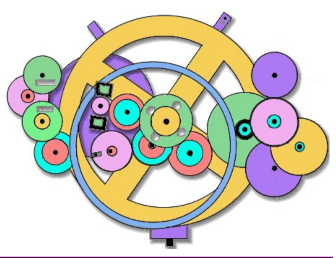
Antikythera Publications



Relational Database Design
http://www.AntikytheraPubs.com
sweiss @ AntikytheraPubs.com

Database Design Note Series

Exploring UTF-8
Multi-script Database Series #3

Prepared by: S. L. Weiss and F. Oberle

Like it or not, Information Technology today – regardless of the type of organization supported – is becoming more global in nature with each passing day.

Earlier technologies limited character data storage to a small subset of the letters required to record our customers' names (as one example) – one or two Scripts at most – but with Unicode and UTF-8 we can record those names in a way those customers will recognize regardless of where they live or what language they speak.

Thus, among the longer term objectives for any organization with global aspirations is the ability to store their far-flung customer's given names as آمنة, นัฐพงษ์, Jennifer, りく or आदित्या. These, of course, are only their given names, and we may want to store transliteration data as well (Aminah, Nattapong, Jennifer, Riku and Aditya respectively), but such design considerations are not in the scope of this paper. Here we'll simply explain the mechanics of storing multi-Script data efficiently (UTF-8), and why certain schema modifications need to be accomplished – thus providing some minimal comfort level for those about to embark on the necessary upgrades, and showing the way for other elements that will need to become Script-agnostic.

Revised for public distribution: 19 December 2016

See page 13 for information on other material from Antikythera Publications.

Copyright © 2016 by the Authors and Antikythera Publications

Permission is granted to distribute unaltered copies of this document, so long as this is not done for commercial purposes.

www.AntikytheraPubs.com

Database Design Note Series on Multi-Language/Multi-Script Databases

1. Exploring Alphabets
2. Exploring Complex Text Layout
3. **Exploring UTF-8**
4. Evaluating Fonts for use in Multi-Lingual Documents
5. Exploring Bidirectional Text Entry
6. Evaluating Bidirectional Text Handling Behavior in Applications

Database Design Note Series – Exploring UTF-8

Definitions and Concepts

We'll first review some very basic terms and concepts that are often not as clearly understood as they might be. Following those definitions we'll cover just enough history to allow the reader to learn why things are the way they are, how UTF-8 is used to efficiently structure Unicode values and, finally, some ramifications to database schema design.

Language

For our purposes, we are using the word Language to mean a form of spoken or written communication between humans. The focus of this paper, however, is more on written forms, and even more on the actual Scripts used by these forms. Further details concerning "Language" can be found in "Languages, Dialects & Countries" in the first Design Note of this series, titled "Exploring Alphabets."[1]

Script

In the context of this discussion, a Script is a collection of symbols used to write or print some subject matter. The alphabet used to print English is a subset of Latin Script, for example. German, French and Spanish use slightly different subsets of Latin Script for their alphabets. German has the ß as well as various characters with umlauts (e.g. ü); French has the ç and some letters with a variety of accents (á), while Spanish uses the ñ, which is not the same as, nor sorted with, the n character.

The Script in use determines the order in which the characters are displayed or printed. Any of the many languages using Arabic Script, for example, will be displayed from right-to-left. As with Languages, Scripts were covered in more detail in "Exploring Alphabets."

The Relationship between Language and Script

Because this paper is primarily targeted towards relational database designers, the easiest way to clarify the relationship between Language and Script is to present it as a pair of Normalized Propositions:

A Language may be written with one or more Scripts.

But: A Language does not always use the entirety of the Script from which its alphabet is taken, and these Script options are not usually used simultaneously for other than educational documents.

For the record, there are a number of Languages which are commonly written in different Scripts, among which are Serbian (can be written in Latin and Cyrillic), Azeri (can be written in Latin, Cyrillic and Arabic – in which case both its characters/glyphs as well as its layout direction change.

A Script may be used with zero or more Languages.

Any given Language may use all or part of one or more Scripts for its writing.

Certain Scripts are not used for Languages per se, but for specialized purposes from mathematics and music to symbols for game pieces, diagrams, and so forth.[2]

Any Language in common use is, of course, a living entity. The English spoken in Great Britain, Australia and the United States is similar, but not identical. The English written by Dickens, Shakespeare, Chaucer and the author of Beowulf might even be considered different Languages depending on how strict a definition is applied to the term.

Characters and Character Cells

The distinction between a Character and a Character Cell must be understood when exploring any of the Unicode Transformation Formats; that is also covered in "Exploring Alphabets" which should be read first.

---

1 Like the other Design Notes in this series, a pdf version of "Exploring Alphabets" is available as a fee download from www.AntikytheraPubs.com/i18n.htm. The word "alphabet" as used in this document may be taken to include Abugidas or Abjads if appropriate. These terms are also discussed in "Exploring Alphabets."

