

## New SAS® Performance Optimizations to Enhance Your SAS® Client and Solution Access to the Database

Mike Whitcher, SAS Institute, Cary, NC

### ABSTRACT

The SQL procedure has been used for years as the way many SAS clients and solutions query for their data. Examine the new SQL performance optimizations that have been added to this bellwether procedure, optimizations designed to greatly expand query pass-through capability to databases and shorten your SAS client and solution query response times. Also, see the new SQL enhancements for use with SAS data sets. Whether you use SAS® Web Report Studio, SAS® Marketing Automation or other SAS clients or solutions, or still submit your SQL queries in batch, you owe it to yourself to see how you can make them run faster.

### INTRODUCTION

If your data is stored in an external database, you know how costly it can be to pull a large table into SAS to execute your query. You want as much of the SQL query as possible to be executed in the database. The theme for many of the SAS 9.2 PROC SQL optimizations is improved SAS client and solution performance with external databases. The primary way to improve performance to external databases is to have more of the query or the entire query executed in the database. PROC SQL implicit pass-through (IP) technology is used to pass SQL queries to the database (via SAS/ACCESS® products). Because performance with external databases is tied to implicit pass-through (IP), IP has evolved to play a critical role in the success of your SAS client and solution (Church 1999).

This paper presents the major SQL optimizations that have been added to PROC SQL to enhance its performance for SAS 9.2. These optimizations are the result of analyzing SQL queries generated by SAS clients and solutions, and finding new and innovative ways to squeeze out more performance. Many of these queries come from real production jobs, created by SAS customers. As data volumes continue to grow at staggering rates, SAS is committed to improving our SQL technology to meet the needs of today's SAS users.

In addition to enhancing performance to external databases, PROC SQL has also been enhanced to improve the performance of SQL queries that access SAS data sets. It should not be surprising that some types of optimizations will enhance both. Therefore, this paper is divided into three parts:

- optimizations when querying databases
- optimizations when querying SAS data sets
- optimizations that give greater performance for both databases and SAS data sets

By combining some of these optimizations, you can gain even greater performance. But did you know that some optimizations can also help you to write more robust queries? Two specific cases will be presented:

- external database queries that contain date or datetime variables
- queries using the TODAY, DATE, TIME, or DATETIME functions

Examine the latest in SQL optimizations for SAS 9.2 that have been designed to enhance the SAS client and solution experience.

**Note:** For detailed information about the SAS functions, formats, and options used throughout this paper, see the SAS online documentation for *SAS® 9.2 Language Reference Concepts*.

### PROC SQL OPTIMIZATIONS WHEN QUERYING THE DATABASE

This section describes PROC SQL optimizations that enhance query execution to external databases. It is divided into the following main topics. Each topic is described in greater detail:

- using unPUT technology
- using enhanced textualization of aliases, joins, views, and inline selects
- using database row count information for PROC SQL planning through WAG and WIG

## TAKING ADVANTAGE OF UNPUT TECHNOLOGY

When SAS clients and solutions work with external databases, they depend on SQL queries being passed down and executed within those databases. Typically, this is accomplished only if the SQL query is free of syntax specific to SAS that the external databases cannot understand. As mentioned in the introduction, PROC SQL already contains technology that allows the procedure to implicitly pass the SQL query to the database if the procedure is free of such syntax and meets other criteria.

SAS Business Intelligence (BI) software, such as SAS® Web Report Studio and SAS® Enterprise Guide®, generates SQL queries to filter and return formatted data. The formatted data values are used to populate tables and graphs in reports that SAS software presents to users. The PUT function is used in SQL queries to describe how PROC SQL formats the resulting values that are used in WHERE or HAVING clauses.

Although the PUT function is specific to SAS software, it is useful when querying data from databases via SAS ACCESS engines, as well as from SAS data sets. When accessing database tables, the SAS ACCESS engine fetches the data from external databases into the PROC SQL process space where the formatting work for the PUT function is performed. Any SQL query that contains a reference to the PUT function must first fetch the data into SAS in order to perform the query. If table sizes are large, say millions or hundreds of millions of observations, the demands on the system (computers and network) to perform such a query will be correspondingly large. Not only does the increased response time required to fetch the data translate into greater expense for the customer, the additional disk and memory space to store the data also adds to the expense.

As the integration of SAS software with external databases expands and the popularity of using formatted data increases, there is an increased probability that queries will not pass to the database. In light of these performance problems, SAS customers need a new technique to reduce the size and time constraints of performing such queries.

The unPUT technology described in this paper allows programmatic transformation of many PUT functions into an alternate SQL syntax that can be passed to a third-party database. This new technique will reduce the following:

- resource requirements by helping to limit the amount of data that must be fetched
- response time for the query

In summary, unPUT technology solves the performance problems noted by transforming the PUT function, when possible, into a different expression that can be passed to external databases. For the initial implementation of this technology, developers targeted PUT functions in the SQL WHERE and HAVING clauses.

The following examples show targeted queries that are accompanied by the transformed equivalent.

