MWSUG 2017 - Paper BB047

Extraction and Use of Text Strings With SAS® When Source Exceeds the 32K String Length Limit

John Schmitz, Luminare Data LLC, Omaha, NE

ABSTRACT

Database Systems support text fields that can be much larger than what are supported by SAS® and SAS/ACCESS® software. These fields may contain notes, unparsed and unformatted text, or XML data. This paper offers a solution for lengthy text data. Using SQL explicit pass-through, minimal native SQL coding, and some SAS macro logic, SAS developers can easily extract these strings as a set of substring elements that SAS can process and store without data loss.

INTRODUCTION

Strings longer than 32K (32767) characters require additional steps to fully extract into a SAS readable format. Although these strings are commonly supported in most database management systems (DBMS), they are not supported in SAS DATA steps or in the ODBC or OLEDB engines that facilitate the transfer of data between the DBMS and SAS. This paper demonstrates a reliable method to extract and leverage data from these longer strings using base SAS logic and your SAS/ACCESS engine capabilities. The process uses SQL explicit pass-through, native SQL code, and SAS macro language to provide an efficient process to extract data from these longer text fields.

This paper assumes the reader is familiar with general SAS coding including the SQL procedure and basic macros, but may not be comfortable with the SQL options or macro coding used in this effort. The paper begins with a basic overview of core components used in the process before presenting the complete process used for string extraction. It continues with a more advanced process for establishing natural breakpoints within the extracted text fields. The breakpoint logic is shown for XML data, but most readers should be able to modify the approach for other data types. The code samples shown use the OLEDB Access engine, but the process works for most SAS/ACCESS engines.

EXPLICIT PASS-THROUGH OVERVIEW

The first core building block to this process requires explicit pass-through logic. The SQL procedure operates in three different pass-through modes:

- Implicit: SAS translates and sends the requested query to the DBMS for processing,
- Explicit: Developer inserts specific logic to directly pass the query to the DBMS,
- NoIPassThru: Developer invokes the NOIPASSTHRU option, thus forcing SAS execution of the query rather than allowing implicit pass-though.

The string extraction process described here requires explicit pass-through to directly access string sections beyond the 32K boundary.

Explicit pass-through logic requires 3 additional SAS statements and 1 additional component (SAS Institute Inc., 2016, pg. 705):

- The CONNECT statement establishes a connection to the DBMS.
- The EXECUTE statement sends dynamic, non-query DBMS-specific SQL statements to the DBMS.
- The CONNECTION TO component in the FROM clause of a PROC SQL SELECT statement retrieves data directly from a DBMS.
- The DISCONNECT statement terminates the connection to the DBMS.

Using these commands, the explicit query structure follows as:

```
proc sql;
    connect to <engine> as <name>
          (DBMAX_TEXT=30000 <options>);
    create table <outtable> as
    select *
    from connection to <name>
          (<sql query>);
    disconnect from <name>;
quit;
```

where

<engine> is the access engine to use for the connection, (ODBC, OLEDB, ORACLE, etc.),

<name> is the alias name you provide for the connection,

<options> provide connection parameters specific to the DBMS and access engine used,

<outtable> is the output SAS table for the final results (if desired),

<sql query> is the query logic that is passed as written to the DBMS and is written in the native syntax to that system.

The CONNECT TO statement supports numerous options that may be required to connect to the database. These options generally follow the same conventions that are used in the libname statement to establish the SAS/ACCESS connection to the database.

Within this structure, the CONNECT and DISCONNECT statements provide the logic to open and close the connection between SAS and the DBMS. The SELECT statement, excluding the logic within the CONNECTION TO component, remains the same as any standard PROC SQL code. But, it is the CONNECTION TO component that provides the added capabilities needed to extract data from these longer strings. The SQL query will be addressed in the next section.

In many cases, the SAS/ACCESS engines will default to a maximum string length of 1024 (1k) length. Although the process we discuss can work with a 1k limit on substring length, this can result in a very large number of required substring fields. This default can be changed. The option DBMAX_TEXT is available for many of the SAS/ACCESS engines and can be included as an option on the connect statement, as shown above, to define a longer character input string.

