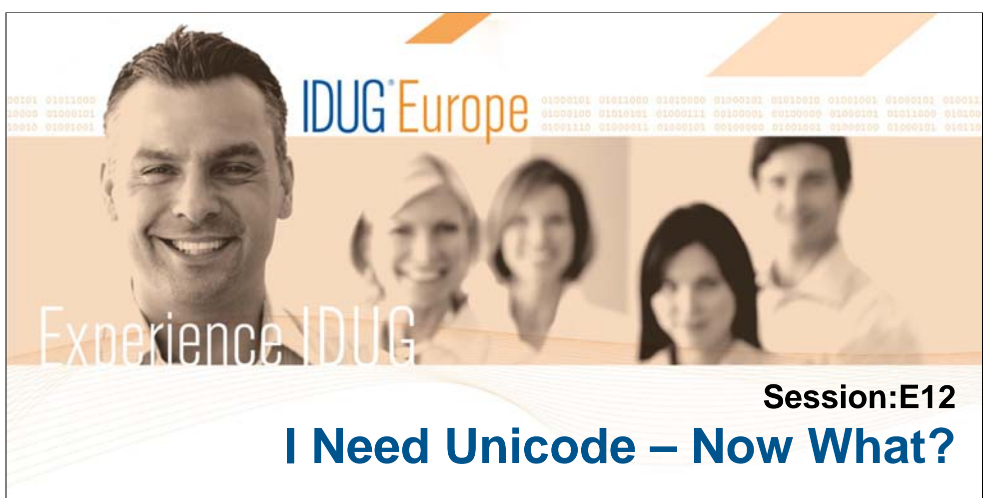


Session:E12

I Need Unicode – Now What?

Christopher J. Crone
IBM – DB2 for z/OS Development

Wednesday October 7th •2:15-3:15 PM
Platform: DB2 for z/OS

Presentation Topics

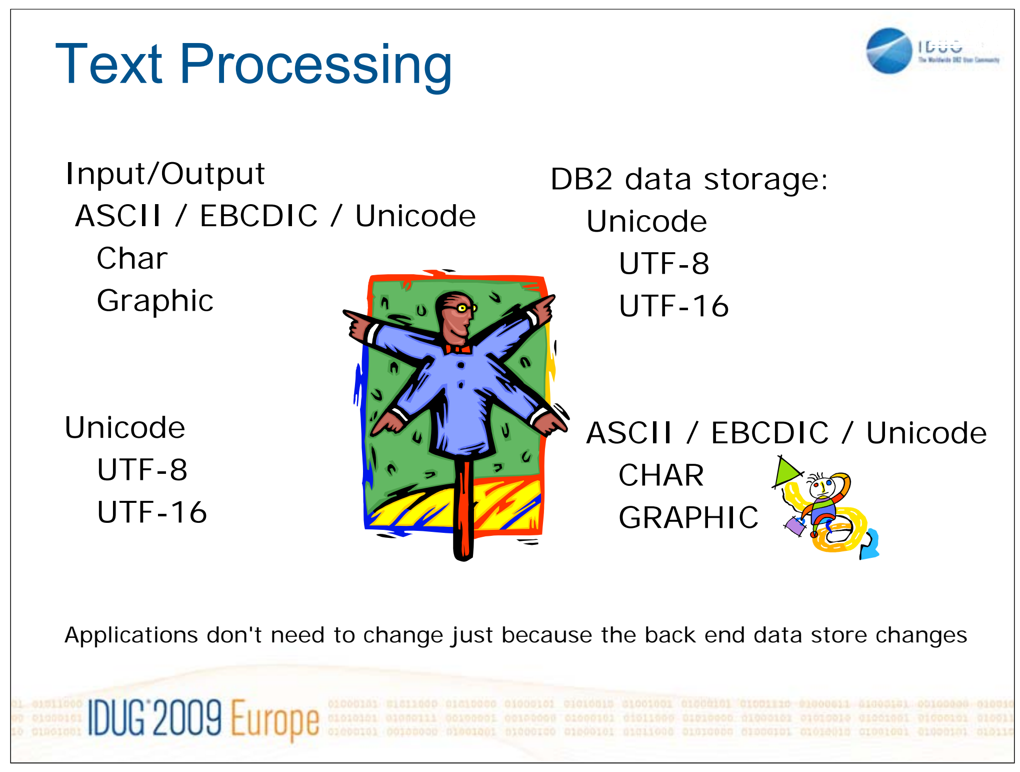
- DB2 Unicode Capabilities
- Common pitfalls.
- Application Support for Unicode
  - COBOL, PL/I, JAVA, ODBC, and other Distributed Access
- When to use UTF-8/UTF-16
- A basic to approach a Unicode project.

