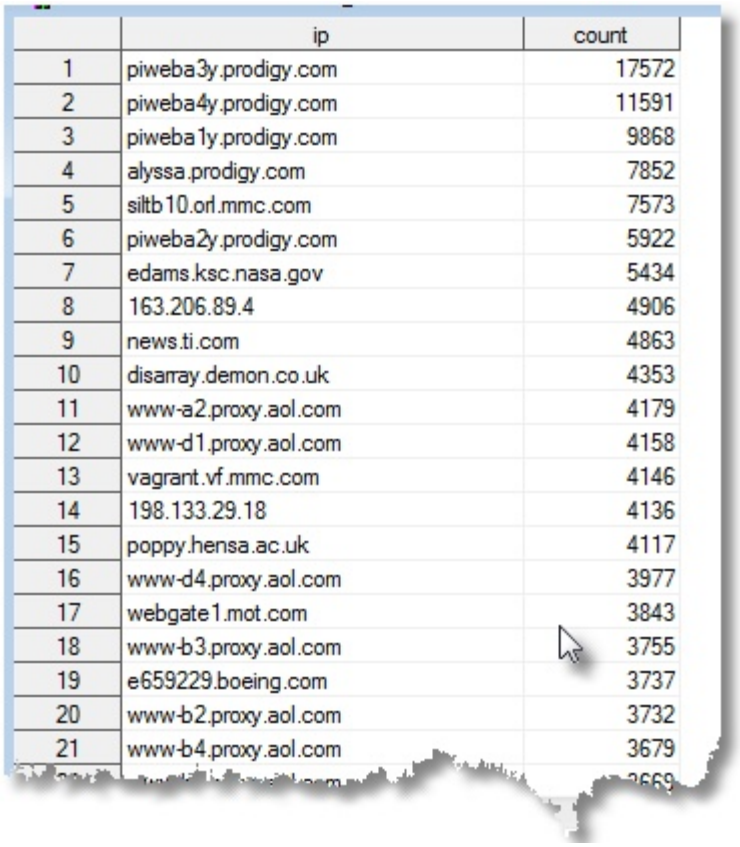
```
proc sql;
    create table &Rlib..&outfile as
    %** user code block to amend starts here **;
        select &key, sum(val) as count
        from &syslast
        group by &key
        order by count desc;
    %** user code block to amend finished here **;

    /* clean up work area */
    drop view work.reduced;
quit;
```

Figure 31. User written REDUCE code

RESULTS

The results from running the SASReduce process is a dataset containing the key/value pairs specified by the developer in the REDUCE.SAS code, discussed above. Using the example NASA data discussed throughout this paper, the final data contains the count of all IP addresses that have accessed the NASA web-server. Figure 32 below shows an excerpt from the final dataset.



	ip	count
1	piweba3y.prodigy.com	17572
2	piweba4y.prodigy.com	11591
3	piweba1y.prodigy.com	9868
4	alyssa.prodigy.com	7852
5	siltb10.orl.mmc.com	7573
6	piweba2y.prodigy.com	5922
7	edams.ksc.nasa.gov	5434
8	163.206.89.4	4906
9	news.ti.com	4863
10	disarray.demon.co.uk	4353
11	www-a2.proxy.aol.com	4179
12	www-d1.proxy.aol.com	4158
13	vagrant.vf.mmc.com	4146
14	198.133.29.18	4136
15	poppy.hensa.ac.uk	4117
16	www-d4.proxy.aol.com	3977
17	webgate1.mot.com	3843
18	www-b3.proxy.aol.com	3755
19	e659229.boeing.com	3737
20	www-b2.proxy.aol.com	3732
21	www-b4.proxy.aol.com	3679

Figure 32. Final dataset showing the count of IP addresses.

CONCLUSION

In this paper we have been introduced to the basic concepts of MapReduce and how this data processing model can be easily replicated with SAS software. We have also been introduced to SASReduce, a framework that provides some MapReduce-like operations for data processing, including: batch submission, parallel processing of MAP tasks, and a queuing mechanism for multiple MAP tasks.

Future work could focus on extending the SASReduce framework to process MAP tasks on individual machines rather than on a single SMP machine, and extending to the framework to cater for more than one 'reducer' than currently provided.

REFERENCES

Buckecker, Michelle.M. "Parallel Processing Hands-On Workshop". Proceedings of the Twenty-Ninth Annual SAS® User Group International (SUGI) Conference (2009)

Dean & Ghemawat. MapReduce: Simplified Data Processing on Large Clusters, OSDI 2004.

DryDeadFish: FileSplit software available from: http://www.drydeadfish.co.uk/filesplit/

NASA Weblog data available from: ftp://ita.ee.lbl.gov/traces/NASA_access_log_Jul95.gz

Thacher, Clarke. "Make Your SAS Code Environmentally Aware" Proceedings of the SAS® Global Forum 2010 Conference

White, Tom. 2012. Hadoop: The Definitive Guide. O'Reilly Media Inc

Whitlock, Ian. "How to Think Through the SAS DATA Step". Proceedings of the Thirty-First Annual SAS® User Group Internation (SUGI) Conference (2006)

ACKNOWLEDGMENTS

CONTACT INFORMATION

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

David Moors
Whitehound Limited
21 Lyons Lane
Appleton, Cheshire, WA4 5JG
UK
whitehound.limited@gmail.com

APPENDIX 1

```python
#!/usr/bin/env python
# Map Task

import sys
for line in sys.stdin:
    ip = line[1:15]
    for key in ip:
        value = 1
        print("%s\t%d"%(key, value))

# Reduce Task

import sys
last_key = None
running_total = 0

for input_line in sys.stdin:
    input_line = input_line.strip()
    this_key, value = input_line.split("\t", 1)
    value = int(value)
    if last_key == this_key:
        running_total += value
        else:
        if last_key:
            print("%s\t%d" % (last_key, running_total) )
            running_total = value
            last_key = this_key

if last_key == this_key:
    print("%s\t%d" % (last_key, running_total))
```