2 See http://unicode.org/charts/ to get an idea of the wide variety of Scripts defined by the Unicode Standard.

Table of Contents

Definitions and Concepts....3
Language....3
Script....3
The Relationship between Language and Script....3
Characters and Character Cells....3
Unicode (ISO/IEC-10646) Code Point Representations....4
Unicode Transformation Formats (UTF)....5
One Byte UTF-8 Representations (7 content bits): Decimal values from 32 through 127 (96 code points)....5
Two Byte UTF-8 Representations (11 content bits): Decimal values from 128 through 2,047 (1,920 code points)....6
Three Byte UTF-8 Representations (16 content bits): Decimal values from 2,048 through 65,535 (63,488 code points)....6
Four Byte UTF-8 Representations (21 content bits): Decimal values from 65,536 through 1,114,112 (1,048,577 code points)....6
One byte UTF-8 Representation – Details....6
Two byte UTF-8 Representation – Details....7
Three byte UTF-8 Representation – Details....7
Four byte UTF-8 Representation – Details....8
Review – Characters and Character Cells....8
Implications of UTF-8 on Database Schema Design....9
Boundary Conditions and Column Sizing....10
Back from UTF-8 to Unicode....10
Thai (Unicode Plane u+0E01-0E7F) Text Sample....11
Hindi (Devanagari Unicode Plane u+0900-097F) Text Sample....12
Hebrew (Unicode Plane u+0590-05FF) Text Sample....12

Unicode (ISO/IEC-10646) Code Point Representations

Unicode provides individual code points, or values, to represent up to 1,114,112 unique symbols used to write all of the world's present and past languages. Not all of these have yet been assigned of course, but for any language likely to be used in today's computer applications, this number is quite sufficient. In binary, that number is a series of twenty-one bits: `100010000000000000000`. Because computers store numbers in bytes – sequences of eight bits – this series of twenty-one bits can be represented as bytes by adding three leading *zero* bits to give twenty-four bits, or an even three bytes as follows: `00010001 00000000 00000000`. In hexadecimal notation that would be `0x110000`.

If all our data storage and transmissions were simply converted from the one-byte characters used by early adopters of computer technology to the three-byte character representations required to store Unicode values, text in all the world's languages could be reliably stored and exchanged, right? The short answer is No. A little thought will show that data reliability can easily disappear in many circumstances.

If transmissions are interrupted, which bytes in the sequence are the first of the three used for each symbol? It might appear that, since we've arbitrarily added some leading *zeros*, we could perhaps use those positions to indicate the leading byte – perhaps by making that byte begin with a 1 rather than a 0. We won't go into detail here, but some further thought will indicate that there is no guarantee at all that some of the second and third bytes might not also legitimately begin with a 1. Byte streams could easily be completely misinterpreted!

And then there is the issue of file size. Since the dawn of the computer age, most data files were produced by countries who concerned themselves only with Latin symbols and their own character sets. One-byte-per-character was often sufficient for two "alphabets," but never for more than that. A conversion to three-byte representations would therefore

result in a significant portion of the world's data files immediately tripling in size; coupled with the higher error rates and resulting repeat transmissions due to the drop in reliable identification of the characters, that simply wouldn't do. But supporting multiple character sets simultaneously was becoming more of a necessity than a desire. Enter UTFs.

UNICODE TRANSFORMATION FORMATS (UTF)

Covering the rocky path to a reasonable solution is beyond the scope of this document but, in brief, a number of attempts were made to settle on some ways to "transform" Unicode bit sequences into patterns that could be reliably distinguished; these eventually became Unicode Transformation Formats, the three most prominent of which were:

- UTF-32: a fixed width format in which each symbol is stored as a four byte (32 bit) unit; UTF-32 is essentially the natural twenty-two bit sequence described above extended to cover standard storage sizes. Thus, all the issues discussed earlier are relevant. This format is also known as UCS-4, although UTF-32 is actually a subset of UCS-4.[3]
- UTF-16: a variable length format where each symbol is stored as either one or two double-byte (16 bit) units, and is therefore either 16 or 32 bits wide; although derived from the earlier UCS-2, it is not the same. Many programming language libraries continue to use UTF-16 for internal storage, although that is slowly changing.[4] And, then, finally came …
- UTF-8: a variable length format where each symbol is stored as anywhere from one to four bytes (8 to 32 bits). Although this might initially seem unwieldy from programming or data transmission standpoints, it is becoming recognized as the most sensible compromise between efficiency and size.

Essentially, each reduction from UTF-32 to UTF-8 represented an improvement in granularity. The remainder of this section will be devoted solely to examples of how UTF-8 layouts correspond to Unicode point values.

Contrary to some on-line discussions, the "8" in the UTF-8 designation does not indicate that each character consists of eight bits (one byte), but rather that each character consists of exact multiples of eight bits, i.e. complete bytes. A UTF-8 character, as mentioned earlier, may consist of one to four bytes. A single byte UTF-8 code point representation always begins with a binary *zero*. The corollary is that any byte beginning with a *zero* bit represents a single Unicode symbol. These single byte characters are direct descendants of what used to be termed "lower ASCII," and several of the non-alphabetic characters in this range are shared among many of the world's scripts.

The first byte of each multi-byte UTF-8 code point representation begins with one, two or three *one* bits in sequence followed by a *zero*. The number of *one* bits indicates the total number of bytes that form the character – thus a byte beginning with three *one* characters followed by a *zero* indicates the beginning of a three byte sequence, and that two more bytes will follow to complete the representation of the code point.

Any byte beginning with a *one-zero* sequence is a continuation byte, and only has meaning when considered with an initial byte, i.e. a byte beginning with two or more *one* bits.

This byte layout structure, while not guaranteeing data reliability during transmissions, significantly improves a system's ability to recover from interruptions by enabling easier synchronization of individual characters.