DB2 Unicode Capabilities

How is Unicode data stored?

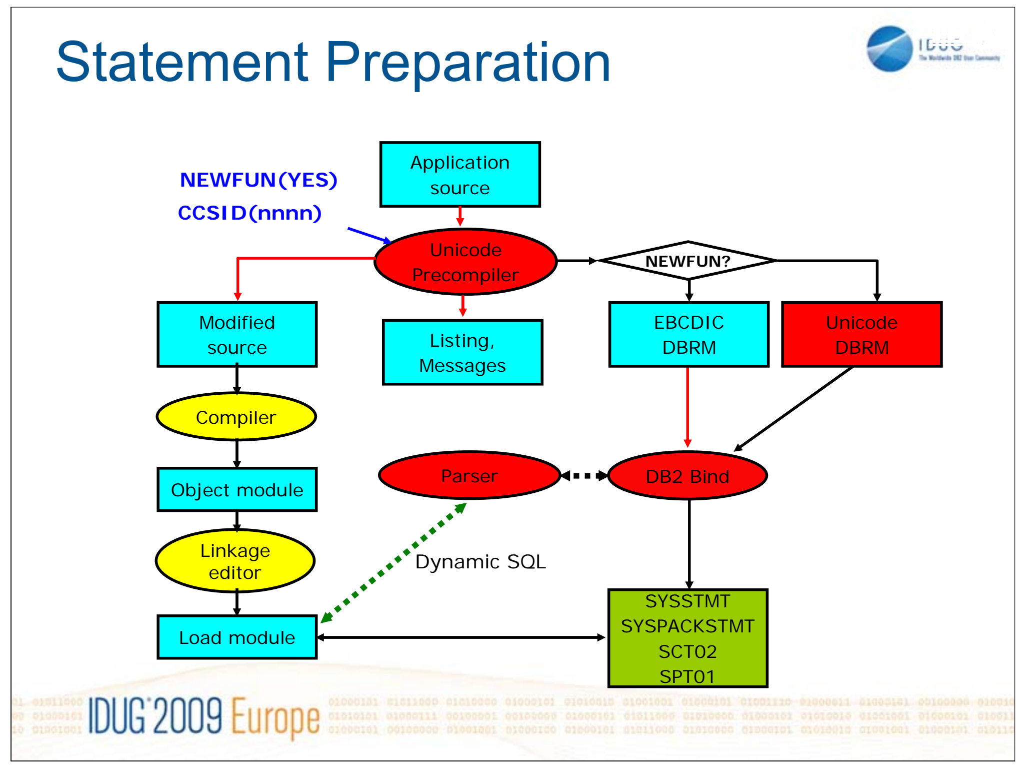
- Storage of Unicode Data
  - Char/VarChar/CLOB FOR SBCS DATA
    - (7-bit) ASCII this is a subset of UTF-8 CCSID 367
  - Char/VarChar FOR BIT DATA
    - CCSID 65535 – No conversion by DB2, padding may occur
  - Char/VarChar/CLOB [FOR MIXED DATA]
    - UTF-8 CCSID 1208
  - Graphic/VarGraphic/DBCLOB
    - UTF-16 CCSID 1200
  - Binary/VarBinary/BLOB
    - No CCSID, No Conversion, No padding

