The SQL Optimizer Project: _Method and _Tree in SAS®9.1

Russ Lavery : Contractor for Numeric, LLC
(Thanks to Paul Sherman)

ABSTRACT

This paper discusses two little known options of SAS® Proc SQL: _method and _tree. The main body of information, and the major opportunity to learn about the topics, exists in the heavily annotated appendix. Efficient use of this material might involve reading, and printing, this paper then working through the appendix. This paper will also attempt to describe some of the logic that the Optimizer employs. Due to the page limitations of the WUSS conference, the full paper can not be put on the CD. The full paper is in the SUGI 30 proceedings.

Proc SQL has a powerful subroutine, the SQL Optimizer, that examines submitted SQL code and the state of the system (file size, index presence, buffersize, sort order etc). The SQL Optimizer creates a “run plan” for optimally running the query. Run plans describe executable programs that the Optimizer will create to produce the desired output. These executable programs can be quite complicated and often involve the creating, sorting and merging of many temporary files. Consistent with the Optimizer’s goal of minimizing run times, the executable programs will trim variables and observations from the input file(s)/working file(s) as soon as they can be removed.

Many details of the run plan can be determined by using two Proc SQL options (_Method and _Tree) and this paper explains output from these two options. _Method and _Tree produce different output that present different aspects of the run plan. Learning to interpret _Method and _Tree can help programmers explain why small variations in code, or system conditions, can cause substantial variations in run time.

INTRODUCTION

This paper introduces two little known options of SAS Proc SQL: _Method and _Tree. The majority of the information, and the opportunity to learn about the topics, exists in the heavily annotated appendix. Reading the appendix is strongly recommended.

In SQL, you tell SQL what you want for results, not how you want the results produced. SAS SQL has a powerful subroutine, the SQL Optimizer, that decides how the SQL query should be executed in order to minimize run time. The Optimizer examines submitted SQL code and characteristics of the SAS system and then creates efficient executable statements for the submitted query. The created code can be quite complicated and often involves the creating, sorting and merging of many temporary files as well as the trimming of variables and observation at times that will minimize run time.

All versions of SQL have Optimizers and, while they perform very well, performance can sometimes be improved by manual intervention (AKA different coding of the query). Tuning SQL, or coding for optimum SQL efficiency, requires that the programmer understanding the logic of the Optimizer’s choices and to change her/his submitted code in a way that allows the Optimizer to make better choices. Basic to the tuning process is the understanding of what the Optimizer did when it ran a “too slow” SQL Query. _Method and _Tree show much of this information.

Unfortunately _Method and _Tree options produce large amounts of output. This paper will present an overview of the subject and the reader is encouraged to work thorough the annotated logs in the appendix of this paper (included on the on the CD). The annotated log files, in the appendix, is one of the larger, more detailed, collections of annotated _Method and _Tree output.

This is part of a planned series of papers on the SAS SQL Optimizer. The papers will build on each other and, hopefully, create a coherent body of knowledge on the Optimizer.

THE LIST OF KNOWN MESSAGES AND BASIC SQL PROCESSES

_Method and _Tree will show much about how a query executed, but use many abbreviations to indicate SAS SQL processes. In order to understand the cost penalties/implications of a particular execution plan, we must understand the abbreviations and some of the details of the processes used by SAS SQL.

A comment on, and apology for, the incomplete nature of this paper is in order. It is only too likely that the Optimizer currently has more subtlety than has been uncovered by the author and presented here. Additionally, SAS Inc. is constantly improving its products. As time goes by, the Optimizer will only become more powerful and more subtle and this paper will only become more incomplete.

TOP-TO-BOTTOM READ:

In certain situations SQL will perform a top to bottom read of the data. It will often do so in a query that does not have a where clause, or has a where clause without a usable index. The query might not have a usable index because 1) there is no index on the variable in the where clause, or 2) the code/syntax in the where clause might have prevented the Optimizer from using the index. SAS has put lots of time into making a top-to-bottom read fast and has been successful. Each read of a single observation is fast, but when millions of observations must be read, the total time (time=Number of Obs. * seconds per observation read) can still be unacceptably long.

Here are two examples of queries that will be executed in a top to bottom read.

Proc sql; Select * from dsn;	Proc sql;/*no index on sub*/ Select * from dsn WHERE SUB=”001”;

EQUIJOIN

An equijoin is the name for a join that has an equality in the where clause. SAS has not implemented the code that an academic might consider a “true equijoin” however it has some fast techniques that the Optimizer can use on equality relationships. As a result, the SQL Optimizer is often able to process equijoins quickly. Some SUGI/NESUG articles have shown techniques to speed up queries by converting them to equijoins.