The tables below illustrates the construction and range of Unicode values that can be represented by the one, two, three, and four byte Unicode UTF-8 transformation layouts:

One Byte UTF-8 Representations (7 content bits): Decimal values from 32 through 127 (96 code points)

Minimum Value		Maximum Value	
Unicode Binary:	o010.0000	Unicode Binary:	o111.1111
Unicode Hexadecimal:	0x20	Unicode Hexadecimal:	0x7f
Unicode Decimal:	32	Unicode Decimal:	127
UTF-8 Binary:	**0**010.0000	UTF-8 Binary:	**0**111.1111
UTF-8 Hexadecimal:	0x20	UTF-8 Hexadecimal:	0x7f
UTF-8 Decimal:	32	UTF-8 Decimal:	127

3 The acronym UCS means Universal Character Set, and refers to the initial attempts to represent all the Unicode code points.

4 As will become apparent, this is not sufficient to handle any Unicode code points with values higher than 2,047.

Two Byte UTF-8 Representations (11 content bits): Decimal values from 128 through 2,047 (1,920 code points)

Minimum Value		Maximum Value	
Unicode Binary:	oooo.o000 1000.0000	Unicode Binary:	oooo.o111.1111.1111
Unicode Hexadecimal:	0x0080	Unicode Hexadecimal:	0x07ff
Unicode Decimal:	128	Unicode Decimal:	2,047
UTF-8 Binary:	**110**0.0010 **10**00.0000	UTF-8 Binary:	**110**1.1111 **10**11.1111
UTF-8 Hexadecimal:	0xc280	UTF-8 Hexadecimal:	0xdfbf
UTF-8 Decimal:	49,792	UTF-8 Decimal:	57,279

Three Byte UTF-8 Representations (16 content bits): Decimal values from 2,048 through 65,535 (63,488 code points)

Minimum Value		Maximum Value	
Unicode Binary:	0000.1000 0000.0000	Unicode Binary:	1111.1111 1111.1111
Unicode Hexadecimal:	0x0800	Unicode Hexadecimal:	0xffff
Unicode Decimal:	2,048	Unicode Decimal:	65,535
UTF-8 Binary:	**1110**.0000 **10**10.0000 **10**00.0000	UTF-8 Binary:	**1110**.1111 **10**11.1111 **10**11.1111
UTF-8 Hexadecimal:	0xe0a080	UTF-8 Hexadecimal:	0xefbfbf
UTF-8 Decimal:	14,721,152	UTF-8 Decimal:	15,712,191

Four Byte UTF-8 Representations (21 content bits): Decimal values from 65,536 through 1,114,112 (1,048,577 code points)

Minimum Value		Maximum Value	
Unicode Binary:	ooo0.0001 0000.0000 0000.0000	Unicode Binary:	ooo1.0001 0000.0000 0000.0000
Unicode Hexadecimal:	0x010000	Unicode Hexadecimal:	0x110000
Unicode Decimal:	65,536	Unicode Decimal:	1,114,112
UTF-8 Bin:	**1111.0**000 **10**01.0000 **10**00.0000 **10**00.0000	UTF-8 Bin:	**1111.0**100 **10**01.0000 **10**00.0000 **10**00.0000
UTF-8 Hexadecimal:	0xf0000000	UTF-8 Hexadecimal:	0xf4908080
UTF-8 Decimal:	4,026,531,840	UTF-8 Decimal:	4,103,110,784

Values below 32 in the single byte representations remain reserved – as they have always been – for control codes such as STX and ETX, the line feed and carriage return, although the code to ring the teletype's bell (`0x07`) to signify an incoming message hasn't been used for many generations.[5] The number of possible code points listed for each section does not imply or suggest that all such positions (e.g. `0x7f`) are valid.

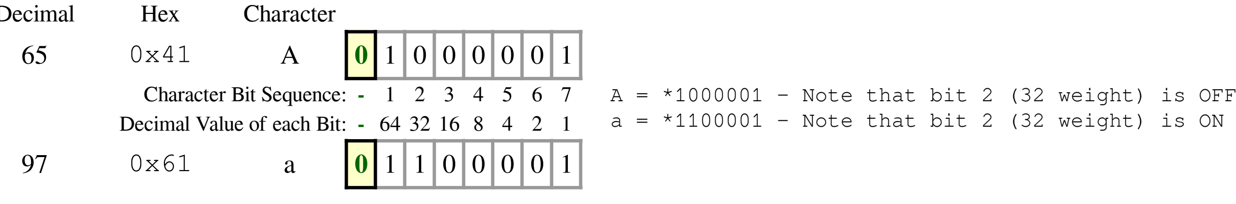
In order to clarify the Unicode transformations and illustrate one drawback of the UTF-8 format, the following charts will provide specific examples of UTF-8 representations for one, two, three, and four byte formats using several scripts, two of which have already been introduced in this document.

The mandatory bit values for UTF-8 encoding are highlighted like this: **0** – the remaining bits form what is known as the "payload" for the byte, i.e. the bits available for storing the actual Unicode values.

One byte UTF-8 Representation – Details

The one byte UTF-8 "transformations" are used for Unicode code points 32 (the space) through 126 (~) as well as for the standard control characters mentioned earlier. This range is usually referred to as "Basic Latin," but as discussed earlier in this paper, only values from `0x41` to `0x60` (decimal 65-90: A-Z), and `0x61` to `0x80` (decimal 97-122: a-z) should be considered as *solely* Latin Characters, since the others are shared by many scripts.

When pressing the A and a keys, the UTF-8 transformation is rather straightforward.

