Efficient Use of PL/SQL String Functions for Unicode Applications

An Oracle White Paper
July 2003

ORACLE

Efficient Use Of PL/SQL String Functions for Unicode Applications

Introduction 3
UTF-8 And PL/SQL String 4
Processing Functions 4
An Example of 6
Bad Programming Practices 6
Byte Mode Of String Functions 7
- Using Byte Mode String Functions When Characters Are of No Concern ... 7
- Use Byte Mode String Functions When You Know That Strings Are Composed of Pure Single-Byte Characters 10
- SUBSTRB and Blank Padding of Partial Characters 10

Summary 12

INTRODUCTION

The World Wide Web has changed the way we do business today, with an emphasis on the global market that has made the supporting of multilingual data a major requirement. While it is not impossible to use legacy character sets to configure a multilingual environment, it is very complicated, difficult, and expensive. This may force usage of separate databases to store data for different languages. These separate databases make information sharing difficult and inefficient, and the maintenance cost is high. To resolve these issues, customers need to host multilingual data in a single central database and support many different languages within the same application. Unicode is an answer to this need.

Unicode is a universal encoded character set that allows you to store information from any language using a single character set. Unicode provides a unique code value for every character, regardless of the platform, program, or language. With Unicode support, virtually all contemporary languages and scripts can be easily encoded. Unicode is fast becoming the standard character set for emerging technologies and has been adopted by many software and hardware vendors. It is the character set required by modern standards such as XML, Java, JavaScript, LDAP, CORBA 3.0 and WML. It is also compatible with the ISO/IEC 10646 standard. Many companies have migrated their legacy character set to Unicode in order to get ready for the data coming from all over the world.

A legacy Oracle database server can be migrated to support Unicode in two ways:

- Migrate the database character set to an Oracle character set with UTF-8 encoding
- Deploying Oracle's Unicode data type (that is, SQL NCHAR data type), migrate all or part of the database's SQL CHAR data to SQL NCHAR data, using Oracle's national character set encoding

For more information on database character set migration and the usage of Oracle's Unicode data type, please visit the Globalization home page on OTN and view the following papers: *Character Set Migration Best Practices* and *Migration to Unicode Datatypes for Multilingual Databases and Applications in Oracle 9i.*

This paper focuses on performance implications of PL/SQL applications for a Unicode database server using a UTF-8 database character set. We will analyze the

costs of using PL/SQL string functions on different types of character set strings and will discuss how good programming practices can keep the processing overhead of variable-width character set strings to a minimum.

UTF-8 AND PL/SQL STRING PROCESSING FUNCTIONS

After migrating from a legacy database character set to a UTF-8 database character set, PL/SQL applications may see a difference in performance. If the original database character set is a legacy multibyte character set, PL/SQL applications should benefit from better performance. If the original database character set is a single-byte character set, PL/SQL applications will likely have some performance degradation, especially when the application contains heavy string processing logic. This is because UTF-8 is a variable-width character set, which means that the number of bytes required to represent a character varies. The string and character processing algorithms for a variable-width multibyte character set need to look ahead to determine where character boundaries lie. Fixed-width character sets, including all single-byte character sets, can always manipulate characters at known byte boundaries. Generally speaking, the cost of identifying a character boundary in a single-byte or a fixed-width character set string is constant. We use the O-notation, O(1), to indicate that the cost will not change with the string length. On the other hand, the cost of identifying a character boundary in a variable-width character set string will increase with the length of the string. We use O(n) to indicate that the cost is a linear function of n, where n is the byte length of the string.

However, because UTF-8 follows a very well-defined encoding pattern, we can easily tell the byte offset and byte size of the character being pointed to without scanning the string from the very beginning. A UTF-8 codepath inside the Oracle RDBMS server has been specifically refined to take full advantage of this fact and hence make some of the UTF-8 string functions perform at a similar level as their single-byte counterparts. For example, the *CONCAT*() function can determine if a string contains any trailing partial characters by checking a few trailing bytes of the string instead of scanning the whole string. This means the cost of *CONCAT()* function is fixed no matter how long the strings to be concatenated are. Some other string functions, for example, *LENGTH*(), *INSTR*(), and *SUBSTR*(), must scan the string either completely or partially because the number of characters in the string is requested.