Proc sql;/*Equijoin*/ Select * from dsn WHERE SUB=”001”;	Proc sql;/*not an Equijoin*/ Select * from dsn WHERE SUB LE ”001”;
Proc SQL; /*Equijioin*/ SELECT L.SUBJID, L.NAME, R.AGE FROM LeftT as L, Right as R Where L.subjid=r.subjid;	Proc SQL; /*not Equijoin*/ SELECT L.NAME, L.Age, R.Name, R.Age FROM Left as L, RightT as R Where L.Age GE R.Age;

SQL, whenever it can save time, delegates operations to lower SAS processes. In simple cases of an equijoin, like where Age=5, the where clause can be delegated to the data engine. The data engine will perform this equijoin (think of it as passing only obs where Age=5) and pass the result to SQL. If the where clause contains code like where age*12=60, all observations would be brought into SQL The multiplication and filtering would happen in SQL. The data engine is usually not smart enough to perform multiplication/division and similar operations.

CARTESIAN PRODUCT OR STEP-LOOP JOIN

Consider merging two tables (LeftT and RightT) in a SQL from clause, one table on the left (LeftT) of the comma and one on the right (RightT). LeftT stands for the table on the left of the comma and RightT stands for the table on the right of the comma. Cartesian products and step loops are related merging processes and the SQL Optimizer employs them as a last resort. They are slow and very much to be avoided.

For these processes, a page of data is first read from the left table and then as much data as can fit in memory is read from the right table. All merges are made (between any observation in the page from LeftT and all observations in memory that came from RightT). The results are then output. Then the observations from RightT are flushed from memory and a new read of RightT pulls in as many observations as can fit in memory. All possible matches are made (between any observation in the page from LeftT and all observations in memory that came from RightT) and results are output. This process continues until SQL has looped through all of the table RightT.

Then SQL takes a step in the left table and reads a new page of data from LeftT into memory. The process of looping through right table is repeated for the second page from the left table. Then a third page is read from the left table and the looping through the right hand table is performed again. The process continues until all the observations in the left table have been read and matched against every observation in the right hand table.

If there is a where clause in the query, it will be applied before the observations are output. SQL joins that are not based on an equality are candidates for the step loop process processes and are therefore to be avoided.

Proc sql; Select * from LeftT,RightT;	Proc sql;/*no index on sub*/ Select * from LeftT, RightT Where LeftT.sub < RightT.sub;

Both examples above are likely to be executed using a step-loop join (depending on system conditions and the decision of the SQL Optimizer).

INDEX JOIN

Even if an index exists on a variable in the where clause, the where clause (the code used for the query logic) can "prevent" the Optimizer from using the index. Additionally, even if the index exists and the code in the where clause does not disable the index, the Optimizer may decide not to use the index. The decision logic for the Optimizer is complex, but a firm index rule is: if index merge will return more than 15% of the indexed file to the result file, the index will not be used. In this case, there is a faster way to perform the query and the Optimzer will look for it. The Optimizer (or the data engine) can access metadata and will estimate output file sizes.

In the case of a one file select (see left box below) the Optimzer checks the percentage of observations that come from PA and their distribution in the file. If the Percentage is small, SQL reads the index file on the indexed variable "state". It locates the desired level(s) of "state" in the index and reads, from that index-observation, the hard drive page number(s) that contain observations where state="PA". In order, the page number(s) are retrieved from the index, page number(s) are passed to the disc controller and data is sent back to the CPU. As each page of observations is received by the CPU, it is parsed and observations with state="PA" are passed back to the working file for the query. SAS keeps track of the most recently read page, as a technique to minimize disk reads. If a page contains several observations that meet the where clause (e.g. state="PA"), a new page will not be read from the hard drive until all the observations in that page from PA have been sent to the query working file.

In an index merge of two files (see right box below), an index exists and the where clause code must be written in a way that allows the SQL Optimizer to use it. In the appendix, several different queries are used to test how indexes are used by the SQL Optimizer. The basic process for index join is that one file is read from top to bottom and the matching observations in are read from the other file via an index lookup.