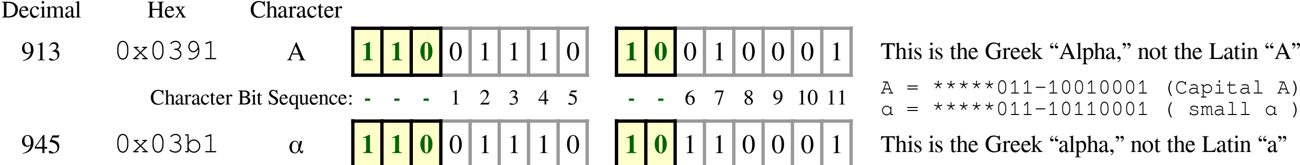


5 Sort of a precursor to AOL's infamous "You've got mail" message. The bell and clatter of old Kleinschmidt teletype machines engendered a greater sense of connection with the world than anything Facebook or Twitter currently provide!

In UTF-8 transformations, the leading 0 indicates this is a one byte character, and the final seven bits are the binary value of the character (i.e. its "payload"). Note that the 32-value bit (bit 2) value continues to determine the case difference between capital and small Latin letters. An "A" (decimal 65) with its 32 bit set becomes an "a" (decimal 97).

Two byte UTF-8 Representation – Details

Two byte transformations are used for Unicode code points `0x0080` to `0x07ff` (decimal 128-2047), and are illustrated here with examples from the Unicode Basic Greek script block `0x0370-03ff`, specifically the letters that result from using a Greek keyboard mapping with a Latin keyboard[6] and typing the same A and a keys as before.



In a UTF-8 layout, two consecutive leading 1 bits followed by a 0 indicates the character is two bytes long. Thus, the two-byte Unicode hexadecimal value `u+0391` is transformed to the two byte UTF-8 `0xce91` value.

Basic Greek, like Latin, also uses capital and small letters and, like the Latin, these are now identified by bit 2 of the payload, equivalent to decimal 32. The sixth through eleventh content bits are carried in the single continuation byte.

As noted above, two byte UTF-8 representations can contain Unicode code point values from 128 through 2,047. Thus, scripts such as Hebrew and Arabic, which are assigned code values from 1,424 to 1,535 (`u+0590-05ff`) and 1,536 to 1,791 (`u+0600-06ff`) respectively, can each be represented in two UTF-8 bytes and, therefore are no longer than the two bytes they would occupy in a straight binary representation of their code point values.

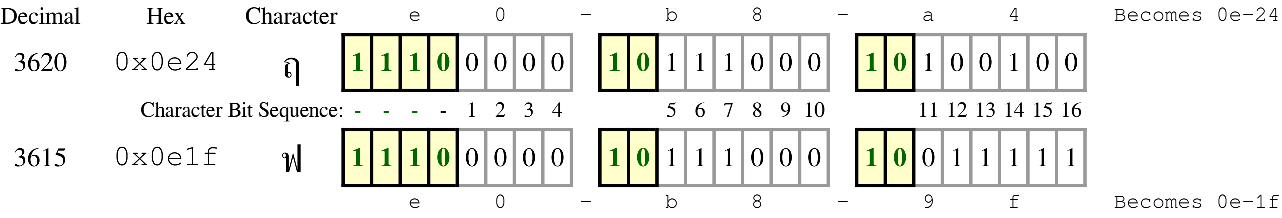
Characters in Devanagari and Thai scripts on the other hand, with assigned code values from 2,304 to 2,431 (`u+0900-097f`) and 3,584 to 3,711 (`u+0e00-0e7f`) respectively, each of which requires only two bytes in a raw binary representation, require three bytes each in their UTF-8 transformations. This illustrates the major compromise of the UTF-8 transformation format. Scripts residing at the range boundaries approach a size that is double[7] what it was without the transformation. The implications of this on setting database column widths will be discussed later in this paper.

As a matter of interest, the N'Ko script used to write the Maninka, Bambara, and Dyula languages and variants used in Guinea, Côte d'Ivoire and Mali, with an assigned Unicode block that ranges from 1,984 to 2,047 (`U+07c0-07FF`), contains the highest value characters in two byte UTF-8 transformations.

Three byte UTF-8 Representation – Details

The lowest values requiring three byte representations in UTF-8 belong to Samaritan, a right-to-left script used in ancient Hebrew and Aramaic and defined in the Unicode block (`u+0800-083f`).[8]

The three byte UTF-8 representation example illustrated here uses Thai script, which is located in the Unicode Block `u+0e00-0e7f`, and uses the letters that result from using a Thai TIS-820 keyboard mapping on a Latin keyboard, and typing the same A and a keys as before.



6 Details for various means of entering these characters, including the use of Input Method Editors (IMEs) can be found in the previous Design Note in this series: "Exploring Complex Text Layout" which should already be familiar to you.

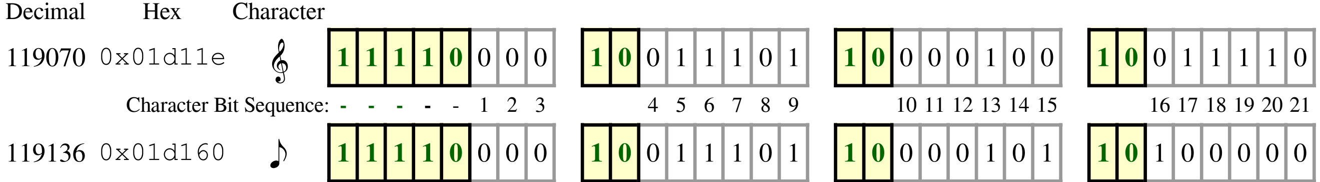
7 These generally are not completely doubled, because of the single byte shared characters (e.g. the space) used by these scripts.