The following table lists the cost comparison of all PL/SQL built-in character functions when they operate on pure single-byte character strings or variable-width UTF-8 character strings:

STRING FUNCTION	AL32UTF8	WE8DEC	NOTES
ASCII	O(n)	O(n)	n is the length of the string in bytes
CONVERT	O(n)	O(n)	
INSTR	O(n*m)	O(m)	m is the length of the pattern string in bytes

INSTRB	O(m)	O(m)	
LENGTH	O(n)	O(1)	
LENGTHB	O(1)	O(1)	
LOWER/UPPER	O(n)	O(n)	
LPAD/RPAD	O(n+m)	O(n+m)	m is the length of the padding string in bytes. Note that because LPAD/RPAD is padding to display length, not storage width, even single-byte character set strings also must scan the whole string to calculate the display width of every character.
LTRIM/RTRIM/TRIM	O(n*m)	O(n*m)	m is the size of the trim set
NLS_LOWER/NLS_UPPER	O(c*n)	O(c*n)	c is a constant greater than 1
REPLACE	O(n*m)	O(n*m)	m is the length of the from string
SUBSTR	O(c*n)	O(1)	c is a constant less than 1
SUBSTRB	O(1)	O(1)	
TRANSLATE	O(n*m1+m2)	O(n*m1)	m1 is the length of from string; m2 is the length of the to string

From the table, we see that most functions implemented in a UTF-8 environment actually cost about the same as their single-byte counterparts. On the other hand, we can see that the *LENGTH*(), *SUBSTR*(), and *INSTR*() string functions may cause a larger performance degradation when strings are long in a UTF-8 environment. Their byte mode counterpart, *LENGTHB*(), *SUBSTRB*(), and *INSTRB*(), cost the same on multibyte character strings and single-byte character strings. So we should pay special attention when using these three string functions in a UTF-8 database. We will discuss how and under what circumstances the byte mode string functions should be used to improve performance in the following sections.

AN EXAMPLE OF BAD PROGRAMMING PRACTICES

Poor programming practices are often a side effect of schedule crunches. In such circumstances, even experienced programmers might write code that hampers performance. Poor programming practices include deploying inefficient algorithms, declaring variables that are never used, passing unneeded parameters to functions and procedures, and placing initializations or computations inside a loop needlessly. Some of these poor programming practices affect Unicode applications much more significantly than they do for a single-byte application. Here is an example:

```
i_writeamount :=0;
i_temp:='';

FOR i IN 1..ecx_print_local.i_tmpxml.count LOOP
   -- get the number of characters in I_tmpxml(i)
   i_writeamount := LENGTH(ecx_print_local.i_tmpxml(i));
   apnd_status := TRUE;

   IF (i_writeamount + LENGTH(i_temp) > 32000) THEN
      -- write to the xml doc
      dbms_lob.writeappend(i_xmldoc,LENGTH(i_temp), i_temp);
      apnd_status := FALSE;
      i_temp := '';
      i_temp := ecx_print_local.i_tmpxml(i);
   ELSE
      -- concatenation when the total writable is less than 32000 characters
      i_temp := CONCAT(i_temp, ecx_print_local.i_tmpxml(i));
   END IF;
END LOOP;

IF apnd_status = TRUE THEN
  dbms_output.put_line('Length of Document '|| LENGTH(i_temp));
  dbms_lob.writeappend(i_xmldoc, LENGTH(i_temp) , i_temp);
END IF;
```

In this code sample, *LENGTH()* has been called many times inside a loop on a local string variable, *i_temp*, to get the number of characters in the string. While new strings are concatenated to *i_temp*, *LENGTH(i_temp)* counts the number of characters in *i_temp* from the very beginning each time, resulting in unnecessary repeated counting. Executing this code on a Unicode database takes about 2.5 times more CPU time than executing it on a single-byte character set database. Because *LENGTH*() is expensive, we should use it with great caution. We will see in the following example that the number of characters in the string *i_temp* can be easily calculated. The underlined code is new:

```
i_temp              VARCHAR2(32000);
i_temp_lenInChar    NUMBER;
...
i_writeamount := 0;
i_temp := '';
i_temp_lenInChar := 0;
FOR i IN 1..ecx_print_local.i_tmpxml.count  LOOP
    i_writeamount := LENGTH(ecx_print_local.i_tmpxml(i));
    apnd_status := TRUE;
    IF (i_writeamount + i_temp_lenInChar > 32000) THEN
       dbms_lob.writeappend(i_xmldoc,i_temp_lenInChar, i_temp);
       apnd_status := FALSE;
       i_temp := '';
```

```
      i_temp := ecx_print_local.i_tmpxml(i);
      i_temp_lenInChar := i_writeamount;
   ELSE
      i_temp := CONCAT(i_temp, ecx_print_local.i_tmpxml(i));
      i_temp_lenInChar := i_temp_lenInChar + i_writeamount;
   END IF;
END LOOP;
IF apnd_status = TRUE THEN
    dbms_output.put_line('Length of Document ' ||i_temp_lenInChar );
   dbms_lob.writeappend(i_xmldoc,i_temp_lenInChar , i_temp);
END IF;
```