Proc sql;/*index on state*/ Select * from LeftT Where state="PA";	Proc sql;/*L_I_sub indexed in LeftT*/ Select * from LeftT , RightT Where Lft_tbl.L_I_sub = Rgt_tbl.subj

For more of an explanation of the process in the right box above; Assume two tables, LeftT and RightT, with an index on the variable L_I_sub in LeftT. The basic index merge process is that we are reading the right file from top to bottom and using an index to find observation(s) with matching subject Ids from left.

1) Select the next observation from RightT (at start, this is the first observation in the data set)
   -if end of file, stop
2) Pass subj from RightT to the SAS index subroutine
3) Search the index on the variable L_I_sub in the file LeftT for the value of subj
4) If found, go to disk and return the required fields for that observation from LeftT
   - output and go to 1)
5) If not found – go to 1)

If the information required by the query is in the index file itself the Optimizer will simply access the index, and not proceed to access the file associated with the index. This situation usually arises in queries when the where clause tests for existence of a match in another file (where a value in the variable subj in RightT is also in LeftT and Left_T has an index on subj). The fact that the Optimizer has automated this speed feature is just one indication of the level of detail that has gone into the designing and programming Optimizer, and of the difficulty of understanding it.

HASHING JOIN

Hashing can be a very fast technique and is automatically considered by the Optimizer. SQL hashing has been installed since V6.08 but since the Optimizer evaluates, and implements, the hashing technique without the programmer's intervention, its existence is not well known. General information on Hashing can be found in articles by Dr. Paul Dorfman in SUGI and NESUG online proceedings.

Hashing will not be considered as a join technique unless certain conditions are met. The SQL Optimizer accesses metadata on the file and "takes a good guess" at the size of the files it needs to join. After removing unneeded observations and variables, the Optimizer checks to see if 1% of the smaller of the two files being joined will fit into one memory buffer. If the smaller file appears to fit, the Optimizer will attempt a hash join. If the smaller file is too large, in relation to the buffer size, hashing will not be selected. A programmer can influence the Optimizer's choice of hashing as a merge technique by manually changing the buffer size with a SAS option.

SQL performs a hash join in the following way. The SQL Optimizer determines which of the two tables is smaller (after keeping only the appropriate variables from the select statement) and checks that smaller table size against the buffer. If the table meets the size criteria, it is loaded into a tree-like structure (a hash table) in memory. The structure of the hash object and the fact that is memory resident, allows for very fast searching. Then SQL processes the large file, from top to bottom, and for every observation that satisfies the where clause, it performs a HASH table lookup for the observation. Details of "data step" hashing are given in an article titled "An Annotated Guide: Resource use of common SAS Procedures" in the NESUG 2004 proceedings.

HASH & INDEX & WHERE USE

The Optimizer will dynamically adjust to new information. In certain situations it will switch from one join method to another-in the middle of execution - and create a hybrid join method. One such example is in the appendix (see examples 9A to 9D) and indicates that SQL simultaneously used a hash join and an index to produce the results of the query.

What happened is that, after tentatively trimming rows and columns from both files, the Optimizer estimated that 1%, of the smaller of the files being joined, would fit in a buffer. This is a strong hint/instruction for the Optimizer to use a hash join and so SQL loaded the smaller table into a hash table.

The Optimizer, as the hash table was being created, counted the number of unique key-variable values being loaded into the hash table. The number of unique values loaded into the hash table was found to be a small number (maybe below 1024) and the Optimizer dynamically changed the plan to take account of this information. In general, if there are fairly few unique values key in the hash table, the Optimizer will take the values from the hash table and use them to build an "in" phrase for a where clause (e.g. where state in("PA", "TX")).

SQL will then use the where clause to select observations but the Optimizer will again re-evaluate it's options in light of currently known information. The method selected for the join can be a top to bottom read or an indexed lookup. This adjustment of code to the details of a particular query is complex, dynamic and automatic. It can be seen in examples 9A to 9D.

SORT MERGE JOIN

This is similar, but not identical to, the data step merge. Under certain situations, the Optimizer determines that the fastest way to execute the query is to sort the tables and process both tables from top to bottom, using a merge that is similar (only similar) to that used by the DATA Step. This SQL merge will produce a Cartesian product, unlike the data step merge, and it does this by looping within the BY-group variables. SQL processes a page of data from the left table and loops through the appropriate by group right table.

GROUPING

Grouping, or aggregating observations, is a multi-step process and can take some time.

SELECT

Select statements specifies variables in the final data set. The optimizer, as part of it's run plan, creates "temporary working files" that are called result sets. Select logic is executed dynamically and as early as possible, keeping results sets small.