8 A form of this script is still in use by a very small group of people in the modern day Palestinian West Bank.

Three consecutive leading 1 bits followed by a 0 indicates the character is three bytes long. There will therefore need to be two continuation bytes, each of which must begin with 10. In this case, the two-byte Unicode hexadecimal value `U+0E24` is transformed to the three byte UTF-8 `0xe0b8a4` value.

Four byte UTF-8 Representation – Details

For an example of a four byte UTF-8 representation, the table below uses symbols from the Unicode Musical Symbols Block `u+1d100-1d1ff`. In this case, the equivalent decimal values for that block range from 119,040 to 119,295, each requiring seventeen bits (three bytes if rounded up) to represent in raw binary, but four bytes when transformed into UTF-8, another example of the increasing in size resulting from transformations near range boundaries.

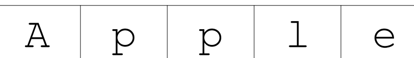


Four consecutive leading 1 bits followed by a 0 indicates the character is four bytes long. There will therefore need to be three continuation bytes, each of which must begin with 10. As with the other examples, the seventeen content bits are placed in the UTF-8 structure, transforming the three byte `u+01d11e` to the four byte `0xf09d849e` transformation.

The site https://docs.oracle.com/cd/B19306_01/server.102/b14225/ch6unicode.htm from Oracle (but not specific to its products) provides additional and more detailed information about Unicode and its various transformation formats.

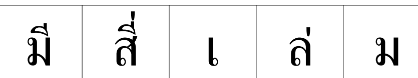
Review – Characters and Character Cells

α. Latin Characters in Virtual Cells



In English, there are 5 characters in 5 virtual character cells.

β. Thai Characters in Virtual Cells



Here there are 9 nine independent characters in 5 virtual character cells.

As mentioned earlier, this distinction was covered at length in the first Design Note of this series (see footnote 1) but, because of its importance in dealing with multi-lingual, multi-script systems, this section will provide a recap.

In English, we consider each character to be an independent entity occupying its own "character cell." When we see a line of text such as "Apple" we make no distinction between the concept of "Character" and "Character Cell." In example α on the right each of the five characters that form the word Apple has its own virtual cell."[9]

But with many Scripts there isn't a simple one-to-one correspondence between a character and a character cell. In Thai, for instance, the sentence "(I) have four books" is "(ฉัน)มีสี่เล่ม"[10], Thai uses a number of "dead keys" that occupy no character cell of their own, but are meant to be "attached" to some other character. These diacritics include both vowels and tone markings. The first displayed cell contains the consonant ม over which the vowel ี is positioned. The second cell contains the consonant ส with the same vowel as well as a tone mark. Only the third and fifth cells contain a single character. What is critical to recognize, however, is that there are nine independent characters stored on disk when this short phrase is entered (twelve if the optional personal pronoun ฉัน is included).

Along with a knowledge of the highest value Unicode character points to be handled in the database, setting the appropriate sizing can be challenging if the implications of this lack of one-to-one correspondence between characters and character cells on the sizing of columns in a database is not understood. The next section will provide some background to support basic approaches to adjusting column sizes in multi-lingual, multi-script data schemas.

---

9 These virtual character cells are shown here as if they were all the same size; in practice, however, that isn't the case when proportionally spaced fonts are in use.

10 For those who wish to type this sample using a Thai IME as described in the document "Exploring Complex Text Layout", the characters to type on an English keyboard are (C y o) , u l u j g] j , . The first person pronoun ฉัน is not required in a Thai phrase if no misunderstanding results, so that isn't shown in example β.

Implications of UTF-8 on Database Schema Design

A segment of the SQL statement used to create the sample bt_Person base table from Chapter 8 of "Business Database Triage"[11] is shown here:

```
CREATE TABLE bt_PERSON
 ( ID          NUMBER(4)       NOT NULL
 , Given_Name  VARCHAR2(12)    NOT NULL
 , Surname     VARCHAR2(12)    NOT NULL  . . . . continued
```

A number of sample entries were inserted into the `bt_Person` table for use in that chapter as well as in later chapters. Similarly, a variety of Company names and other sample data were provided in similar tables. Aside from providing some examples of how a database schema could be adapted to accommodate internationalization and localization, there was no discussion of column sizing changes that would result from reconfiguring the database to handle names and similar data in Scripts beyond Latin.

With the schema given above, the following statement fails, indicating that the values for the names exceed the column sizes, even when the database has been configured to handle Unicode characters:

```
INSERT INTO bt_PERSON (ID, Given_Name,   Surname,    . . . . continued)
             VALUES (93, 'ไดเอน์',       'โอเบอลี'     . . . . continued)
```

Using the Oracle RDBMS, the key portion of the error message is shown below:

```
ORA-12899: value too large for column ..."GIVEN_NAME" (actual: 18, maximum: 12)
```

In other words, we are attempting to stuff our six Thai characters into a column sized for twelve. The question is twelve *what*? For many years, the sizing (in this case 12) was referred to indiscriminately as 12 bytes or 12 characters, and such references can still be found today. To be fair, until relatively recently, a character always occupied one byte in any but some very specialized databases, but that habit must be broken. Today, with the advent of multi-lingual, multi-script databases, making that distinction is critical. In Oracle, the example we're using here, the size of `CHAR` and `VARCHAR2` fields in a UTF-8 (AL32UTF8 for recent versions) database is given in Bytes, not Characters.