The new code reduces the performance overhead greatly, and the execution time is almost the same whether it is run on a single-byte database server or a UTF-8 database server.

BYTE MODE OF STRING FUNCTIONS

Several PL/SQL built-in string functions allow you to manipulate string data based on bytes, UCS-2 code units, UCS-4 code units, and Unicode characters. They are *LENGTH(), SUBSTR() ,INSTR()* and their variants, *LENGTHB(), LENGTH2(), LENGTH4(), LENGTHC(), SUBSTRB(), SUBSTR2(), SUBSTR4(), SUBSTRC()* and *INSTRB(), INSTR2(), INSTR4(), INSTRC().* The default behavior of *LENGTH()*, *SUBSTR()* and *INSTR()* is based on the UCS-4 encoding for all character sets except for UTF8 and AL16UTF16, where the UCS-2 encoding is used. Pay special attention when using these string functions because they may return different results when the character set changes. To guarantee correct results, you should use variants of these functions designed for multibyte character sets. Please refer to *Oracle9i SQL Reference* and *Oracle9i Globalization Support Guide* for detailed descriptions of their behavior for different kind of character sets.

In this section, we focus on the byte mode of *LENGTH()*, *SUBSTR*(), and *INSTR()* and illustrate how they can improve UTF-8 string processing performance.

Using Byte Mode String Functions When Characters Are of No Concern

Let's start with the following example:

```
DECLARE
   TYPE  t_simple_table
   IS TABLE OF VARCHAR2(2000) INDEX BY BINARY_INTEGER;
   l_simple_table t_simple_table;

   -- procedure string_to_table put string fields into a table, the string field is
   -- separated by a token, p_separator, which is a single character
   PROCEDURE string_to_table(p_separator  IN VARCHAR2,
                             p_string     IN VARCHAR2,
                             p_table      OUT t_simple_table)
   IS
            l_value_index NUMBER := 1;
           l_index_start  NUMBER;
           l_index_next   NUMBER;
           l_loop_count   NUMBER;
```

```
        l_result        VARCHAR2(2000);
        l_count_ex    BOOLEAN;

BEGIN
    -- start from the first character following the first token in the string
    l_index_start := INSTR(p_string, p_separator, 1, 1) + 1;

    -- NULL string or string contains no specified token
    IF (l_index_start = 1 OR l_index_start IS NULL) THEN
        RETURN;
    END IF;

    l_loop_count := 0;
    LOOP
        -- character offset of next token in the string
        l_index_next := INSTR(p_string, p_separator, l_index_start, 1);

        IF (l_index_next = 0) THEN                    -- No more token in the string, get last field
            l_result := SUBSTR(p_string, l_index_start, LENGTH(p_string) + 1 -
l_index_start);
        ELSE                                                -- Get the field: starting from character offset
l_index_start, get
                                                        -- 'l_index_next - l_index_start'   characters
from string p_string
            l_result := SUBSTR(p_string, l_index_start, l_index_next - l_index_start);
        END IF;

        IF l_result = NULL' THEN
            l_result := NULL;
        END IF;

        p_table(l_loop_count) := l_result;
        l_index_start := l_index_next + 1;
        l_loop_count := l_loop_count + 1;
        EXIT WHEN l_index_next = 0;
        IF (l_loop_count > 30000) THEN
            l_count_ex := TRUE;
        END IF;
    END LOOP;
END string_to_table;
```