### Query that Includes a User-Defined Format

This first example uses a user-defined format to query for certain size categories.

```
proc format;
  value udfmt 1-3='small' 4-6='medium' 7-9='large';
run;

proc sql;
  select style as SmallStyles from db.clothes
  where put(size, udfmt.) = 'small';
quit;
```

As written, the query cannot be passed to the database for execution because it uses a PUT function that contains a user-defined format. During the PROC SQL planning process, unPUT technology recognizes the expression **put(size, udfmt.) = 'small'**. The unPUT optimization looks up the definition of the udfmt format to find the allowed values for the label 'small' (1-3 in this example) and then constructs a new WHERE clause:

```
(1 <= size and size <= 3)
```

The unPUT optimization parses the new expression and reinserts it back into the in-memory structure (known as the

SQL tree) that is used to process the query. Query processing continues, and with the PUT function removed, the query can now be passed to the database. As a result, it is likely that fewer records are returned, resulting in increased performance and processing of fewer records by PROC SQL. The final query that the unPUT optimization passes to the database looks similar to this example:

```
select style as SmallStyles from db.clothes
  where (1 <= size and size <= 3);
```

### Query with a WHERE Clause that Contains NOT and IN Operators

The unPUT optimization works with many different types of expressions that use PUT functions and complex operators. Here, a WHERE clause contains NOT and IN operators.

```
proc format;
  value udfmt 1-3='small' 4-6='medium' 7-9='large';
run;

proc sql;
  select style from db.clothes where put size, udfmt.)
  not in ('small', 'large');
quit;
```

The unPUT optimization transforms the NOT and IN operators and then generates a simpler expression that can be passed to the databases.

```
NOT( (7 <= size and size <= 9) OR (1 <= size and size <= 3) )
```

The unPUT optimization supports many types of PUT function expressions, including those in the following list:

- equals, put(birthday, date9.) = '01JAN2000';
- not equals, put(birthday, date9.) ^= '01JAN2000';
- variable expressions, put(anniv+30, date9.) = '11Feb2000'
- expressions using inequalities, index(put(color, nudfmt.), "RED") > 0
- IN, NOT IN clauses, put(size, udfmt.) in ('small', 'large')
- LIKE, NOT LIKE clauses, put(size, udfmt.) like "small"
- CONTAINS, NOT CONTAINS clauses, put(type, udfmt.) contains "s"
- BETWEEN, NOT BETWEEN clauses, put(size, udfmt.) between 'small' and 'large'
- STRIP or TRIM(LEFT()) functions, strip(put(x, udfmt.)) in ('small', 'medium')
- UPCASE function, upcase(put(bday, date5.)) = '01JAN'

### Query that Compares Formatted Dates

Comparing formatted dates is a very common operation. This query shows how the unPUT optimization simplifies the comparison of formatted dates. For example, this query helps you examine a population sample of individuals who were born on New Year's Day. By typing the date into a GUI, the query that is generated looks like this:

```
proc sql;
  select name from db.employees where
  (put(birthday, date5.) = '01JAN');
quit;
```

There is no user-defined list of values to substitute for the label '01JAN' in this example. Instead, the unPUT optimization makes use of other functions, like DAY and MONTH, to perform the transformation. The expression, (put(birthday, date5.) = '01JAN'), is changed to the following:

```
(MONTH(birthday)=1 AND DAY(birthday)=1 )
```

The SAS SQL IP technology maps the MONTH and DAY SAS functions to their equivalent database functions, like EXTRACT in Oracle®, in order to perform the proper operation. Again, the unPUT optimization has transformed a query with formatted data into a query that can be passed to the external databases.

**Note:** Each database has its own set of functions. The SAS/ACCESS engines map SAS functions to their equivalent database function. For more information about which functions are supported in this mapping process, see the SAS online documentation for SAS/ACCESS software at <http://support.sas.com/documentation>.

### Query that Uses Nested Functions

Because of the popularity of nested functions, the unPUT optimization also recognizes two examples of such functions, namely UPCASE and STRIP.

If you place a call to the UPCASE function around the PUT function call, you will convert the result from the PUT function to uppercase. This is a common practice when you perform a case independent compare. The unPUT optimization performs the conversion as part of its expression rewrite. Consider this query:

```
proc sql;
  select name from db.employees
         where (upcase(put(birthday, date5.)) = '01JAN');
quit;
```

Just as in the previous example of the query that compares formatted dates, the SQL procedure's unPUT optimization substitutes the WHERE clause for the clause above:

```
proc sql;
  select name from db.employees
         where (MONTH(birthday)=1 AND DAY(birthday)=1)
quit;
```

The query now passes to the database.

### Query that Uses STRIP or TRIM Functions

Comparing data columns with leading and trailing blanks is a common problem in SQL queries. To resolve this difficulty, programmers often use the STRIP function or TRIM(LEFT()) function. The unPUT optimization can transform expressions that contain such functions. Consider this query:

```
proc sql;
  select style as Styles from db.clothes
         where strip(put(size, udfmt.)) in ( 'small', 'medium');
quit;
```

The WHERE clause is rewritten as:

```
(4 <= size and size <= 6)    OR    (1 <= size and size <= 3)
```

As a part of regenerating the query, the unPUT optimization handles all activity that is required to normalize the labels (that is, to remove leading and trailing spaces before performing label comparisons) and also checks for missing labels.

If an expression can never be true, the unPUT optimization returns a zero (0) WHERE clause that further simplifies the processing of the query.

Similar to the preceding examples, unPUT works with user-defined formats and date formats. However, the unPUT optimization also works with SAS date-and-time formats, time formats, the \$ format, and BEST (or BEST12.). UnPUT also supports over 275 SAS formats and different widths for each format.

The unPUT optimization supports all date-and-time formats and width ranges that appear in the following table:

Supported unPUT Formats & Widths				
AFRDFDD 2-10	AFRDFDE 5-9	AFRDFDN 1-32	AFRDFDT 4-40	AFRDFDWN 1-32
AFRDFMN 1-32	AFRDFMY 5-7	AFRDFWDX 3-37	AFRDFWKX 2-38	CATDFDD 2-10
CATDFDE 5-9	CATDFDN 1-32	CATDFDT 7-40	CATDFDWN 1-32	CATDFMN 1-32
CATDFMY 5-32	CATDFWDX 3-40	CATDFWKX 2-40	CRODFDD 2-10	CRODFDE 5-9
CRODFDN 1-32	CRODFDT 7-40	CRODFDWN 1-32	CRODFMN 1-32	CRODFMY 5-32
CRODFWDX 3-40	CRODFWKX 3-40	CSYDFDD 2-10	CSYDFDE 10-14	CSYDFDN 1-32
CSYDFDT 12-40	CSYDFDWN 1-32	CSYDFMN 1-32	CSYDFMY 10-32	CSYDFWDX 8-40
SYDFWKX 2-40	DANDFDD 2-10	DANDFDE 5-9	DANDFDN 1-32	DANDFDT 7-40
DANDFDWN 1-32	DANDFMN 1-32	DANDFMY 5-7	DANDFWDX 3-18	DANDFWKX 2-31
DATE 5-11	DATEAMPM 7-40	DATEIME 7-40	DAY 2-32	DDMMYY 2-10
DDMMYYB 2-10	DDMMYYC 2-10	DDMMYYD 2-10	DDMMYYN 2-8	DDMMYYP 2-10
DDMMYYYS 2-10	DESDFDD 2-10	DESDFDE 5-9	DESDFDN 1-32	DESDFDT 7-40
DESDFDWN 1-32	DESDFMN 1-32	DESDFMY 5-7	DESDFWDX 3-18	DESDFWKX 2-30
DEUDFDD 2-10	DEUDFDE 5-9	DEUDFDN 1-32	DEUDFDT 7-40	DEUDFDWN 1-32
DEUDFMN 1-32	DEUDFMY 5-7	DEUDFWDX 3-18	DEUDFWKX 2-30	DOWNAME 1-32
DTDATE 5-9	DTMONYY 5-7	DTWKDATX 3-37	DTYEAR 2-4	DTYYQC 4-6
ENGDFDD 2-10	ENGDFDE 5-9	ENGDFDN 1-32	ENGDFDT 7-40	ENGDFDWN 1-32
ENGDFMN 1-32	ENGDFMY 5-7	ENGDFWDX 3-32	ENGDFWKX 3-37	ESPDFDD 2-10
ESPDFDE 5-9	ESPDFDN 1-32	ESPDFDT 7-40	ESPDFDWN 1-32	ESPDFMN 1-32
ESPDFMY 5-7	ESPDFWDX 3-24	ESPDFWKX 1-35	FINDFDD 2-10	FINDFDE 8-10
FINDFDN 1-32	FINDFDT 7-40	FINDFDWN 1-32	FINDFMN 1-32	FINDFMY 8
FINDFWDX 3-20	FINDFWKX 2-37	FRADFDD 2-10	FRADFDE 5-9	FRADFDN 1-32
FRADFDT 7-40	FRADFWDN 1-32	FRADFMN 1-32	FRADFMN 1-32	FRADFWDX 3-18
FRADFWDX 3-27	FRSDFDD 2-10	FRSDFDE 5-9	FRSDFDN 1-32	FRSDFDT 7-40
FRSDFDWN 1-32	FRSDFMN 1-32	FRSDFMY 5-7	FRSDFWDX 3-18	FRSDFWKX 3-27
HUNDFDD 2-10	HUNDFDE 12-14	HUNDFDN 1-32	HUNDFDT 12-40	HUNDFDWN 1-32
HUNDFMN 1-32	HUNDFMY 9-32	HUNDFWDX 6-40	HUNDFWKX 3-40	IS8601DA 10
IS8601DN 10	IS8601DT 19-26	IS8601DZ 20-35	IS8601TM 8-15	IS8601TZ 9-20
ITADFDD 2-10	ITADFDE 5-9	ITADFDN 1-32	ITADFDT 7-40	ITADFWDN 1-32
ITADFMN 1-32	ITADFMN 1-32	ITADFWDX 3-17	ITADFWKX 3-28	JULDATE 5-32
JULDAY 3-32	MACDFDD 2-10	MACDFDE 5-9	MACDFDN 1-32	MACDFDT 7-40
MACDFDWN 1-32	MACDFMN 1-32	MACDFMY 5-32	MACDFWDX 3-40	MACDFWKX 3-40
MDYAMPM 8-16	MMDDYY 2-10	MMDDYYB 2-10	MMDDYYC 2-10	MMDDYYD 2-10
MMDDYYN 2-8	MMDDYYP 2-10	MMDDYYS 2-10	MMYY 5-32	MMYYC 5-32
MMYYD 5-32	MMYYN 4-32	MMYYP 5-32	MMYYYS 5-32	MONNAME 1-32
MONTH 1-32	MONYY 5-7	NLDDFDD 2-10	NLDDFDE 5-9	NLDDFDN 1-32
NLDDFDT 7-40	NLDDFDWN 1-32	NLDDFMN 1-32	NLDDFMY 5-7	NLDDFWDX 3-37
NLDDFWKX 2-38	NORDFDD 2-10	NORDFDE 5-9	NORDFDN 1-32	NORDFDT 7-40
NORFDWN 1-32	NORDFMN 1-32	NORDFMY 5-7	NORDFWDX 3-17	NORDFWKX 3-26
POLDFDD 2-10	POLDFDE 5-9	POLDFDN 1-32	POLDFDT 7-40	POLDFDWN 1-32
POLDFMN 1-32	POLDFMY 5-32	POLDFWDX 3-40	POLDFWKX 2-40	PTGDFDD 2-10
PTGDFDE 5-9	PTGDFDN 1-32	PTGDFDT 7-40	PTGDFDWN 1-32	PTGDFMN 1-32
PTGDFMY 5-7	PTGDFWDX 3-37	PTGDFWKX 3-38	QTR 1-32	QTRR 3-32
RSTDODB 6-32	RSTDOMN 1-32	RSTDOMY 12-32	SLODFDD 2-10	SLODFDE 5-9
SLODFDN 1-32	SLODFDT 7-40	SLODFDWN 1-32	SLODFMN 1-32	SLODFMY 5-32
SLODFWDX 3-40	SLODFWKX 3-40	SVEDFDD 2-10	SVEDFDE 5-9	SVEDFDN 1-32
SVEDFDT 7-40	SVEDFDWN 1-32	SVEDFMN 1-32	SVEDFMY 5-7	SVEDFWDX 3-17
SVEDFWKX 3-26	TIME 2-20	TIMEAMPM 2-20	TOD 2-20	TWMDY 15-35
WEEKDATE 3-37	WEEKDATX 3-37	WEEKDAY 1-32	WORDDATE 3-32	WORDDATX 3-32
XYMMDD 6-12	YEAR 2-32	YYMM 5-32	YYMMC 5-32	YYMMD 5-32
YYMMDD 2-10	YYMMDDB 2-10	YYMMDDC 2-10	YYMMDDD 2-10	YYMMDDN 2-8
YYMMDDP 2-10	YYMMDDS 2-10	YYMMN 4-32	YYMMP 5-32	YYMMS 5-32
YYMON 5-32	YYQ 4-32	YYQC 4-32	YYQD 4-32	YYQN 3-32
YYQP 4-32	YYQR 6-32	YYQRC 6-32	YYQRD 6-32	YYQRN 5-32
YYQRP 6-32	YYQRS 6-32	YYQS 4-32	YYQZ 4-6	