Knowing that we are tasked with supporting Thai characters, and knowing that Thai characters occupy three bytes in their UTF-8 representation, we can make an initial guess that since Thai characters are three times the size of the Latin characters supported by our original design, changing the column specification to `VARCHAR2(36)` will eliminate the error. It might, but:

Thai names tend to be shorter than western names; perhaps we could get away with only a `VARCHAR2(32)`; again, maybe, but:

What was the basis of your organization's decision to size the `Surname` field as 12 bytes in the first place? Was some study of area telephone listings or even government-provided name lists used? Or, as we've often found, no one seems to know.[12]

Should a similar approach be undertaken for each new culture, language, or script that now needs to be supported?

And, what about other countries (Tibet? Japan?) with which you are now planning to do business? Even those using Scripts that occupy three bytes per character may differ widely in the average sizes of their Surnames – to say nothing of other language-specific data you need to accommodate in your database.

The `VARCHAR2` used above is a "variable width" data type. These differ from earlier "fixed width" data types to help reduce the size of stored data. Whereas earlier `CHAR(12)` types always allocated 12 bytes – regardless of the number of bytes stored, a variable width data type only used what was needed for each field (up to 12 of course).

This has led to recommendations from several sources that simply setting variable column sizes to their maximum permitted value will greatly simplify the tedious evaluation procedures described here.

This is an ignorant and dangerous recommendation on every level. While many details are beyond the scope of this document, consider all of the internal (stored procedures) and external (data entry forms and report writers) applications that allocate memory based on querying the data dictionary for the maximum size of the data elements they might expect to retrieve.

Imagine the greedy memory allocations made if only the `Given_Name` and `Surname` columns were specified as `VARCHAR2(4000)`. **Gulp!** Don't fall into this trap!

11 See information on page 13.

12 These organizations often have different sizing for employee names, customer names, and so forth. Hobgoblin or poor design?

Boundary Conditions and Column Sizing

Finally, when choosing reasonable upper limits for variable strings stored in your new multi-lingual, multi-script database and the applications that will utilize and support it, you need to be aware of which Scripts will require UTF-8 sizing larger than their actual Unicode values would suggest.

Listing all of the potential such boundary conditions would produce a large – and most likely irrelevant – list of scripts, and is therefore not attempted here. In *Three byte UTF-8 Representation – Details* on page 7 above, Devanagari and Thai scripts were mentioned as examples of those whose Unicode values could normally be represented with only two bytes in a straight decimal to binary conversion, but that require three bytes in their UTF-8 transformations.

BACK FROM UTF-8 TO UNICODE

An exploration of UTF-8 would not be complete without providing a few examples of how the UTF-8 sequences stored on disk are converted back to their actual Unicode Values for use in the system.

In the next Design Note of this series, *Evaluating Fonts for use in Multi-Lingual Documents*, a shell script is given to assist in such evaluations, and which optionally generates an XML (.fodt) version of a standard open document text (.odt) file listing fonts identified by that utility. Below is a segment of a resulting file we can use as an example:

```
<?xml version="1.0" encoding="UTF-8"?>
 < Sections between line above and line below have been removed for this example >
 office:version="1.2"
 office:mimetype="application/vnd.oasis.opendocument.text">
 <office:body>
  <office:text>
   <text:p >Now is the time for all good men to abandon CTL styles.</text:p>
   <text:p/>
   <text:p >นี่ คือ</text:p>
   <text:p >हिनदी भाषा</text:p>
   <text:p >שפת עברית</text:p>
   <text:p >This is some Hebrew text: שפת עברית and it's right-to-left.</text:p>
   <text:p/>
  </office:text>
 </office:body>
</office:document>
```

Looking at the bytes stored on disk in their UTF-8 transformation, the illustration on the right shows a "hex dump" of the segment beginning with `0x31` (the "1" character, `0x2E` (the period/ decimal point character), `0x32` (the "2" character), `0x22` (the ") and so forth.

This corresponds to the highlighted portion beginning with the last four characters in the third line of the file segment shown above. The `0x0A` in the fifth position is a Linux line ending; this will vary of course for other operating systems.