Data stored in Unicode tables in DB2 will be stored in one of the following CCSIDs
-CCSID 367 is used to store data in columns defined as FOR SBCS DATA.
    -CCISD 367 is a (7 bit ASCII CCSID) that is a subset of UTF-8
-CCSID 65535 data is not subject to conversion, but may be padded
-CCSID 1208 - Unicode UTF-8 - is used by default, or when FOR MIXED DATA is specified
-CCSID 1200 – Unicode UTF-16 – is used when graphic columns are defined in Unicode tables
-BLOB data has no CCSID, is not subject to conversion, and will not be padded by DB2.



When dealing with Unicode tables, we have torn down the barrier between CHAR and GRAPHIC.

This means your back end data store can be either UTF-8 or UTF-16 and you can use ASCII, EBCDIC, or Unicode character or graphic host variables and DB2 will perform the necessary conversions to/from the CCSID of the host variable even if the host variable doesn't match the column type (for ASCII and EBCDIC back end data stores, in most cases char and graphic are incompatible).
Note: for V7, Graphic Unicode host variables are incompatible with ASCII / EBCDIC SBCS tables. V8 allows this combination.



CCSID(nnnn) input parameter to the Unicode Precompiler specifies the CCSID of the application source to ensure proper conversion to unicode for processing. The default value of the CCSID option is the EBCDIC system CCSID as specified on the panel DSNTIPF during installation.

The modified source program (an output of precompilation) remains in its original CCSID. If the DBRM is later bound to a server that does not support UTF-8, the SQL statements are then converted from CCSID 1208 (UTF-8) and sent in the EBCDIC system CCSID.

- NEWFUN(YES)
  - Accept V8 new syntax
  - Unicode DBRM
- NEWFUN(NO)
  - Reject V8 new syntax
  - EBCDIC DBRM
- CCSID(nnnn)
  - It is important to note that this CCSID applies to the source of the application program. It

Controlling Encoding in the App

- DECLARE VARIABLE statement
- Mechanism to allow CCSID to be specified for host variables
- Example
  - EXEC SQL DECLARE :hv1 CCSID UNICODE;
  - EXEC SQL DECLARE :hv2 CCSID 37;
- Directive to specify hostvar CCSID
- Useful for PREPARE / EXECUTE IMMEDIATE statement text
  - EXEC SQL PREPARE S1 FROM :hv2;
- May be used with any string host variable on input or output

The DECLARE VARIABLE statement, added in DB2 V7, can be used to specify the CCSID of a particular host variable.

This is a precompiler directive that causes the precompiler to specify the CCSID of the host variable in any SQLDA that the precompiler generates to reference the host variable. This directive works for both input and output host variables.

Controlling Encoding at BIND*

- Application Encoding Scheme - ENCODING
  - System Default
    - Determines Encoding Scheme when none is explicitly specified
  - Bind Option
    - Allows explicit specification of ES at an application level. Affects Static SQL - Provides default for dynamic
    - System Default used if bind option not specified
  - Special Register
    - Allows explicit specification of ES at the application level. Affects Dynamic SQL
    - Initialized with value from ENCODING Bind Option
  - DRDA
    - OPTION is ignored (for CCSID info) when packages are executed remotely
    - DRDA specified Input CCSID, Data flows as is to client
  - Used in SET and Multiple CCSID statements (V8 and above)

* Be careful specifying ENCODING(UNICODE)

Also new to DB2 V7 is the specification of Application Encoding Scheme.

This allows a default Application Encoding to be specified
Preset to EBCDIC
The Application Encoding Scheme can also be specified on BIND PLAN or PACKAGE
If not specified, the system default value is used for the bind option

Plans/Packages bound prior to V7 are assumed to be EBCDIC

The option applies to Static SQL
The Application Encoding Scheme special register can be used to affect dynamic SQL
Initial value is the value of the Bind Option.
The SET statement, in V7 and V8, is processed using the ENCODING bind option
In V8, the ENCODING bind option (and special register for dynamic statements) also affects processing of multiple CCSID statements.