### Query that Uses \$ and BEST Formats

To reiterate a previous point, unPUT technology supports the \$ and BEST formats. Because they are the default formats for SAS Web Report Studio, put(s, \$.) or put(x, best.) occur often in the queries that this SAS client

generates. The put(s, \$.) expression is also an expression that the unPUT optimization can optimize in the select list. For example:

```
proc sql;
  select put(s, $.) as DIR_1 from db.t1
  where put(size, best.) = '11';
quit;
```

The unPUT optimization transforms this query as follows:

```
select s as DIR_1 from db.t1
  where x = 11;
```

### Query that Includes User-Defined Formats and Inequalities

This final query illustrates a more complex transformation that involves PROC FORMAT style formats and inequalities. The following example comes from a field request to pass a query that involves inequalities to the database:

```
proc format;
  value nudfmt 0='RED'
  1='REDHEAD'
  2='NOTRED'
  3='GREEN'
  other='BLACK';
run;

proc sql;
  select * from db.data_i
  where (index(put(color, nudfmt.), "RED") > 0);
quit;
```

The unPUT optimization generates the following query transformation:

```
select * from db.data_i
  where color IN(0,1,2);
```

This example highlights the necessity of using the FORMAT procedure's OTHER= option. In order to render expressions that are mathematically sound, the unPUT optimization must know the full range of possible outcomes for a user-defined format. If the OTHER= option is not present, a call to the PUT function for a column value that does not match results in a character string of that column value, whether the input column is numeric or character.

In the following example, if color is 5, and there is no OTHER= for nudfmt, then the PUT function below returns the character string '5'. That is, column value 5 does not match one of the defined values in the nudfmt format (1, 2, or 3), so the number is converted to its character-string equivalent.

```
put(color, nudfmt.)
```

Without the OTHER= option, it is impossible to use the unPUT optimization for a PUT expression that uses PROC FORMAT style formats. By adding the OTHER= option to the PROC FORMAT definition, the format now has a default label to which it can refer for non-matching values.

Experience with internal testing suggests that it is atypical for SAS programmers to include the OTHER= option when defining PROC FORMAT style formats. For customers to use this technology, it might be necessary to retrofit the OTHER= option to their PROC FORMAT style definitions.

There is a new message that appears in the SAS log when the unPUT optimization fails because the PROC FORMAT- style format does not include the OTHER= option.

```
NOTE: Optimization for the PUT function was skipped because
the referenced format, NUDFMT, does not have an OTHER= range
defined.
```

Customers who encounter this new message and want to perform unPUT optimization should add the OTHER= option to their FORMAT procedure definition. For examples of using the OTHER= option, see "The FORMAT Procedure" chapter in *Base SAS 9.2 Procedures Guide* in the SAS online documentation.

### Query Problems UnPUT Cannot Solve

The unPUT technology has been retrofitted and made available to SAS 9.1.3 customers, starting with hot fixes E9BB70 and E9BB72. Here is one example where unPUT raised some customer questions. In this example, SAS Web Report Studio is used to build a query to populate a table. The query has a numeric category variable and also a SAS Web Report Studio filter that selects a range of values for the category variable. The PROC SQL code for this query looks like the following:

```
proc sql;
  select distinct PUT(table0.ranges, BEST.) as DIR_1,
                 SUM(table0.counts)      as DIR_2
  from db.sample table0
  where strip(PUT(table0.ranges, BEST.)) BETWEEN '1' and '5'
  group by 1;
quit;
```

There are a couple problems with the query. First, the filter is likely not what the user intended, because it is performing a lexicographical string comparison on what intuitively appears to be number, and not numeric comparisons. If the user's data is anything outside the range of 0 - 9, they will get unexpected results. For example, assume this is the data for the sample table.

```
data db.sample;
  input ranges counts;
  cards;
1 1
2 1
10 1
;;;
run;
```

The SQL query above correctly returns all three values of 1, 2, and 10, even without using unPUT, when at first glance only 1 and 2 are expected to be returned. The formatted string '10' is lexicographically between '1' and '5'. The SAS Web Report Studio user probably wanted to filter unformatted data and return formatted data. This is accomplished by changing the filter to be something like the following that does pass through to the database.

```
WHERE table0.range BETWEEN 1 and 5
```

This same problem can occur when using the CONTAINS and LIKE operators. It can also occur with inequalities. In general, be careful when using formatted data with range set operators like BETWEEN.

A second issue is that PUT functions in the SELECT list will not go down to the database. UnPUT technology is not designed to address PUT functions in the SELECT list. The unPUT optimization only works on PUT functions in the WHERE and HAVING clauses.

**Note:** PUT(s, \$.), described in a previous section, is the only exception to this rule. SAS in-database technology, currently in research, is the next opportunity to address this portion of the formatting issue. When executed, the rows for this query are brought back into SAS where the PUT function and the GROUP BY clause are applied.