In the code below, only the variables name, age and sex are all brought into the original SQL query space (AKA result set). This initial removal of variables (height, weight) is handled by the data step engine and functions much like a keep option on a data set. Height and weight never become part of a SQL result set.

Observations with sex NE "M" are filtered out during the initial read of the data and that variable (sex) is eliminated from the query space, by the Optimizer, after completion of the read. After the completion of the first read, the result set contains name and age. The result set is then sorted (by calling Proc Sort) by age. After the data is sorted, age is not required and the Optimizer eliminates that variable from the result set.

```
proc sql _method _tree;
create table lookat as select name from sashelp.class
where sex="M"   order by age;
```

As the above explanation details, the Optimizer has automated "Good Programming Practices" and eliminates both variables, and observations, as soon as it can.

HAVING

The having statement is very useful to SQL programmers but requires that SQL perform several steps. Please examine the code below.

```
proc sql _method _tree;
   title "this illustrates a having clause";
       select name, sex, age
           from sashelp.class
       group by sex
       having age=max(age);
   quit;
```

In the code above, SQL must process the entire table sashelp.class, reading in the only the three variables in the select clause. SQL stores the observations in a result set (a temporary table that the SQL Optimizer directs be created). Then SQL makes a pass through the temporary table to find the max age, within each group, and tries to store that information in a "pipe line". As a speed/storage technique, the Optimizer tries to avoid the creation of temporary files.

Sometimes applying a "having" requires additional passes through the working data set (AKA the result set) to check the having criteria against each observation. If there is no Note in the log mentioning re-merging, the Optimizer was able to produce the desired result in one pass using "pipe lines" to apply the having criteria. The query above produces the log below, where the word remerging indicates an additional pass was required to find the observations with the maximum age for each gender.

```
NOTE: The query requires remerging summary statistics back with the original data.
NOTE: SQL execution methods chosen are:
sqxslct
          sqxsumg
              sqxsort
                  sqxsrc( SASHELP.CLASS )
```

DISTINCT

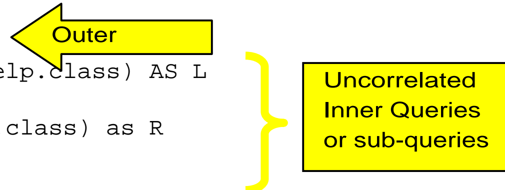
SQL usually implements distinct-ing by passing the result set to a Proc Sort with a nodup/nodupkey option. Distinct-ing is usually performed late in the query process as an additional pass through the data. This is a sorting and sorts are to be avoided because they take both time and space.

Under certain conditions (see examples 6A - 6D), the Optimizer can eliminate this last pass through if the query is distinct-ing a variable that has a unique index. The elimination of the distinct-ing saves time. There is an example of this, in compare-and-contrast format, in the appendix. It would be appropriate to use this as an example of how much effort has been put into making the Optimizer produce fast code. This situation does not happen often but the Optimizer has logic to help the programmer when it occurs.

UNCORRELATED SUB-QUERY

The code in an uncorrelated sub-query is processed just once by the SQL Optimzer. The results of this first evaluation are held in a result set until they are needed by the outer query. The code below is an example of an uncorrelated query. The result sets L and R are created once and accessed many times.

```
proc sql _method _tree;
 *title show inner join merge  using a comma;
 create table ex5 as
 select coalesce (l.name, r.name) as name
         FROM ( select Name, Height from sashelp.class) AS L
           ,
              ( select Name, sex from sashelp.class) as R
        where l.name  =r.name;
```



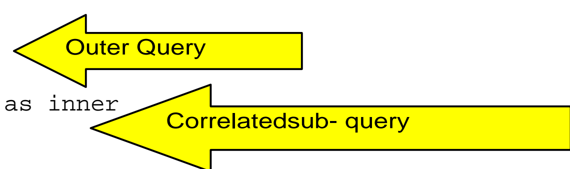
CORRELATED SUB-QUERY

A Correlated sub-query is processed differently from an un-correlated sub-query and again shows the power of the SQL Optimizer. A correlated sub-query uses information from each of the observations in the outer query to drive a "look up" process (a SQL query) against another table. In the worst case, SQL might have to execute the "look up process" for each row in the outer table. The look-up might have to be executed for every observation in the outer query but the Optimizer creates code that avoids that, whenever possible.