Multiple CCSIDs – Example

```
SELECT  a.name, a.creator, b.charcol, 'ABC',
                 :hvchar, X'C1C2C3'
FROM sysibm.systables a,
          ebcdictable b
WHERE a.name = b.name AND
             b.name > 'B' AND
             a.creator = 'SYSADM'
ORDER BY b.name;
```

In V7 and below, since both tables have the same system EBCDIC CCSID set, the comparisons are done in EBCDIC and the result data is EBCDIC.

To maintain compatibility with previous releases, statements that do not reference objects with more than one CCSID set, will continue to use the old rules.

- For compatibility with prior releases:
  - IF an SQL statement which references table objects with **only one CCSID set**
  - THEN **the results** will continue to have the same **result encoding scheme (CCSID set) as the table objects** and basic semantics.
- All string objects and special registers in the SQL statement are converted to this result encoding scheme (CCSID set) vs. using the application encoding scheme like multiple CCSID set SQL statements.

In this example, all the tables reference a single encoding scheme.

The results of this query will be EBCDIC.

Multiple CCSIDs – Example (cont)

```
SELECT  a.name, a.creator, b.charcol, 'ABC',
                    :hvchar, X'C1C2C3'
FROM sysibm.systables a,
          ebcdictable b
WHERE a.name = b.name AND
          b.name > 'B' AND
          a.creator = 'SYSADM'
ORDER BY b.name;
```

Result or Evaluated:
EBCDIC
Unicode
Application Encoding Scheme

Assuming a Unicode catalog, the result will contain multiple CCSIDs and the comparisons and ordering will be dependent on the context.

In this example, where we have a Unicode catalog, some columns are EBCDIC and others are Unicode. Literal values are interpreted using the Application Encoding option in effect.
Queries that cross encoding schemes are allowed once ENFM (Enabling New Function Mode) is entered. This is to ensure that applications that access the catalog continue to function.

Statements with Multiple CCSID sets

- Comparison and resulting data types for multiple CCSID sets...
  - If an expression or comparison involves two strings which contain columns with different CCSID sets,
    - Drive to Unicode if necessary
    - WHERE T1.C1 = T2.C1
  - If an expression or comparison involves two strings with different CCSID sets where only one of them contains a column,
    - Drive to the column's CCSID set
    - WHERE T1.C1 = X'C1C2'
  - If an expression or comparison involves two strings with different CCSID sets and neither contains a column,
    - Drive to Unicode
    - WHERE GX'42C142C2' = 'ABC' -- GX literal and 'ABC' are different CCSIDs
  - String constants and special registers in a context by themselves use the application encoding scheme
    - SELECT 'ABC' FROM T1 . . .

Here are some examples of how the new DB2 rules for multiple CCSID sets work

-In the first case we have two columns being compared – in this case, drive to Unicode if they are not already Unicode

-In the second case, we drive to the column CCSID set

-In the third case, neither side is a column, so drive to Unicode

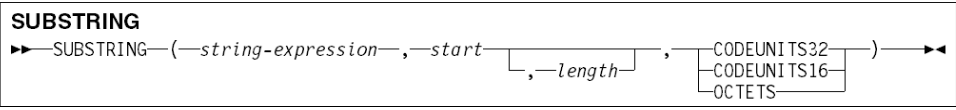
-In the fourth case we use the APPLICATION ENCODING SCHEME – the thought here is that the application is providing the string, we should interpret the string in the same encoding as the application.