## If UnPUT Needs to be Turned Off

Every effort has been made to make the unPUT technology robust and efficient, but if necessary it can be turned off by using the following SAS option.

```
options SQLREDUCEPUT=NONE; /* SAS 9.2, Turn the unPUT optimization off */
```

The unPUT optimization can also be turned off for SAS 9.1.3 by using the following macro variable.

```
%let SYS_SQLREDUCEPUT=NONE; /* SAS 9.1.3, Turn the unPUT optimization off */
```

It can also be controlled from the PROC SQL statement. For example:

```
proc sql reduceput=none;  
...  
quit;
```

See the SAS 9.2 documentation for a complete description of this new option.

## Summary

Use of the unPUT optimization is intended to ensure that external databases perform a greater portion of the subsetting for queries that SAS clients and solutions generate. For example, running a query with the SAS Web Report Studio query cache turned on should yield a configuration whereby the initial subsetting query is sent to the database, and follow-up queries, for individual tables and graphs within a report, can execute queries against cached data. These circumstances result in better response times for the user.

## ENHANCED TEXTUALIZATION OF ALIASES, JOINS, VIEWS, AND INLINE SELECTS

SAS Research and Development is constantly refining PROC SQL to improve its effectiveness at generating SQL code that can be passed to external databases. There are two primary components in PROC SQL responsible for this process:

- data flow analysis, or data flow resolution, or DFR
- textualization

As PROC SQL parses a query, it stores the various parts of the query into an in-memory tree. Data flow analysis sweeps over the tree and attempts to model the tree into an efficient form for execution of the query. When DFR was first written by SAS co-founder, John Sall, back in the early 1980s, the SAS/ACCESS engine architecture we have today had not yet been conceived, and the ANSI SQL standard for aliasing had not been formalized. As a result, the main goal of DFR was to push much of the tree to lower leaf levels, and teach the Base SAS engine to handle it. This strategy worked well for many years.

Today, current SAS/ACCESS engine functionality and the ANSI SQL standard (that external databases support) mean DFR must perform more efficiently. There is an ANSI SQL 2 INNER JOIN syntax that is common and popular among query writers. Table and column aliasing are also prevalent. PROC SQL IP and all external databases support this syntax (American National Standards Institute 2003). Changes have been made to DFR to account for these factors. These changes produce an SQL tree that provides the highest probability for IP to occur that in turn generates a query that successfully passes to the database.

Textualization is the process of taking an SQL tree and generating SQL text (for example, a SELECT statement) that can be sent to the database. SAS/ACCESS supports many different external databases and often many releases within each database. SQL queries can contain a mixture of semantics specific to SAS that can be processed only in SAS, so the process of re-generating SQL text that will pass down to a specific database can be difficult. The textualizer has been enhanced to adjust to the changes in DFR. It has also been enhanced to handle increasingly more difficult types of join combinations, table and column aliases, as well as inline expressions and inline views. By setting the SAS option, DBIDIRECTEXEC, the SQL textualizer will also enable pass-through for SQL CREATE TABLE and DELETE statements.

```
options dbidirectexec;
```



These combined efforts result in greater stability and a 40% increase in the number of queries from our own database test suites passed through (even partially). While SAS continues to improve in this area, today's performance benefits accessing database tables are immediate.

## **WAG AND WIG, USING DATABASE ROW COUNT INFORMATION FOR PROC SQL PLANNING**

Since its first release, PROC SQL has contained a planner (optimizer) phase. The role of the SQL planner is to analyze a query to determine the best methods (steps) of query execution. It is well known that certain types of algorithms perform better for certain data volumes and data table characteristics. Most external databases have a way for database administrators to pre-compute (or compute on the fly) information about database tables. Often this is called database statistics. The most common database statistic is row count information. Many clients and solutions need to have a priori knowledge of how many rows are in a table, so there are two good reasons to compute and keep current the database information on your tables, especially row count information.

Before SAS 9.2, PROC SQL did not use database row count information in its determination of a join method. The information was not available from the SAS/ACCESS engines, so PROC SQL would make what it hoped was a wildly accurate guess (WAG). This WAG had been *tuned* to try and fit the types of data models typically used by SAS customers. Of course, when it comes to guessing, sometimes you are right and sometimes your WAG becomes a wildly inaccurate guess (WIG). A WIG can cause PROC SQL to make planner choices that perform poorly. Furthermore, a WIG can be propagated to multiple planner stages of a query producing an even greater negative compounding effect. For example, in addition to making a poor choice of algorithm, PROC SQL might under-estimate buffer sizes that lead to inefficient movement of data. If you've ever used the SAS BUFNO option in conjunction with PROC SQL, you might have experienced this problem.

The SAS/ACCESS engines have been enhanced, along with PROC SQL, to obtain and use row count information on database tables. Two uses of row count information by PROC SQL are deciding which method to use to join two tables, and the calculation of buffer sizes for intermediate stages. Having your database administrators keep the database row information for your data model up-to-date is an excellent way to take advantage of these enhancements. Keeping database row count information current gives the PROC SQL planner information it needs to make sure to use a WAG instead of a WIG.

### **Summary**

Using unPUT technology allows SAS client and solutions to perform more of the query subsetting for formatted data on the database.

Improved efficiency of data flow resolution and implicit pass-through mean more of your query is pushed to the database.

Keeping database row count information up-to-date will allow PROC SQL to make optimal planning decisions.

## **PROC SQL OPTIMIZATIONS WHEN QUERYING SAS DATA SETS**

SAS data sets continue to be a popular way for many SAS users to hold their data. The goal then is to continue to improve the performance of queries to SAS data sets. Read the list of enhancements below and you will see that some are useful when PROC SQL is processing database tables. So why are they in this list?, you might ask. The reason is because that was not the primary motivation for making the enhancement. Remember, the goal for IP is to have as much of the query passed through to the database as is possible. If there is language syntax specific to SAS that is preventing IP from happening, then yes, the techniques below would speed the processing done in SAS. But it is their use with SAS data sets that give them the most value.