Pass in a string, *'1|2|3|4|5|6|7|8|9|10|'* and *'|'* as the separator, we'll get a ***simple_table*** result:

```
p[0] = '1'
p[1] = '2'
p[2] = '3'
p[3] = '4'
p[4] = '5'
p[5] = '6'
p[6] = '7'
p[7] = '8'
p[8] = '9'
p[9] = '10'
p[10] = NULL
```

This PL/SQL code may run much more slowly on a Unicode database than on a singlebyte database because of the heavy use of string functions *SUBSTR()*,

INSTR() and *LENGTH*(). First, remove the call to *LENGTH(P_STRING)* in the following line:

```
l_result := SUBSTR(p_string,l_index_start, LENGTH(p_string)+1-l_index_start);
```

This code line gets the end of the string, *p_string*, from the character offset *l_index_start*. The code should be changed to:

```
l_result := SUBSTR(p_string, l_index_start);
```

When the size of the substring is omitted, *SUBSTR* takes the rest of the string starting from the given offset. There is an optimized implementation for this special case of *SUBSTR*: it checks a few trailing bytes of the source string to make sure that no partial character is copied to the resulting substring. No string scanning from the *offset* to the end of the string needs to be done. While the original code line scans the whole *p_string* (caused by *LENGTH(p_string)*), the modified code line needs to check only several trailing bytes of *p_string*.

Second, in this tokenizer-like code, it is always safe to use the byte mode of *LENGTH*(), *SUBSTR*() and *INSTR*(), that is, to use *LENGTHB*(), *SUBSTRB*() and *INSTRB*() if the *p_string* and *p_separator* contain no partial or invalid characters, because it does not matter if the index offset is the character offset or the byte offset. The absolute position is always the same, whether it is on the *nth* character or on the *mth* byte. With a little change to the original code, we can switch to the byte mode variants of *LENGTH()*, *SUBSTR()*, and *INSTR()* and still get the same result even in a UTF-8 database. The performance will also be the same as in a single-byte database. The highlighted lines have been changed in the following example:

```
DECLARE
TYPE t_simple_table
IS TABLE OF VARCHAR2(2000) INDEX BY BINARY_INTEGER;
l_simple_table t_simple_table;

-- procedure STRING_TO_TABLE put string fields into a table, the string field is
-- separated by a token, p_separator
PROCEDURE string_to_table(p_separator  IN VARCHAR2,
                          p_string     IN VARCHAR2,
                          p_table      OUT t_simple_table)
IS
        l_value_index NUMBER := 1;
        l_separator_len NUMBER;
        l_index_start NUMBER;
        l_index_next NUMBER;
        l_loop_count NUMBER;
        l_result varchar2(2000);
         l_count_ex BOOLEAN;
BEGIN
   -- byte length of the separator
   l_separator_len  :=  LENGTHB(p_separator);
    -- start from the first byte following the first token in the string
   l_index_start := INSTRB(p_string, p_separator, 1, 1) + l_separator_len;

   -- NULL string or string contains no specified token
   IF (l_index_start  =  l_separator_len  OR  l_index_start IS NULL)  THEN
```

```
      RETURN;
    END IF;
    l_loop_count := 0;
    LOOP
      -- byte offset of next token in the string
      l_index_next := INSTRB(p_string, p_separator, l_index_start, 1);

      IF (l_index_next = 0) THEN                    -- No more token in the string, get last field
        l_result := SUBSTRB(p_string, l_index_start);
      ELSE
        -- starting from character offset l_index_start, get 'l_index_next - l_index_start' characters
        l_result := SUBSTRB(p_string, l_index_start, l_index_next - l_index_start);
      END IF;

      IF l_result = 'NULL' THEN
        l_result := NULL;
      END IF;

      p_table(l_loop_count) := l_result;
      l_index_start := l_index_next + l_separator_len;
      l_loop_count := l_loop_count + 1;
      EXIT WHEN l_index_next = 0;

      IF (l_loop_count > 30000) THEN
        l_count_ex := TRUE;
      END IF;
    END LOOP;
  END string_to_table;
```

This new code illustrates that a simple but powerful arrangement takes care of the character boundaries properly when byte mode string processing functions are used. We should never assume that a character is one byte long.