Character Based Functions

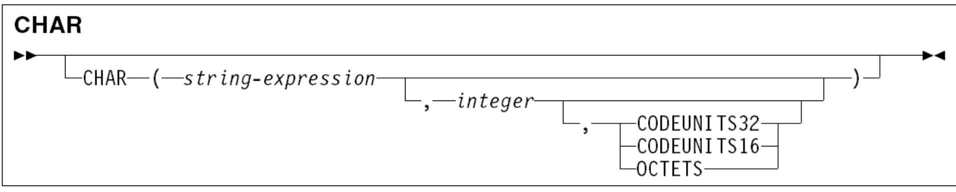
- New functions
  - CHARACTER_LENGTH
  - POSITION
  - SUBSTRING
- Updated Functions
  - CHAR
  - CLOB
  - DBCLOB
  - GRAPHIC
  - INSERT
  - LEFT
  - LOCATE
  - RIGHT
  - VARCHAR
  - VARGRAPHIC
- CAST Specification
  - Changes to enable specification of Code Units

Character Based Functions (sample syntax)

SUBSTRING



CHAR



Character Based Functions - Example

Assume that NAME is a VARCHAR(128) column, encoded in Unicode UTF-8, that contains 'Jürgen'. The following query:

```
SELECT CHARACTER_LENGTH(NAME,CODEUNITS32)
FROM T1 WHERE NAME = 'Jürgen';
```

or

```
SELECT CHARACTER_LENGTH(NAME,CODEUNITS16)
FROM T1 WHERE NAME = 'Jürgen';
```

returns the value 6. A similar query:

```
SELECT CHARACTER_LENGTH(NAME,OCTETS)
FROM T1 WHERE NAME = 'Jürgen';
```