Let us look at the following PROC SQL optimizations for SAS data sets.

- improved processing for SELECT DISCINT
- improved processing for SELECT COUNT(DISTINCT)
- improved hash join performance
- improved processing by pushing records verses pulling them

## IMPROVED PROCESSING FOR SELECT DISTINCT

Finding the unique items for category variables in a table is a common operation for many clients and solutions. In SQL terms this is known as *distinct processing*. For years, SAS programmers have built indexes on key variables in their SAS data sets. Doing so can often help their query processing performance by avoiding a full scan through the data. PROC SQL has been enhanced to use SAS data set indexes when processing SELECT DISTINCT queries. In addition to simple column references, SELECT DISTINCT queries containing the following types of clauses and phrases are now supported.

- WHERE clause expressions
- expressions in the select list
- aliased columns
- aliased expressions

Supported expressions include those containing constants, index variables, function calls, case expressions, and so on. Although they appear only once in the index, columns can be used more than once in the select list, and can be aliased. PROC SQL will pick out and use the variables inside expressions and aliases, and project the data from the source table to the select list data buffers as necessary. Once in place, calculations for any expressions are performed. This includes WHERE clause evaluation, if a WHERE clause is present. In this way, processing of records is similar to processing of records from a source table, but the data is being pulled from the table's index instead. Compared to most types of full table scans, it is very fast, especially when used with a WHERE clause, and can make all the difference in your Web application having acceptable user response times.

For PROC SQL to consider using an index, the index must contain all the variables being referenced in the query, and all the variables in the index must also be used in the query. Partial usage is not supported, but order of occurrence within the index versus the query is not important. Multiple references of the same column within the select distinct are also acceptable.

Up-front knowledge of your client or solution query patterns is important for constructing the right indexes. It takes time to build an index, especially on a large table, or tables with higher numbers of distinct values, so some analysis of what columns are being used in your select distinct queries will help you maximize the return on your index/query investment.

Finally, the index must also have been built using the UNIQUE keyword (that is, CREATE UNIQUE INDEX <idxname> ON ...). That is, it must be a unique index. Without this keyword, your index can contain duplicates that make it unusable for SELECT DISTINCT processing.

Depending on the number of distinct values in your data, when compared to a full table scan, this optimization can yield stunning performance gains of +1000%. The first time this optimization was tested, a SAS data set containing 30 million rows was created. The SAS data set had a single column with 10,000 distinct values. When executing a SELECT DISTINCT query without the index, it took two minutes to scan the table. Then a unique index was built on the column and the same SELECT DISTINCT query was executed again. It completed in under a second; so fast I thought there had been a mistake. There was no mistake. By using the index, a much smaller amount of data is read, and the query completes in a fraction of the time.

## IMPROVED PROCESSING FOR SELECT COUNT(DISTINCT)

Using an index is nice, but sometimes it is impractical to build an index on a particular variable, it is not cost effective, or PROC SQL cannot use the index. Without the availability of an index, you will have to perform a full table scan, and if your query contains a COUNT(DISTINCT) phrase, PROC SQL will perform the distincting by using a temporary indexing subsystem that writes the distinct values to local disk storage. Here is the sample query:

```
proc SQL;
  select count(distinct personid) from fact_tbl;
quit;
```

The execution step to process the count distinct is simple. With the current value for personid from the table, perform a look-up in the temporary index. If the value is not found, add it to the index, and increment the count for personid. Otherwise, we have a duplicate entry (that does not contribute to the count). The problem is this simple algorithm becomes horribly inefficient as the number of unique values in the table grows. For example, if the column has a

*small* number of distinct values, why write them to disk (even if it is an index file)? It would be better to keep these values in memory until forced to go to disk. With today's larger memory systems, why not allow users to specify how much memory to dedicate to this operation? Also, the balancing of the nodes in the temporary index begins to churn as the size of the index increases. Because this index is temporary, you can adopt a less costly insertion policy.

All of these improvements are accomplished for SAS 9.2. PROC SQL uses the same in-memory index subsystem that the SAS DATA step uses for its hash objects. This means count distinct processing for columns whose list of distinct values can fit into memory will be fast. The SAS startup option, MEMSIZE, is now referenced to determine how large the in-memory index is allowed to become. The minimum number of distinct values per in-memory index is 1024. If counting multiple columns in the same query, then PROC SQL performs balancing of memory resources.

If available MEMSIZE memory becomes constrained, within about 50MB left, new values are written to temporary indexes on disk (while maintaining as much of the in-memory index system as possible). For really large counts of distinct values, the temporary index can even become sluggish. So instead of continuing to increase the insertion cost for every value added, a second temporary index is opened, and new values are placed into it. This cycle is repeated, until the full table scan is completed. If at any time, PROC SQL notices that available MEMSIZE memory has diminished to 5MB or less, it will begin writing out the in-memory indexes to disk, until more memory is again available. In this way, PROC SQL is expected to hold up under client/server load scenarios.

So what is the maximum number of records that get written to a temporary index before another one is opened? By default, it is 2,000,000. You can use the SYS\_SQL\_MAXOBS\_XCI macro variable to control this threshold.

```
%let SYS_SQL_MAXOBS_XCI=1000000; /* change the default to 1 million obs per
temporary index file */
```

The minimum value is one million. By changing the value you vary the performance of each temporary index used, at the expense of having more (or less) open temporary files. With these changes, PROC SQL is able to complete even larger count distinct queries in less time.

## IMPROVED HASH JOIN PERFORMANCE

Hash join is one of those algorithms that can perform well if:

- One of the tables is small enough to fit (mostly) into memory.
- A good hash algorithm that maps well to your buffers is available.
- A good low cost strategy to handle duplicates is available.
- Data movement is minimized.
- Lookup into the hash table is cheap.