In the query below, as SQL processes each observation in the outer query, it seems to be passing the gender to the subquery and asking the subquery to find the maximum age for the current (in outer) value of gender. In fact, the Optimizer will process the sub-query for the first observation and store the results in a result set that is both temporary and indexed.

When additional observations from "outer" are processed the Optimizer first tries to find the needed information (in this case, has SQL calculated max age for that sex before) in the temporary, indexed result set. If it can find the required information in the temporary indexed result set, it takes information from the temporary indexed result set and does not execute the sub-query. If the information is not in the temporary indexed result set, SQL will run the sub-query, pass results to the outer query and then add the results of the query to the indexed result set. The temporary indexed result set grows in size as unique values of the equality variable are found in the outer file. When the query is done, the temporary indexed result set is deleted from the work library. The query below produces the _method output that follows it.

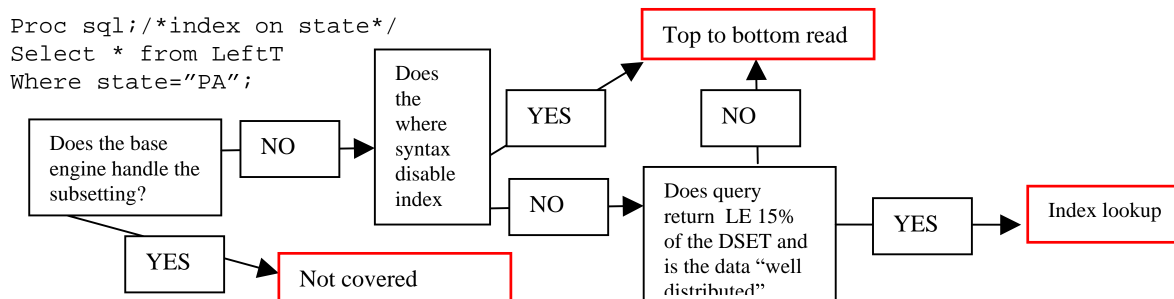
```
proc sql _METHOD _TREE;
TITLE "A SIMPLE CORRELATED QUERY";
select * from sashelp.class as Outer
 Where  Outer.AGE =
  (select Max(age) from sashelp.class as inner
        where outer.sex=inner.sex);
quit;
```




```
NOTE: SQL execution methods chosen are:
  Sqxslct
      Sqxfil
          sqxsrc( SASHELP.CLASS(alias = OUTER) )
NOTE: SQL subquery execution methods chosen are:
  Sqxsubq
      Sqxsumn
         sqxsrc( SASHELP.CLASS(alias = INNER)
```

SIMPLE SUBSETTING LOGIC

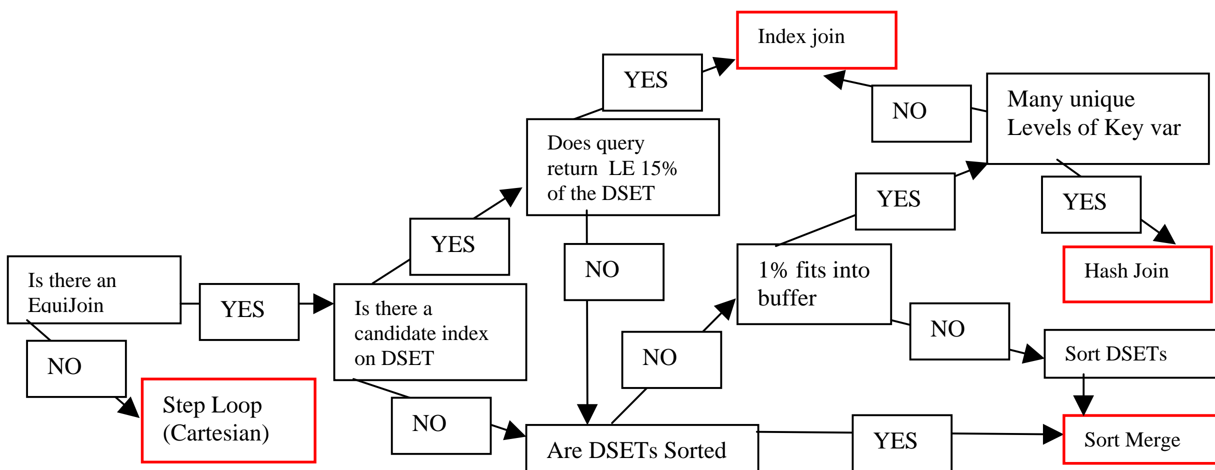
For a one file SQL query, as shown below, the Optimizer must first decide if it should push the where down to the data engine or handle it in SQL. The Optimizer's second decision is between using an index or a top-to-bottom read. The Optimizer has access to metadata on the file. The Optimizer considers both the percent of the file that will be returned and the distribution of the values in the file as it creates a plan to minimize run time.