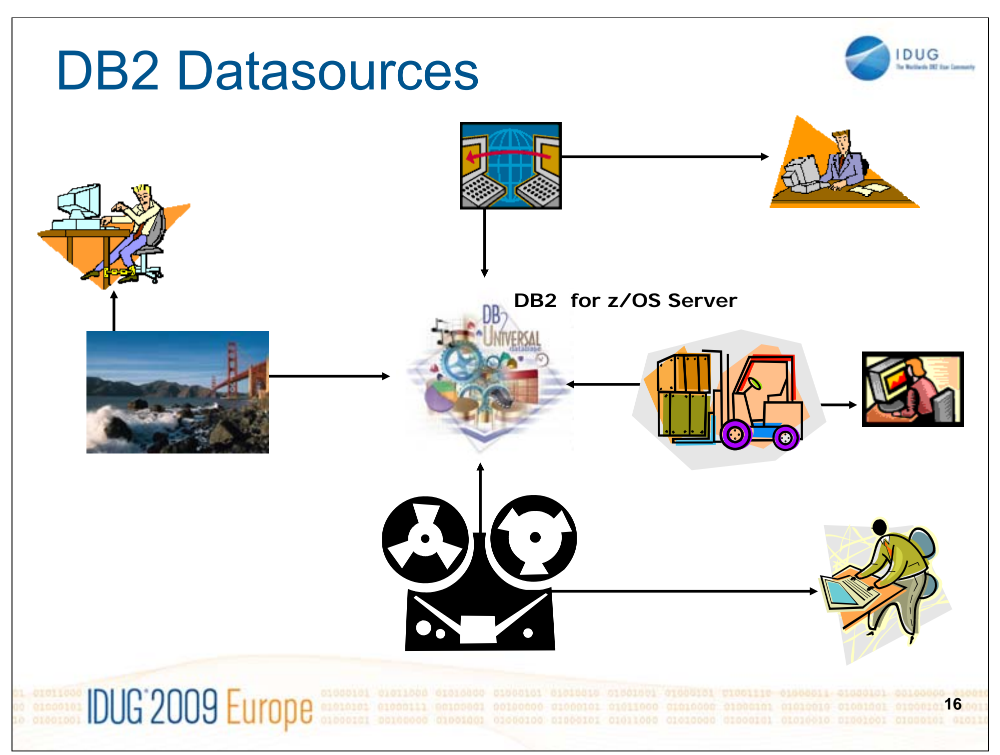
or

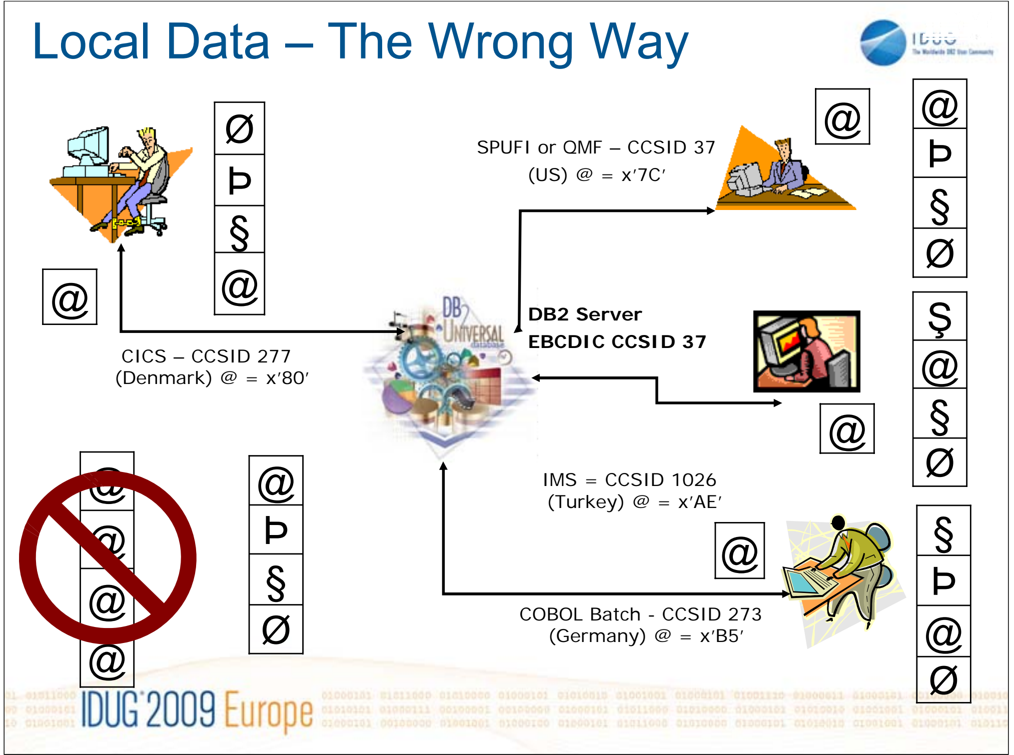
```
SELECT LENGTH(NAME)
FROM T1 WHERE NAME = 'Jürgen';
```

returns the value 7.

Name	UTF-8 Representation	UTF-16 Representation	UTF-32 Representation
Jürgen	x'4AC3BC7267656E'	x'004A00FC007200670065006E'	x'0000004A000000FC0000007200000067000000650000006E'