**Note:** For more background information about hash join and PROC SQL methods in general, see "The SQL Optimizer Project: \_Method and \_Tree in SAS 9.1" (Lavery 2005).

These were some of the issues that were important to good hashing dating back to 1986 when PROC SQL's hash join algorithm was first implemented, and they still apply today. The difference is today's data volumes have grown considerably. Fortunately, hashing techniques have also continued to improve.

To improve PROC SQL's hash join algorithm, we once again relied on the in-memory subsystem that the DATA step uses for hash objects to hash the key for each row. By using the index, the entire data buffer can be used, instead of the average 71% usage in the old algorithm. The in-memory index also has resolution of duplicates built-in, so by hashing the key and the record offset into the buffer, you completely eliminate any data movement that occurs by maintaining a least recently used (LRU) list.

Hash join has also been enhanced to use more available memory. Hash block sizes are chosen based on the SAS startup option, MEMSIZE. Block allocations are allowed to consume up to 75% of available MEMSIZE memory. Twenty-five percent of memory is held in reserve to ensure enough for the rest of query processing. If this limit is exceeded, the SQL planner will switch to an alternate join method, such as merge join.

With all the upgrades in place, the new hash join is 10-365% faster than the old algorithm. Smaller hashed tables yield smaller performance gains. As long as the table fits into memory, larger hashed tables take greater advantage of the performance savings and run quicker. Best of all, SAS users do not have to change their SQL queries to achieve the results.

## **PUSHING RECORDS VERSUS PULLING RECORDS**

The SAS threaded sort routine is a powerful tool that reads from an input file, sort the records, and writes them to an output file. The records are pushed into the sort buffers, where they can be quickly sorted and written back out. This routine is shared between PROC SORT and PROC SQL when GROUP BY and ORDER BY statements are used. PROC SQL's staged execution pipeline is built around a get-next-record model. Starting with the beginning execution stage, each stage takes its turn asking the stage below it for the next record. Records percolate through the execution pipeline, are processed, and sent along as the query is processed.

In the early days, there was difficulty staging the push record sort step to pull record steps. In cases when the pull step was more than just a file reference (for example, the result of a join step), the records were pulled and written to a temporary file. The temporary file then became the input file to the sort step. The sort step also pushes records out. There were times when a temporary file was needed to capture the sorted records so they could be fed into the next execution stage. Not only was this extra work awkward, it was very time consuming.

PROC SQL has been enhanced to eliminate many instances when temporary files are being used to sort records. The create table, select, and group by stages have been fitted with additional methods that allow them to operate in either push record or pull record mode. When sorting is not needed, these stages continue to work as usual. When an ORDER BY stage is present these three stages can switch to push record mode to increase performance. The performance gains depend on the layout and size of your data, but it can be 200% or more.

### **Summary**

Creating indexes on SAS data sets can greatly enhance the processing of select distinct queries in PROC SQL.

The DATA step hash object is being used in PROC SQL to enhance performance for hash join and count distinct processing.

Enhancements to the execution stages eliminated the need for temporary files for some sort steps.

## **COMMON PROC SQL OPTIMIZATIONS FOR BOTH DATABASES AND SAS DATA SETS**

The PROC SQL optimizations in this section highlight an event in software development when changes in one area have positive impact on the entire system. Whether you query against external database tables or SAS data sets, the optimizations in this area will enhance both. Refer to the following list.

- convert TODAY, DATE, TIME, and DATETIME function to constants
- convert function calls with literal arguments to constants
- constant folding (PROC SQL and SAS WHERE clauses)

### **CONVERT TODAY, DATE, TIME, AND DATETIME FUNCTIONS TO CONSTANTS**

Date and time-based reporting is common among business intelligence operations. Through this reporting, clients can answer questions such as:

- How do third-quarter sales from this year compare with the same quarter from last year?
- What are YTD unit sales?
- When a Web user clicks on an advertisement, how long does it take for the user to make a decision to purchase a product?

To answer all these questions, you must first determine the current date or time. Consequently, the TODAY, DATE, TIME, and DATETIME functions form the basis for determining the answers to such questions. A PROC SQL optimization has been added to allow these critical functions to pass through to the database.

The optimization is to convert the DATE, TIME, and DATETIME functions into their equivalent date (or time) constants. Once expressed as constants, they pass easily to the database. This allows SAS clients and applications (such as SAS Web Report Studio and SAS® Marketing Automation) to pass through more types of date and time-based subsetting queries.

This optimization adds robustness to queries. When we were first testing the changes for this optimization, we realized that using constants has two added benefits.

- PROC SQL can resolve queries that initiate at or near a date or time boundary, with execution continuing across the boundary during the query planning stage.
- Because the ANSI SQL standard requires that the datetime value remain fixed, PROC SQL needs to compute the value only once for the entire result set.

The requirement to resolve the date/time value only one time reduces the overhead required to re-compute the SAS date/time function result for every row. Results are more consistent when date/time calculations appear in query expressions.

A common operation is to add a value to the result of a SAS date/time function. When you add such values to a database datetime value, unexpected results might occur. For example, an attempt to add a number of days to the current date might cause you to add seconds instead of days.

In this optimization, PROC SQL converts date/time functions and marks them as date, time, or datetime constants. As a result, there is a far higher probability that the SAS/ACCESS engines (as well as the database) will compute the expression as intended.

#### **CONVERT FUNCTION CALLS WITH LITERAL ARGUMENTS TO CONSTANTS**

Most SAS functions are *deterministic*. In other words, they return the same result value, given the same input value. Hence, if the function's arguments are constants, the result should be a constant as well. With this optimization, PROC SQL will pre-compute such functions, and use the constant during execution.