Here's how to read the Hex Dump:

```
00000884 31 2E 32 22 0A 20 6F 66 66 69 63 65 3A 6D 69 6D 65 74 79 70 1.2". office:mimetyp
00000898 65 3D 22 61 70 70 6C 69 63 61 74 69 6F 6E 2F 76 6E 64 2E 6F e="application/vnd.o
000008ac 61 73 69 73 2E 6F 70 65 6E 64 6F 63 75 6D 65 6E 74 2E 74 65 asis.opendocument.te
000008c0 78 74 22 3E 0A 20 3C 6F 66 66 69 63 65 3A 62 6F 64 79 3E 0A xt">. <office:body>.
000008d4 20 20 3C 6F 66 66 69 63 65 3A 74 65 78 74 3E 0A 20 20 20 3C   <office:text>.   <
000008e8 74 65 78 74 3A 70 20 3E 4E 6F 77 20 69 73 20 74 68 65 20 74 text:p >Now is the t
000008fc 69 6D 65 20 66 6F 72 20 61 6C 6C 20 67 6F 6F 64 20 6D 65 6E ime for all good men
00000910 20 74 6F 20 61 62 61 6E 64 6F 6E 20 43 54 4C 20 73 74 79 6C  to abandon CTL styl
00000924 65 73 2E 3C 2F 74 65 78 74 3A 70 3E 0A 20 20 20 3C 74 65 78 es.</text:p>.   <tex
00000938 74 3A 70 2F 3E 0A 20 20 20 3C 74 65 78 74 3A 70 20 3E E0 B8 t:p/>.   <text:p >..
0000094c 99 E0 B8 B5 E0 B9 89 20 E0 B8 84 E0 B8 B7 E0 B8 AD 3C 2F 74 ....... .........</t
00000960 65 78 74 3A 70 3E 0A 20 20 20 3C 74 65 78 74 3A 70 20 3E E0 ext:p>.   <text:p >.
00000974 A4 B9 E0 A4 BF E0 A4 A8 E0 A4 A6 E0 A5 80 20 E0 A4 AD E0 A4 ............... .....
00000988 BE E0 A4 B7 E0 A4 BE 3C 2F 74 65 78 74 3A 70 3E 0A 20 20 20 .......</text:p>.
0000099c 3C 74 65 78 74 3A 70 20 3E D7 A9 D7 A4 D7 AA 20 D7 A2 D7 9B <text:p >...... ....
000009b0 D7 A8 D7 99 3C 2F 74 65 78 74 3A 70 3E 0A 20 20 20 3C 74 65 ....</text:p>.   <te
000009c4 78 74 3A 70 20 3E 54 68 69 73 20 69 73 20 73 6F 6D 65 20 48 xt:p >This is some H
000009d8 65 62 72 65 77 20 74 65 78 74 3A 20 D7 A9 D7 A4 D7 AA 20 D7 ebrew text: ...... .
000009ec A2 D7 9B D7 A8 D7 99 20 61 6E 64 20 69 74 27 73 20 72 69 67 ....... and it's rig
00000a00 68 74 2D 74 6F 2D 6C 65 66 74 2E 3C 2F 74 65 78 74 3A 70 3E ht-to-left.</text:p>
00000a14 0A 20 20 20 3C 74 65 78 74 3A 70 2F 3E 0A 20 20 3C 2F 6F 66 .   <text:p/>.  </of
00000a28 66 69 63 65 3A 74 65 78 74 3E 0A 20 3C 2F 6F 66 66 69 63 65 fice:text>. </office
00000a3c 3A 62 6F 64 79 3E 0A 3C 2F 6F 66 66 69 63 65 3A 64 6F 63 75 :body>.</office:docu
00000a50 6D 65 6E 74 3E 0A                                           ment>.
```

Hex Dump segment of the Open Document Text Sample above

We'll begin with byte `0x94A`, the `0xE0` value highlighted in the tenth display row containing bytes `0x938` to `0x94B`. Since Hex Dump utilities[13] simply display bytes, and neither know nor care about encoding schemes such as UTF-8, there is no corresponding character shown for that position on the right panel of the display. Any byte outside the limits of "lower ASCII" is simply represented as a period (or dot, or decimal point – take your pick).

The hexadecimal `E` character is a binary `1110` and, knowing (as you now should) that this byte represents the start of a three byte UTF-8 character sequence, you realize that you should interpret the sequence `0xE0B899` as *one character*.

In the three tables below, the different script segments (Thai, Hindi, and Hebrew) are expanded to show each individual byte (in hexadecimal) on the first row. Their UTF-8 binary representation is shown in the second row as it is stored in memory and on disk.

Row three in each example shows the portions of the UTF-8 bit stream representing the actual Unicode values, while the fourth row arranges them in their normal single or double byte notation. Row 5 shows the Unicode values represented by those bytes, while the sixth shows the Latin keys used to enter these characters with Input Method Editors as described in the previous Design Note *Exploring Complex Text Layout*.

Thai (Unicode Plane u+0E01-0E7F) Text Sample

The nineteen bytes (four character cells in green) of the Thai text segment นี้ คือ beginning at `0x094A` are:

E0 B8 99	**E0 B8 B5**	**E0 B9 89**	**20**
11100000 10111000 10011001	11100000 10111000 10110101	11100000 10111001 10001001	00100000
0000 111000 011001	0000 111000 110101	0000 111001 001001	0100000
00001110 00011001	00001110 00110101	00001110 01001001	00100000
u+0E19 (3609)	u+0E35 (3637)	u+0E49 (3657)	u+20(32)
o = น	u = ี	h = ้	space
E0 B8 84	**E0 B8 B7**	**E0 B8 AD**	
11100000 10111000 10000100	11100000 10111000 10110111	11100000 10111000 10101101	
0000 111000 000100	0000 111000 110111	0000 111000 101101	
00001110 00000100	00001110 00110111	00001110 00101101	
u+0E04 (3588)	u+0e37 (3639)	u+0e2D(3629)	
8 = ค	n = ื	v = อ	

In UTF-8 format, any byte beginning with a '1' is part of a multi-byte character. Two or more leading '1' bits indicate the first byte of such a group; any byte beginning with '10' is a continuation byte. In the first byte, the number of leading '1' bits indicate the total number of bytes in the character.[14]

Thus, when packaged as a UTF-8 stream in memory or on disk – shown as a hex dump in line one and as the equivalent twenty-four bits in line two – the sixteen bit Thai Unicode character น (`U+0E19`) is extracted from the UTF-8 representation into groups of 4, 6, and 6 bits as shown in the third line.

What is significant to note in this example is that the Thai base character น and its two diacritics (a vowel and a tone mark) are stored separately in memory *as they were entered*. Thus, it is only on screen or in print that we see them occupying a single character cell as the composite นี้. Thai script is therefore considered to be 'complex.' The reality, however, is that this situation is really no different than typing a French word having the letter á, except for the fact that, rather than storing the letter 'a' (`U+0061`) and the acute accent ´ (`U+00B4`) separately, they are converted on input to the single á (`U+00E1`) character.

Note also in this Thai sample that the fourth character cell consists of only one byte (the tenth) and is an example of how many "alphabets" (or the "modified Abjad" in this case) share non-alphabetical characters with the Latin script.