Common Pitfalls


DB2 Datasources
IDUG
DB2 for z/OS Server
IDUG 2009 Europe
16



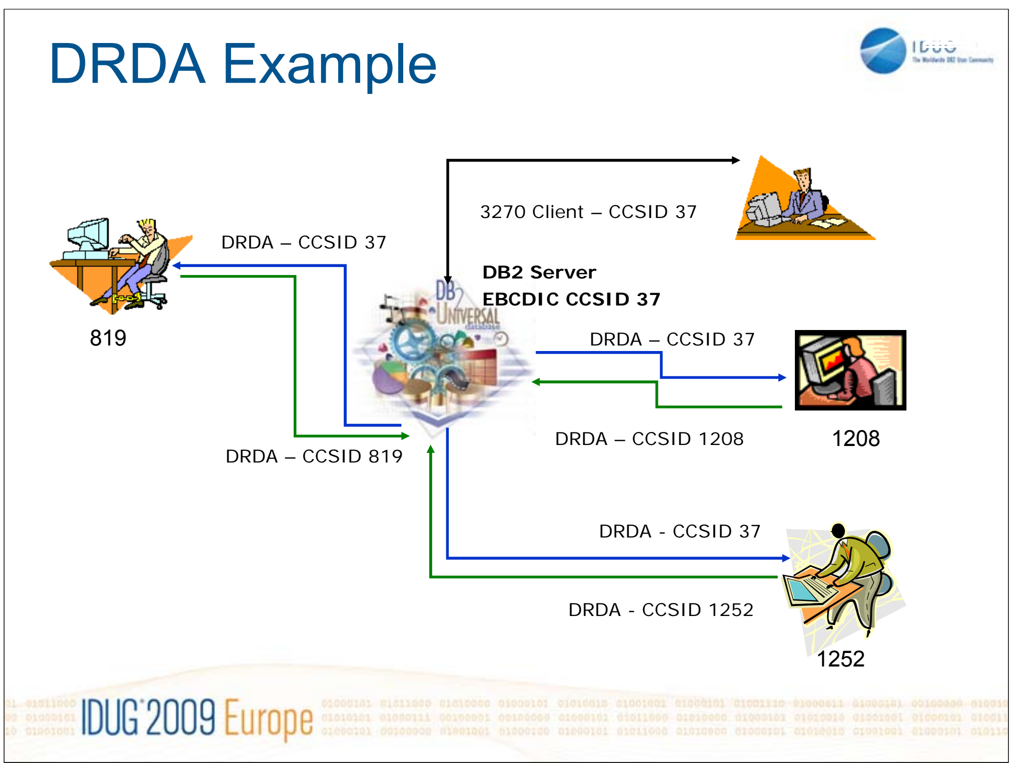
In this example, assume that all the clients are referencing an EBCDIC table (CCSID 37 in this case). Each of the local applications is attempting to insert an "@" into a DB2 table. In general, the results will be incorrect unless DB2 "knows" that the data coming from the CICS, IMS, and COBOL applications is not CCSID 37. There are various ways of the application to tell DB2 the CCSID of the data that they are giving DB2 (in input host variables) or they want to receive from DB2 (in output host variables). These are:

-CCSID overrides in the SQLDA – when using a descriptor

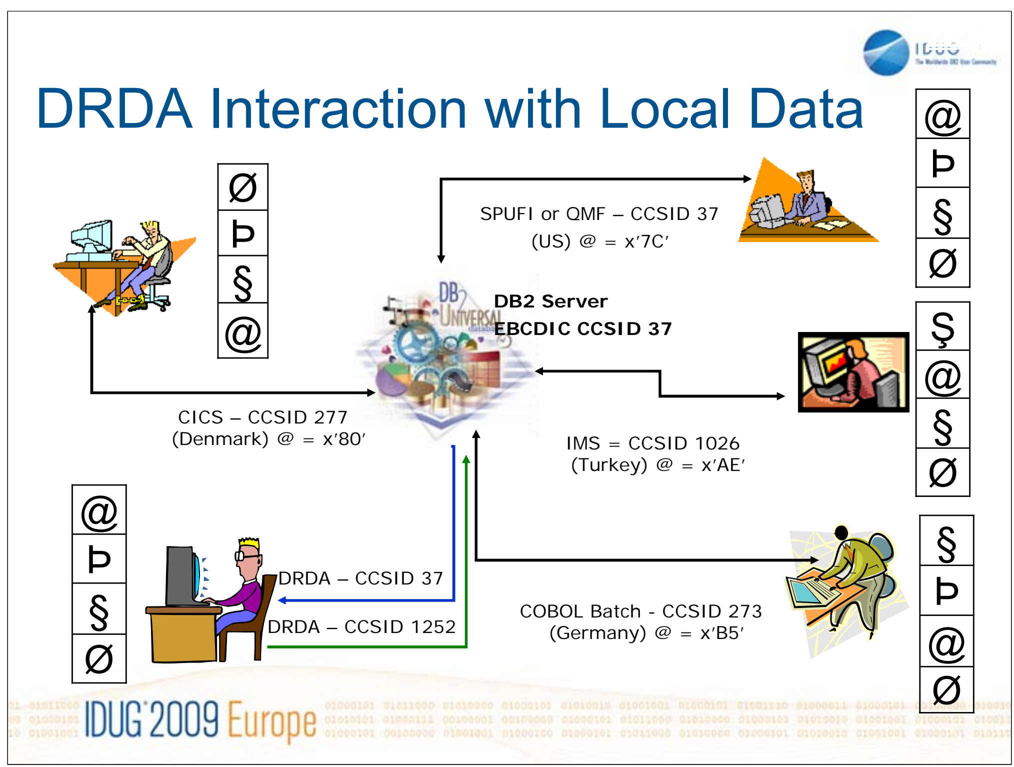
-DECLARE VARIABE – precompiler directive