The Optimizer has the ability to examine the metadata for the table and determine information useful for running the query. The metadata not only tells if there is an index on the variable in the where clause, but allows the Optimizser to determine both 1) what percent of the file will be returned by the where and 2) how the values are distributed through the file. This information lets the Optimizer make intelligent decisions on how to quickly access data.

MERGE LOGIC

The approximate logic for selecting a particular join is shown below (DSET means Data Set or table). Unfortunately this flowchart, like the one above, is a working model, rather than a definitive description of the Optimizer logic. The author's only consolation, is that whatever the current logic is, SAS Inc's commitment to improving it's product means that the current Optimizer will soon be replaced with one with more effective and subtle decision rules.



It is known that a where clause containing variables from two, or more, files can not be passed to the data engine.

OUTPUT FROM _METHOD

Method sends little output to the log. Below is typical output. It is important to note that an "indentation level" indicates the existence of a result set (working table, or temp file). Output in the appendix has been annotated.

The query

```
proc sql _method;
  title Ex1 - show *  select;
  create table ex1 as  select *  from sashelp.class;
```

produces the following _Method output in the log.

```
NOTE: SQL execution methods chosen are:
      Sqxcrta
          sqxsrc( SASHELP.CLASS)
```

A table of abbreviations is required to interpret the _Method output. Note that all abbreviations shown start with SQX. This prefix stands for "**SQ**L E**x**ecution" code. Below, please find the abbreviations I have been able to collect while investigating _Method and a short explanation of the abbreviations. It is likely that more exist.

Name code	Description
SqxCRTA	Create table as select
SqxSLCT	Select
SqxJSL	Step loop join (Cartesian)
SqxJM	Merge Join
SqxINDX	Index Join
SqxHASH	Hash Join
SqxSORT	Sort
SqxSRC	Source rows from table
SqxFIL	Filter rows
SqxSUMG	Summary stats with group by
SqxSUMM	Summary stats with NO group by

OUTPUT FROM _TREE

The Optimizer creates a program, a multi-step program, and the output from tree can go on for pages. Understanding the run plan requires information from both _Method and _Tree. Below is manually annotated (the red numbers in parenthesis) _Tree output from the query above. Since _Tree output is quite complex, and explained in the annotated logs, only a cursory explanation will be given here.

```
Tree as planned.
                               /-SYM-V-(class.Name:1 flag=0001)
                     /-OBJ----|   (5)
                    |   (3)    |--SYM-V-(class.Sex:2 flag=0001)
                    |          |--SYM-V-(class.Age:3 flag=0001)
                    |          |--SYM-V-(class.Height:4 flag=0001)
                    |           \-SYM-V-(class.Weight:5 flag=0001)
          /-SRC----|
         |   (2)     \-TABL[SASHELP].class opt=''
--SSEL---|              (4)
  (1)
```

Processing Sequence is:
1) rightmost level first
2) from bottom to top inside a level
3) at the the top of each level, summarize what will be passed to the level to the left
4) when a level is done, then step to the left one level

The query is processed, in levels, from right to left, and within "levels" from bottom to top. The idea of "passing data and information/instructions to the level to the left" is a useful mental concept. At the top of each level is a summary of the variables being passed to the level to the left. The result set being passed to the left is object (3) and it consists of variables (5).

In the output above,

(4) SQL reads SAShelp.Class with no options processes by SQL
(keep, drop etc are processed by the data set option processor).

(5) SQL reads variables from a SAS dataset (in sashelp) called class. The variables are:
Name: which is variable 1 in the data set class
Sex: which is variable 2 in the data set class
…..
Weight: which is variable 5 in the data set class

(3) is a summary of variables being passed to the level to the left.
(2) indicates that the branches to the right describe a data source
(1) indicates that this is a select type query (SAS developers can see other types)

Below, please find the abbreviations I have been able to collect while investigating _Tree and a short explanation of the abbreviations.   Thanks to people at SAS for help with explanations.