LEVERAGING NATIVE SQL CODE

Once explicit pass-through is initiated, we can use the native SQL syntax to access sections of strings beyond the first 32K characters. Coders who are more comfortable with the specific databases' query language can use it to substring and create any number of elements as desired. For our purpose here, the goal is to substring the source into a series of character fields with none exceeding the 32K length limit. In this example, strings are divided into 30,000 byte lengths which reserves space for some subsequent string manipulations. In most database systems, this is accomplished using the substring command. One can select the first 30,000 bytes of a string with:

```
substring (field, 1, 30000) as field_1
```

This returns a variable (`FIELD_1`) with the text from the first 30,000 characters of `FIELD`. A second substring can select the next set of 30,000 bytes:

```
substring (field, 30001, 60000) as field_2
```

Assuming that `FIELD` is no longer than 90,000 characters, a basic query can be constructed that would substring the data into three smaller character variables: `FIELD_1`, `FIELD_2`, and `FIELD_3`. This is:

```
select
    table_key
    , substring (field,     1, 30000) as field_1
    , substring (field, 30001, 60000) as field_2
    , substring (field, 60001, 90000) as field_3
from <table>
order by table_key
```

where table is the SQL table name. Depending on the specific access engine and settings within the connect statement, it may be necessary to include schema.table or even database.schema.table naming for the input table definition.

This query could be used as <sql query> in the explicit pass-through logic above to produce a SAS data set with the table key plus three fields which contain the entirety of the original text string.

GENERALIZING THE SQL LOGIC FOR ANY STRING LENGTH

The above example works fine for simpler cases, but does not generalize well to longer and more varied string lengths. SAS macro logic can be used to generalize the structure to handle any string length. Since the macro processor will resolve code before code execution, the macro language can be used as a code generator within the explicit pass-through query. Thus, it can be used to generalize the SQL code to work with any length and number of substrings.

The generalization process occurs in three steps. The first step determines the maximum string length of any string to be returned, while step 2 determines the total number of substrings required to store the entire string contents. The final step uses a macro to generate the native SQL code necessary to extract each of the required substrings.

DETERIMINE MAXIMUM STRING LENGTH

In some cases, the developer may know in advance the maximum string length. This is often true when strings may be long but very consistent in length. In other cases, string length may vary greatly, especially if the string is derived from multiple business processes. If the length is known or can be set adequately large, this first step can be skipped. If not, SAS can simply query the database to determine the maximum string length to be returned by the query. To accomplish this, construct an explicit pass-through query, using equivalent WHERE logic to that required in the final extract and use the SQL LEN function to return the input string length. The explicit pass-through can be combined with the INTO logic to create a macro variable that records the maximum string length. Note that the use of MAX, a grouping function, with no GROUP BY will result in one record returned with the maximum length recorded across all queried rows.

Suppose your goal is to extract the text from `FIELD` for `DATE` since '`1/1/2017`' and `TYPE` equal '`A`'. The query to determine the maximum string length and store it in a macro variable `MaxLen` would look like:

```
proc sql noprint;
    connect to OLEDB AS SRC (<options>);
    select MaxLen
          into :MaxLen
    from connection to SRC
          (
          select
                max(len(field)) as MaxLen
          from <table>
          where type = 'A'
                and date >= '01/01/2017'
          );
    disconnect from SRC;
quit;
```

For those unfamiliar with the INTO logic used here, use of the INTO statement with a variable name proceeded by a ': ', would produce a similar result as %let with the same variable name set equal to the result of the query.

The explicit portion of the query selects the desired field for output from `<table>`, determines the length of `FIELD` for each record, and returns the maximum value for all records selected as `MaxLen`. The SAS portion of this query captures the value of `MaxLen` and generates a macro variable named `MaxLen` with the resulting value. The NOPRINT option is used to turn off listing of the results which would typically occur when no output table is used. The only result of interest is captured in the macro variable.

Alternatively, the user can simply create a macro variable `MaxLen` to the desired value *a priori* and thus eliminate the need to execute this query.

DETERMINE THE NUMBER OF SUBSTRINGS

It is necessary to determine the number of substrings required to capture the longest field length. This is dependent on the length of the longest string and the length assigned to each substring field. It is important to retain both the substring length used and the string count required. These values can be generated using the custom `%compute_str_cnt` macro:

```
%macro compute_str_cnt (max_len =, string_length=30000);
    %global Str_Len Str_Cnt;
    %let Str_Len = &string_length;
    %let Str_Cnt = %eval(%eval((&max_Len-1)/&str_len) + 1);
    %put *** STR_CNT: &str_cnt  Str_Len = &str_len;
%mend compute_str_cnt;
```

This macro uses a default length of 30,000 but can be readily changed by adding that parameter to the macro call statement. The global macro statement is used to retain these two values for later use.