13 This particular view was made using the Linux Bless utility, but any other hex viewer or editor should suffice.

14 This scheme facilitates efficient determination of cursor movements as well as deleting and backspacing, which users expect to work on character cells, as opposed to individual elements. Applications permit navigating those elements in a variety of ways.

Hindi (Devanagari Unicode Plane u+0900-097F) Text Sample

The initial sixteen bytes (of twenty-eight) of the Hindi text segment हिनदी beginning at `0x0973` are:

E0 A4 B9	E0 A4 BF	E0 A4 A8
11100000 10100100 10111001	11100000 10100100 10111111	11100000 10100100 10101000
0000 100100 111001	0000 100100 111111	0000 100100 101000
00001001 00111001	00001001 00111111	00001001 00101000
u+0939 (2361)	u+093F (2367)	u+0928 (2344)
h = ह	i = ि	n = न

E0 A4 A6	E0 A5 80	20	
11100000 10100100 10100110	11100000 10100101 10000000	00100000	*Sixteen Bytes displayed*
0000 100100 100110	0000 100101 000000	0100000	*in Six Character Cells*
00001001 00100110	00001001 01000000	00100000	
u+0926 (2342)	u+0940 (2368)	u+20 (32)	
d = द	I = ी	space	

The Devanagari script also uses three bytes to represent each character in UTF-8. Of interest here is that, while the stored order in memory is the same as the entry order, the rendering order of the first two characters is exchanged, in this case because the ि vowel always precedes its associated consonant.[15]

Hebrew (Unicode Plane u+0590-05FF) Text Sample

The initial fifteen bytes (eight character cells) of the Hebrew segment שפת עברית beginning at `0x09A5` are:

D7 A9	D7 A4	D7 AA	20	
11010111 10101001	11010111 10100100	11010111 10101010	00100000	*Fifteen Bytes displayed*
10111 101001	10111 100100	10111 101010	0100000	*in Eight Character Cells*
00000101 11101001	00000101 11100100	00000101 11101010	00100000	
u+05E9 (1513)	u+05E4 (1508)	u+05EA (1514)	u+20 (32)	
a = ש	p = פ	, = ת	space	

D7 A2	D7 9B	D7 A8	D7 99	D7 AA
11010111 10100010	11010111 10011011	11010111 10101000	11010111 10011001	11010111 10101010
10111 100010	10111 011011	10111 101000	10111 011001	10111 101010
00000101 11100010	00000101 11011011	00000101 11101000	00000101 11011001	00000101 11101010
u+05E2 (1506)	u+05DB (1499)	u+05E8 (1512)	u+05D9 (1497)	u+05EA (1514)
g = ע	f = כ	r = ר	h = י	, = ת

Although the Hebrew characters are stored sequentially as they are entered, they are displayed and printed in right-to-left order as would be expected with any Unicode blocks in the `u+0590-08FF` range.

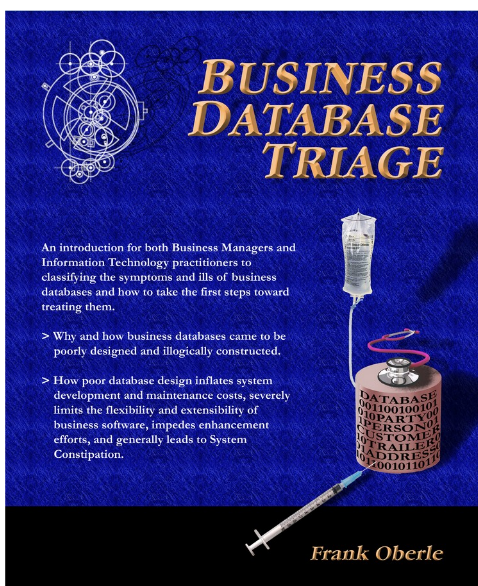
The Bottom Line: with very few exceptions, although LibreOffice needs an awareness of script and language related differences, it doesn't need to care about the arbitrary and obsolete 'Asian' and 'Complex' classifications currently responsible for many of its quirks. The Input Method handled all these examples.

This concludes the introduction to the Unicode UTF-8 transformation format. The next (fourth) Design Note in this series is "Evaluating Fonts for use in Multi-Lingual Documents." While fonts may not obviously seem to be the concern of database custodians, the very broad interpretation of "Documents" exposes many ways that data can be lost or distorted. As a database designer/maintainer, you need to be able to recognize and have an idea how to react to some common symptoms that crop up in the migration to your new multi-lingual, multi-script database.

15 … as explained in the earlier document in this series "Exploring Complex Text Layout" on page 25.

Other Publications

Antikythera Publications



More information and sample pages at:
www.AntikytheraPubs.com

In addition to an ongoing series of Database Design Notes, Antikythera Publications recently released the book "*Business Database Triage*" (ISBN-10: 0615916937) that demonstrates how commonly encountered business database designs often cause significant, although largely unrecognized, difficulties with the development and maintenance of application software. Examples in the book illustrate how some typical database designs impede the ability of software developers to respond to new business opportunities – a key requirement of most businesses.

A number of examples of solutions to curing business system constipation are presented. Urban legends, such as the so-called object-relational impedance mismatch, are debunked – shown to be based mostly on illogical database (and sometimes object) designs.

"*Business Database Triage*" is available through major book retailers in most countries, or from the following on-line vendors, each of which has a full description of the book on their site:

CreateSpace: https://www.createspace.com/4513537

Amazon:

www.amazon.com/Business-Database-Triage-Frank-Oberle/dp/0615916937