	Abbreviation	This abbreviation can be found in Example Query Number found in the Appendix	Process
	ADIV	15	Divide
	AGGR	7, 8, 11	This indicates an aggregation, but SQL does several types of aggregation. This is associated with an aggregation operation like "select sex, sum(x) as totl group by sex", or "select name, Min(x) as smallest" Sometimes processing the AGGR requires a separate pass through the data set (look for re-merging note in the log as an indicator of a separate pass) and some times it does not.
	AMUL	14	Multiply **– A**rithmetic **Mul**tiplication
	ASC	3, 6A, 6C, 8, 13	Sort in **ASC**ending order

	ASGN	4, 5, 7	Assign. Create a new variable or **As**si**gn** a value to a new variable. If the SQL code is
			
			`"select sum(x) as totl"`
			
			X will be summed and the result **as**si**gn**ed to a variable named "totl". If the programmer names the new variable, the name will be used in output and the variable will be easy to identify/track through the output. If the programmer does not name the variable, it will be numbered and can be tracked via the number.
			
			It is suggested that created variables be named as shown below
			`Select coalesce(l.name, r.name) as Cname , r.age as Rgt_age`

	CEQ	4, 5, 8, 13	This indicates a logical instruction to be passed to the level to the left, where it is executed. CEQ means "check if these 2 leaves are equal" CEQ is the symbol for both numeric and character equality testing.
	DESC	13	The sort, on the level to the left, should be in descending order. This DESC code is "information passed to the left" on the output. The descending sort is performed (files are created and time is spent) in the sorting in the level to the left of the DESC.
	Dlist		List of variables with distinct values that are participating in an aggregation/summation/grouping. See Slist and Tlist. Distincting gets rid of duplicates and Dlist reports on the distincting process. Variables on a dlist have to have duplicates removed before you can apply the aggregation.
	Empty	3, 4, 5, 6B, 6C, 7 , 8	This is a place holder in the output. The Optimizer has the capacity to do additional operations at this point- operations that were not performed.
	FCOA	4, 5,17,18,	This indicates a function, like a SAS function, of type

			coalesce.
	FIL	23	FILter is used to handle the situations that can not be handled by the data engine that is feeding into it. Fil indicates that SQL applied an additional predicate late in the processing. The clearest example of this is Ex 23. The where clause contains a variable that is not in the source data set (age_mo=age*12). It first must be created by SQL and then the filter predicate can be applied by SQL.
	Flags (`class.Sex:2 flag=0001)`		Flags are for developers and are also used internal to SQL. They are beyond the scope of this paper but, as one example, (001) means that the variable is used higher up in the SQL processing.
	JTAG	17, 17A, 19, 19A, 19B, 19C, 19D, 20A, 20B, 20c, 20D, 21A, 21B, 21C, 21D	This is a code that tells what type of join was applied. JDS=1 indicates a left join, JDS=2 Indicates a right join and JDS=3 indicates a full join
	FROM	4, 5	From indicates that data sources are being passed to higher (more leftward) process.
	GRP	8,	Group –is a multi-step process and can take some time to perform
	JOIN	4, 5	This indicates the SQL did a join, but not which type
	LAND	12	A **L**ogical **AND** should be performed. This is usually in a where or a select.
	LITC	Not shown	This indicates the use of a **Lit**eral **C**onstant (character string) as in: where state="PA"
	LITN	13,14 ,	This indicates the use of a **Lit**eral **Number** (ie numeric constant) as in: Where age LT 12
	OBJ	1,2, 3, 4, 5, 6A, 6B, 6C, 7, 8, 13, 14,	This indicates the existence of an object, or result set. Obj indicates a description of columns in a result set that is passed leftward for more processing. Objects come from data sets or lower objects.
	OBJE	4, 5, 7,14, 15,	This indicates the existence of an evaluated object, the result of an assignment of a value to a variable. An OBJE is a variable that is typically added/merged to another object (result set) at the same level. When a programmer codes `Select sex, max(age) as Mage` The value of maximum age will be assigned to Mage and mage will be, for a brief moment before the merge into the result set for that level, an OBJE. See example 4.
	ORDR	3, 6A, 7, 8,	Order By, On the tree, contains ordering information that is passed to the sort to the left. Order is information and not an instruction. It is not, and does not create, a result set. The sort, to the left of the ORDR, creates the result set.
	OTRJ	17, 18, 19	This stands for any of the following joins: Left join, right join, full join. OTRJs are paired with JTAGs and the JTAG indicates the type of join.
	SLST	7, 8,15	Indicates a list of things that are involved in an aggregation/summarization. This is a "Not distinct" List of things that "participate" in the summarization – See Dlist and Tlist
	SORT	3, 7, 8, 13	Sort, at this level, in the order described to the right.