MACRO TO GENERALIZE SQL CODE

The final step is to generate SQL code for the required substring fields. The logic here follows the substring statements show above, but now generate these statements using macro loop logic. The macro requires parameters to identify the field to be parsed, the length of each substring, and the number of substrings to return. The substring variables will be returned with a format length that matches DBMAX_TEXT, so that value must be at least as large as STRLENGTH. In some cases, it may be advantageous to make DBMAX_TEXT larger than the specified string length but it cannot exceed 32K.

The SQL code is generated using the custom macro `%SQL_STR_LOOP`:

```
%macro sql_str_loop (field=, strlength =, count=);
    %local c;
    %do c = 1 %to &count;
          ,substring(&FIELD,  %eval(&strlength.*(&c. - 1) + 1),
          %eval(&strlength.*&c.)) as &field._&c.
    %end;
%mend sql_str_loop;
```

This macro generates new substring fields using the original field name with an appended numeric counter. Since the macro will be called from within the explicit pass-through logic, syntax must conform to native SQL code.

The SQL code shown above can now be updated with this new macro and appears like:

```
select
    table_key
```

```
    %sql_str_loop (field=FIELD, strlength = &str_len, count=&str_cnt.)
    ,len(FIELD) as Field_Len
from <table>
order by table_key
```

Because the macro begins each SQL line with a comma, it is essential to include at least one field, such as a table key, before the macro call. Note, there is no comma between the table_key field and the macro call, since the macro logic inserts that comma.

Users may add any additional fields to the query as needed for their purpose. In this example, the length of the original string is recorded as `FIELD_LEN`. This can be used later to validate that all characters were successfully captured.

THE DATA STEP SIDE

Armed with the code elements above, the various substrings can be readily collected and processed within a DATA step. DATA step arrays make a great method here so that any processing can occur across ALL substrings. An ARRAY statement is needed to define the initial array:

```
array field {*} field_1-field_&str_cnt.;
```

Here, an array named `FIELD` is created with `&str_cnt` elements, containing each of the substrings generated by the SQL query. The array can be combined with a DO loop to process each field in the array. For instance, to search the entire string for the existence of the keyword, 'SAS', you could include code such as:

```
found = 0;
do c = 1 to dim(field) until (found);
    found = findw(field(c),'SAS',,'i');
end;
```

The logic here uses an iterative DO UNTIL statement. This is similar to a standard iterative DO where the logic applies a counter from 1 to N, but in this case, extends that logic to automatically terminate the loop if the UNTIL statement evaluates as TRUE. The DIM function used here returns the number of elements contained within the referenced array. The FINDW function searches each substring for the keyword SAS and will terminate the search once it is found. The 'i' modifier is used to make the search case insensitive.

This logic falls short on one critical exception. There is a possibility that the keyword began at position 29999, creating a field break in the middle of the keyword: `FIELD_1` would end with '`SA`' and `FIELD_2` would begin with '`S`'. The find function would fail to identify this instance.

In simpler cases, a concatenate function can be used to splice initial characters from the next string into the FIND function search string. That logic would appear like:

```
found = 0;
do c = 1 to dim(field) until (found);
    if c < dim(field) then
          found = findw(field(c) || substr(field(c+1),1,5), 'SAS',,'i');
    else
          found = findw(field(c), 'SAS',,'i');
end;
```

In this case, each substring is 30,000 characters while the SAS string limit is 32767. Under this structure, a maximum of 2767 characters can be concatenated from the next string segment.

This concatenation process can be easily used but may not lead to efficient code design. More advanced methods may be desired to define where the break between strings should occur. Breakpoints will be addressed later, but first, let's tie together the elements we have discussed so far.