Use Byte Mode String Functions When You Know That Strings Are Composed of Pure Single-Byte Characters

A string is composed of pure single-byte characters if the number of characters in the string is the same of the number of bytes in the string. That is, if *LENGTH(str) = LENGTHB(str),* then *str* contains only single-byte characters. It is always safe to use byte mode string functions on these kinds of strings. Use knowledge of the contents of the string to gain better performance.

SUBSTRB and Blank Padding of Partial Characters

By default, *SUBSTRB*() will replace any partial characters in the result substring with blanks. Assume a UTF-8 string, *'A1A2B1C1C2C3D1D2'*, in which *A1A2* represents a 2-byte character, *B1* represents a 1-byte character, *C1C2C3* represents a 3-byte character, and *D1D2* represents another 2-byte character:

```
SUBSTRB('A1A2B1C1C2C3D1D2', 2, 6)
```

Requests a substring starting from byte offset 2 and containing 6 bytes, which is *'A2B1C1C2C3D1'*. Because A2 is the second byte of the 2-byte character *A1A2*, and *D1* is the first byte of the 2-byte character *D1D2*, *SUBSTRB*() will replace *A2* and *D1* with blanks. So the result of

```
SUBSTRB('A1A2B1C1C2C3D1D2', 2, 6)
```

Is *'[space]B1C1C2C3[space]'*, where *[space]* is a single-byte space character. You can change this default behavior of *SUBSTRB*() by setting a event in the *init.ora* file as follows:

```
event = "10943 trace name context forever, level 524288"
```

Event 10943 indicates that *SUBSTRB*() will not replace the partial characters with spaces but simply skips them. So the result of the above call to *SUBSTRB*() will be *'B1C1C2C3'* when event 10943 is set in the *init.ora* file.

This feature can improve performance in specific situations. For example, due to the maximum size limitation of a string buffer allowed in PL/SQL, a big chunk of character data may need to be cut into many smaller buffer-sized pieces and then processed consecutively:

```
...
chr_offset  := 1;
LOOP
    buffer  := SUBSTR(local_msg,  chr_offset, 2000);
    EXIT WHEN buffer = NULL;
    -- operations on buffer
   ....
   chr_offset := chr_offset + 2000;
END LOOP;
...
```

The *SUBSTR()* in the loop is a very expensive operation in a UTF-8 environment because it has to scan *local_msg* from the beginning to find the character offset *chr_offset,* and then count another 2000 characters to get the requested substring. We can use *SUBSTRB*() to avoid expensive repeated string scanning:

```
...
byte_offset := 1;

LOOP
    buffer  := SUBSTRB(local_msg,  byte_offset, 2000);
    EXIT WHEN buffer = NULL;
    -- operations on buffer
   ....
   byte_offset := byte_offset + LENGTHB(buffer);
END LOOP;
...
```

Here we update the *byte_offset* in the loop by adding *LENGTHB(buffer)* to it because the 2000th byte starting from *byte_offset* may not fall on a character boundary, in which case *substrb* will cut the ending partial character when the following is specified in the *init.ora* file:

event = "10943 trace name context forever, level 524288"

SUMMARY

Because of the specific characteristic of UTF-8 as a variable-width character set encoding, it always requires more processing power for UTF-8 string manipulations compared to single-byte character string manipulations. Programming carefully and using full knowledge of the string data can help reduce the UTF-8 performance overhead. Keep the following rules in mind when dealing with UTF-8 strings:

- Avoid calling unnecessary string functions
- Keep the string being processed as short as possible
- Use byte mode string functions when it is easy to control the character boundaries in the program logic

White Paper Title
[June] 2003
Author: lei zheng
Contributing Authors:

Oracle Corporation
World Headquarters
500 Oracle Parkway
Redwood Shores, CA 94065
U.S.A.

Worldwide Inquiries:
Phone: +1.650.506.7000
Fax: +1.650.506.7200
www.oracle.com

Copyright © 2003, Oracle. All rights reserved.
This document is provided for information purposes only and the contents hereof are subject to change without notice. This document is not warranted to be error-free, nor subject to any other warranties or conditions, whether expressed orally or implied in law, including implied warranties and conditions of merchantability or fitness for a particular purpose. We specifically disclaim any liability with respect to this document and no contractual obligations are formed either directly or indirectly by this document. This document may not be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose, without our prior written permission.
Oracle is a registered trademark of Oracle Corporation and/or its affiliates. Other names may be trademarks of their respective owners.