	SRC	1,2, 4, 5, 6A, 6B, , 6C,7 , 8,	Information to the right of this is a data source. Typically an object is being passed to the left.
	SSEL	1,2, 3, 4, 5, 6A, 6B, 6C,7 , 8	SSEL indicates that the query is a Select query. Not all queries are of type select. There are modify queries and drop queries and others.
	SUBP	25	This indicates an input to the subquery. It is a parameter passed to the subquery
	SYM-A	4, 5, 7, 14,	This identifies a variable as an assigned/created variable- one that did not get read from a data set. A SYM-A is the result of Assigning a value to a variable through a calculation or function as in: `Select name ,Age_yr=age*12 as YRS`
	SYM-G	7, 8	A SYM-G is the result of assigning a value to a variable through a **G**rouping calculation or function as in: `Select name ,sex, Max(age) as Mage Group by sex;`
	SYM-V	25	This identifies a variable as a having been read from a data set.
	Sym-v lib.name Flag=0001	1, 2, 3, 4, 5, 6A, 6B, 6C, 7, 8,14,	This shows a combination of abbreviations as they might appear in the log. A variable was read from a source table (SAS data set lib.name). See Flag above
	Table [lib].fname opt=" "	1, 2, 3,4 , 5 , 6A, 6B, 6C, 7, 8	This shows a combination of abbreviations as they might appear in the log. Tables are identified with a two part name. See opt=
	TLST	7, 8,15	Indicates a summarization. A temp List of things that "participate" in the summarization. See Dlist and Slist
	UNIQUE	6A, , 6C,	This is the message to the log when select distinct is coded. Creating unique values is usually implemented by passing the current result set to Proc Sort with a nodup/nodupkey
	opt=" "		SQL allows data set options inside the Query. An example might be `From class(keep=name age)` Some of these options are processed by Base SAS and some are processed by SQL. If an option is processed by SQL, it will show up in the opt=" " note.

CONCLUSION

Due to the page limitations of the WUSS conference, the full paper can not be put on the CD. For the full paper, see the SUGI 30 proceedings. The SUGI article has over 40 pages of annotated output from method and Tree.

The SQL Optimizer is a tremendous help to programmers, allowing them to write very efficient queries with absolutely no thought. The amount of work that the SQL Optimizer does, independently of programmer input and totally behind the scenes, is amazing.

In some cases the performance may be improved by re-coding the query and passing the Optimizer different instructions. These issues will be explored in future papers.

If SQL performance is causing problems, knowing what the Optimizer created for a plan of execution is essential if the programmer want to attempt to improve performance. _method and _tree allow the programmer to see how SQL executed the query and to see the effects of her/his programming changes.

REFERENCES

TS553–SQL Joins –the Long and the Short of it, by Paul Kent – available on the SAS web site
TS320-Inside PROC SQL's Query Optimizer, by Paul Kent – available on the SAS web site

Church (1999), Performance Enhancements to PROC SQL in Version 7 of the SAS® System
Performance Enhancements to PROC SQL in Version 7 of the SAS® System, Proceedings of the Twenty-fourth Annual SAS Users Group International Conference", 24 , paper 51

Kent, Paul (1995) "SQL Joins – The Long and The Short of IT Proceedings of the Twentieth Annual SAS Users Group
International Conference, Cary, NC: SAS Institute Inc., 1995, pp.206-215.

Kent, Paul (1996) "An SQL Tutorial – Some Random Tips",Proceedings of the Twenty-First Annual SAS Users Group International Conference pp. 237-241.

For non-SAS explanations of SQL execution, see the article by Dan Hotka at www.odtug.com

ACKNOWLEDGMENTS

The author wishes to thank the Paul Dorfman, Sigurd Hermansen, Kirk Lafler, and Paul Sherman for their contribution to the SAS community on SQL and for inspiring this paper. Thanks to Paul Sherman for his review and comments.

Thanks for the help from SAS institute, especially help from Paul Kent and Lewis Church.

CONTACT INFORMATION

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

Russell Lavery Email: russ.lavery@verizon.net
or c/o Numeric, LLC
5 Christy Drive
Bldg.2 Suite 107
Chadds Ford, PA, 19317 (610)642-0700

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 trademarks of their respective companies.

The Appendix can be found in the SUGI 30 article of the same name.