COMBINING THESE ELEMENTS

The preceding logic and elements can be combined to generate a SAS data set that contains all characters from the longer text field contained with the DBMS. Excluding the macros that were already shown, the final code looks like:

```
proc sql noprint;
    connect to OLEDB AS SRC (... );

    select MaxLen
          into :MaxLen
    from connection to SRC
          (
          select
                max(len(field)) as MaxLen
          from table
          where type = 'A'
                and date >= '01/01/2017'
          );

    disconnect from SRC;
quit;

%compute_str_cnt (max_len = &MaxLen.);

proc sql;
    connect to OLEDB AS SRC (DBMAX_TEXT=32767 ...);

    create view raw_vw as
     select
          *
    from connection to SRC
          (
          SELECT
                [TABLE_KEY]
                ,[TYPE]
                ,[DATE]
                %sql_str_loop (field=FIELD, strlength = &str_len.,
                      count=&str_cnt.)
                ,len(FIELD) as Field_Len
          FROM table
          where type = 'A'
                and date >= '01/01/2017'
          );
    disconnect from SRC;
quit;

data raw;
    set raw_vw;
    array field {*} field_1-field_&str_cnt.;

    found = 0;
    do c = 1 to dim(field) until (found);
          if c < dim(field) then found =
                findw(field(c) || substr(field(c+1),1,5), 'SAS',,'i');
          else
```

```
                found = findw(field(c), 'SAS',,'i');
    end;
    drop c;
run;
```

DEFINING STRING BREAKPOINTS

Long strings often have natural breakpoints between words, phrases or sentences, codes, or tags. These breakpoints were not considered during the initial SQL extract. However, through use of some basic SAS string and loop functions combined with the extra 2767 characters held in reserve, SAS can be used to generate more natural breaks between the various substrings. When properly implemented, this method can simplify any subsequent string processing by eliminating the need for the concatenation logic used above.

One of the more challenging scenarios is to create breakpoints within an XML string. The tag structure of XML data provides a great scenario for both long strings and the need to incorporate natural breakpoints. First, a brief review of XML data for those who may not be familiar with XML tags.

INTRODUCTION TO XML TAGS

Although this paper is not the suitable place for instruction on XML, there are a couple basics that are required to follow the logic provided below. Consider a sample taken from within an XML string:

```
<tag1>Data for Tag 1</tag1><tag2>Data for tag 2</tag2>
```

Now suppose, the existing break occurred such that the first substring ended as:

```
<tag1>Data for Tag 1</tag1><tag2>Data
```

The desired break would be at the end of the first tag, or after </tag1>:

```
<tag1>Data for Tag 1</tag1>
```

Within XML structures, you can also observe tags where data are reported in attributes that are internal to the tag so that only one tag is used. In this case, the XML may appear like:

```
<tag3: Attribute="Data for Tag 3"/>
```

A suitable breakpoint would occur following this tag as well.

SPECIAL CONSIDERATION FOR SUBSTRINGS THAT TERMINATE WITH A SPACE

It is possible that the extracted string terminates with a space. This is problematic since the SQL extract will include an implicit RTRIM before loading into SAS. We can also leverage the breakpoint logic to ensure these spaces are returned to the original string.

In the tag1 example above, there is a fundamental and important difference between:

```
"<tag1>Data for Tag 1</tag1><tag2>Data"
```

and

```
"<tag1>Data for Tag 1</tag1><tag2>Data "
```

The first case works correctly since the space remains as the leading character of the next string. In the second case, the extract process will truncate the space and return a string that is 1 character shorter than expected. This trailing space was truncated by an implicit RTRIM function. Hence, when the two are concatenated, result would be "`Datafor tag 2`".

Identification and correction of this issue relies on the string length indicator. The original extraction used a string length set to `&Str_Len.`, set to 30,000 by default in the `%compute_str_cnt` macro. A string

that returns with length of 29998 would indicate two trailing spaces were lost during extraction. These spaces can be identified and reinserted as part of this process.

CREATING NATURAL BREAKPOINTS

The logic defined here searches each substring from right to left to determine the desired breakpoint. Any characters to the right of this point are cut from this string and pasted to the beginning of the next substring. Truncated spaces are also identified and recreated.