There are a couple of immediate benefits. First, if the query is executed using a SAS engine (for example, a Base SAS engine), the function is called once when the query is being planned, and the result is used throughout execution. The reduction of a function call for every record processed can be significant when processing a large table.

If your query is being sent to the database you have the added benefit that removing function calls can enhance implicit pass-through. Every database has some set of functions that it supports, but SAS has a very rich function library. Many SAS functions will not pass to any database, so if the function can be reduced to a constant, it increases its chances of being passed through.

Not every function can be supported. Obvious exceptions are those functions that are non-deterministic. The RANUNI function, which returns a random number from one call to the next, is an example. Non-deterministic functions, like RANUNI, are not supported by this enhancement.

#### **CONSTANT FOLDING (PROC SQL AND SAS WHERE CLAUSE)**

If you have the expression  $1 + 2$  in a query, why compute  $1 + 2$  for each row processed in the table? Why not compute  $1 + 2$  and use the value thereafter? Even if the expression is being sent down to the database, why not just send 3? That was the basis for this optimization, and most compilers these days have the ability to evaluate literal expressions and fold the result back into the query. It is called *constant folding*, and PROC SQL is now able to perform this optimization too.

The real benefit of this optimization comes when it is combined with the function call optimizations of the prior section. Consider an example. From the SAS documentation for the INTNX function, calendar calculations, consider the following expression if it were in a WHERE clause (SAS/ETS User's Guide, 2003). The following

expression is a formula for computing the number of weekdays between today and the second Friday of the following month (ending dates included).

```
intck( 'weekday', today() - 1,  
      intnx( 'week.6', intnx( 'month', today(), 1 ) - 1, 2 ) + 1 );
```

For more information about the INTCK and INTNX function calendar calculations, see the SAS 9.1.3 online documentation for *SAS/ETS® User's Guide*.

Business Intelligence queries frequently subset data based on a range of date values, and the INTCK and INTNX are powerful functions for computing date ranges. Unfortunately, expressions like this would not pass down to a database. The TODAY, INTCK, and INTNX functions are specific to SAS. Even when using the Base SAS engine, we do not need to compute this constant expression for each row. By combining our SQL optimizations we are able to apply and then reapply each of the optimizations as often as needed to reduce the expression to a single numeric constant. For example, the answer to the expression when TODAY was 16Feb2008 is 20.

A constant expression in any portion of the SQL query can be optimized with constant folding. When used in a WHERE clause there is an added bonus. These optimizations have been added to the SAS WHERE processing subsystem shared by all SAS procedures and the DATA step. This allows function reduction and constant folding on any WHERE clause. Whether using a SAS engine or a database, you will enjoy the benefits of SAS 9.2 SQL optimizations.

## CONCLUSION

SQL query response times are critical to users of SAS clients and solutions. This paper introduced new SAS 9.2 PROC SQL performance optimizations designed to reduce query response times by allowing more of the query to be passed through to an external database. Database optimizations include the new unPUT technology that can rewrite PUT function expressions into alternate, yet equivalent, expressions that can be sent to the database. It shows how some expressions can be pre-evaluated into constants, reducing query complexity. Couple these transformations with improved SQL data flow resolution and query textualization, and the changes present a major advancement in SAS PROC SQL Implicit pass-through technology.

Many of these optimizations benefit queries using SAS data sets. In addition, improvements for SELECT DISTINCT and COUNT DISTINCT give better response times when SAS data sets are your data store. All these optimizations started with feedback from SAS users and a need to process more data in a shorter amount of time. What grew out of these relationships was a spirit innovation to solve today's business problems.

## REFERENCES

American National Standards Institute. 2003. WG3:HBA-003 = H2-2003-305 = 5WD-02-Foundation-2003-09, WD 9075-2 (SQL/Foundation). New York: American National Standards Institute.

Church, Lewis. 1999. "Performance Enhancements to PROC SQL in Version 7 of the SAS® System." *Proceedings of the Twenty-Fourth Annual SAS Users Group International Conference*. Cary, NC: SAS Institute Inc. Available at <http://www2.sas.com/proceedings/sugi24/Advttutor/p51-24.pdf>.

Lavery, Russ. 2005. "The SQL Optimizer Project: \_Method and \_Tree in SAS® 9.1." *Proceedings of the Thirtieth Annual SAS® Users Group International Conference*. Cary, NC: SAS Institute Inc. Available at <http://www2.sas.com/proceedings/sugi30/101-30.pdf>.

SAS Institute Inc. 2008. "The FORMAT Procedure." *Base SAS® 9.2 Procedures Guide*. Cary, NC: SAS Institute Inc. Available at <http://support.sas.com/documentation/onlinedoc/base/>.

SAS Institute Inc. 2008. *SAS® 9.2 Language Reference: Concepts*. Cary, NC: SAS Institute Inc. Available at <http://support.sas.com/documentation/onlinedoc/base/>.

SAS Institute Inc. 2003. *SAS/ETS® 9.1.3 User's Guide*. Cary, NC: SAS Institute Inc. Available at <http://support.sas.com/documentation>.

## **ACKNOWLEDGMENTS**

Rick Langston and Howard Plemmons for their work on unPUT technology, Lewis Church extended PROC SQL to support database row count information, Gordon Keener for his help to add constant folding to PROC SQL and SAS where processing, Mauro Cazzari for his work in the area of SQL implicit pass-through to external databases.

The author would like to express his appreciation to Jason Secosky, Margaret Crevar, and Lewis Church for their help in the review of this paper.

## **CONTACT INFORMATION**

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

Mike Whitcher  
SAS Institute Inc.  
SAS Campus Dr.  
Cary, NC 27513  
Work Phone: (919) 531-7936  
Fax: (919) 677-4444  
E-mail: [Mike.Whitcher@sas.com](mailto:Mike.Whitcher@sas.com)  
Web: [www.sas.com](http://www.sas.com)

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.