The process results in a DATA step that includes arrays and loops with some standard string functions:

```
data break;
    set raw;
    array field{&str_cnt.} field_1-field_&str_cnt.;

    length lst_string $2767
          tmp_string $32767;

    retain blanks '                    ';

    ** PROCESS ALL BUT LAST STRING WITH LOOP             **
    ** SEARCH FOR A TERM IS NOT REQUIRED FOR LAST STRING **;
    do c = 1 to dim(field) - 1;
          ** INITIALIZE PROCESS FOR FIRST STRING **;
          if c = 1 then do;
                lst_string = '';
                spaces = 0;
          end;

          ** CREATE TMP_STRING FROM ELEMENTS CARRIED FORWARD **
          ** PLUS ANY SPACES TO REINSERT AND THE NEW STRING  **;
          if spaces > 0 then
                lst_string = catx('',lst_string,substr(blanks,1,spaces));
          tmp_string = catx('',lst_string,field(c));

          ** SET SPACES FOR USE BY NEXT ITERATION **;
          spaces = &STR_LEN. - length(field(c));

          ** DETERMINE LENGTH OF NEW TMP_STRING **;
          str_len = length(tmp_string);

          ** FIND POSITION OF LAST XML TAG END MARKER **;
          tag = 0;
          do l = str_len to 1 by -1 until (tag);
                if substr(tmp_string,l,1) = '>' then tag = l;
          end;

          ** CONTINUE BACK UNTIL A TERMINAL TAG IS FOUND **;
          mark = 0;
          do l = tag-1 to 1 by -1 until (mark);
                if substr(tmp_string,l,2) = '</' or
                      substr(tmp_string,l,2) = '/>' then mark = l;
          end;

          ** FIND END OF TERMINAL TAG MARKER **;
          term = 0;
          do l = mark +1 to tag by 1 until (term);
                if substr(tmp_string,l,1) = '>' then term = l;
```

```
            end;

            ** CAPTURE CHARACTERS TO RIGHT OF TERM to ADD TO NEXT STRING **;
            LST_STRING = substr(tmp_string,term+1);

            ** UPDATE FIELD STRING WITH CHARACTERS TO LEFT OF TERM **;
            field(c) = substr(tmp_string,1,term);
      end;

      ** UPDATE LAST RECORD WITH ANY CARRYOVER FROM PREVIOUS SUBSTRING **;
      c = dim(field);
      if spaces > 0 then
            lst_string = catx('',lst_string,substr(blanks,1,spaces));
      tmp_string = catx('',lst_string,field(c));
      field(c) = tmp_string;

      ** FINAL LENGTH CHECK **;
      field_len2 = 0;
      do c = 1 to dim(field);
            field_len2 = field_len2 + len(field(c));
      end;
      check = (field_len=field_len2);

      drop tmp_string lst_string blanks spaces c;
run;
```

This DATA step uses many features already discussed including arrays and the iterative DO UNTIL. It does add a couple of unique features worthy of a brief discussion.

The process adds three character fields: TMP_STRING, LST_STRING, and BLANKS. LST_STRING captures any characters to be moved from one substring to the next. BLANKS is a simple empty character string that is used to replicate spaces that were truncated during the extract process. TMP_STRING is the concatenation of LST_STRING, any required SPACES, and the current substring field. It serves as the input to the breakpoint search process.

The process as defined is limited to a maximum LST_STRING length of 2767 characters and is the upper limit on characters that may be copied to subsequent strings plus truncated spaces to be inserted. The 2767 limit can be expanded, but it would require a matching reduction in the STR_LENGTH parameter used throughout the code.

The process iterates over all but the last substring. The assumption here is that a natural break occurs at the end of the string, so there is no need to search for one. It does require logic for the last substring to prepend any carryover in LST_STRING to it.

The process also uses a final length check. The total of lengths from each substring is generated (FIELD_LEN2) after the various string manipulation and compared to the original string length (FIELD_LEN) captured during the initial query. This will validate that all strings, spaces, and characters have been captured. It also provides a method to detect truncation that could occur if LST_STRING exceeds its upper length limit. Check is a 0/1 field with 1 indicating the string lengths match.

CONCLUSION

The process described in this paper provides a clear and efficient method to use SAS to process longer strings from a DMBS. I have used this approach with XML strings in excess of 200k with great success. The process provides flexibility without placing undue advanced programming burdens on the user. Many steps can be taken from here, depending on specific business requirements.

This process is not always the best approach. In some cases, users may only require specific elements from the DBMS string. In this case, it may be far more efficient to leverage explicit pass-through logic to capture just the required elements and retrieve only those items. Users who are more comfortable with the SQL syntax may wish to build the breakpoint logic elements into the initial explicit pass-through query. These options are beyond the scope of this discussion, but worthy of consideration.

That being said, the logic provided here does provide a solid foundation for the extraction and use of strings that exceed the 32K string limit and provides the necessary foundation so that SAS analysts can fully leverage the information contained within such strings.

REFERENCES

SAS 9.4 Programming Documentation, SAS Language Reference: Concepts, 6th Edition, SAS Institute, INC., Cary NC, 2016

CONTACT INFORMATION

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

John Schmitz
Luminare Data LLC
john.schmitz@luminaredata.com