-ENCODING bind option

-APPLICATION ENCODNG special register



In this example, assume that all the clients are referencing an EBCDIC table (CCSID 37 in this case) the green lines represent data flowing from the client to the server. The server is responsible for converting the data to CCSID 37 to store in the table. The blue lines represent data flows from the server to the clients. The clients are responsible for converting the data from CCSID 37 back to the CCSID they need. The black line represents a local 3270 client – no conversion is necessary in this case.



In this example, assume that all the clients are referencing an EBCDIC table (CCSID 37 in this case). Each of the local applications is attempting to insert an "@" into a DB2 table. In general, the results will be incorrect unless DB2 "knows" that the data coming from the CICS, IMS, and COBOL applications is not CCSID 37.

This example demonstrates that remote clients see what a correctly configured local client will see. This is because the EBCDIC data that is sent to the DRDA client is tagged as CCSID 37. When the data is converted from EBCDIC to ASCII CCSID 37 is used as the source of the data. DB2 has no way to tell that some of the data may in fact not be CCSID 37.

Application Support for Unicode

COBOL

Enterprise COBOL V3R1+ Supports Unicode

- NATIONAL is used to declare UTF-16 variables
  - MY-UNISTR pic N(10). -- declares a UTF-16 Variable
- N and NX Literals
  - N'123'
  - NX'003100320033'
- Conversions
  - NATIONAL-OF Converts to UTF-16
  - DISPLAY-OF Converts to specific CCSID
    - DECLARE Greek-EBCDIC pic X(10) value " Ξ Σ Φ Λ Θ Ζ Δ Γ Ω ".
    - UTF16STR pic N(10).
    - UTF8STR pic X(20).
    - Move Function National-of(Greek-EBCDIC, 00875) to UTF16STR.
    - Move Function Display-of(UTF16STR, 01208) to UTF8STR.

Cobol has recently added support for Unicode characters

Included in this support

-New NATIONAL data type

-N and NX literals

-Conversion operations

COBOL & DB2 (getting DB2 to convert for you)

```
EXEC SQL BEGIN DECLARE SECTION;
   01 HOST-VARS.
         05 GREEK-EBCDIC PIC X(10) VALUE "Ξ Σ Φ Λ Θ Ζ Δ Γ Ω".
         05 UTF16STR PIC N(10).*
         05 UTF8STR PIC X(20).
   EXEC SQL DECLARE :UTF8STR VARIABLE CCSID 1208.
EXEC SQL END DECLARE SECTION;

INSERT INTO T1 (C1) VALUES(:GREEK-EBCDIC) END EXEC.**

EXEC SQL
  SELECT C1, C1 INTO :UTF16STR, :UTF8STR
END EXEC.
```

* COBOL will use an implicit DECLARE VARIABLE for PIC N data.

** This example assumes T1 is a table encoded in EBCDIC CCSID 875 and that the ENCODING bind option for this appl is also CCSID 875.

Using the previous example as a starting point, it is also possible to get DB2 to perform the needed conversions for you.

In this example, the EBCDIC 875 data is INSERTed into an EBCDIC CCSID 875 table. Then the COBOL application fetches the data into the UTF-16 and UTF-8 host variables.

Note that only one DECLARE VARIABLE statement is used needed because PIC N data is implicitly UTF-16, and the PIC X data is assumed to be in the CCSID specified by the ENCODING bind option.

COBOL - *national decimal*

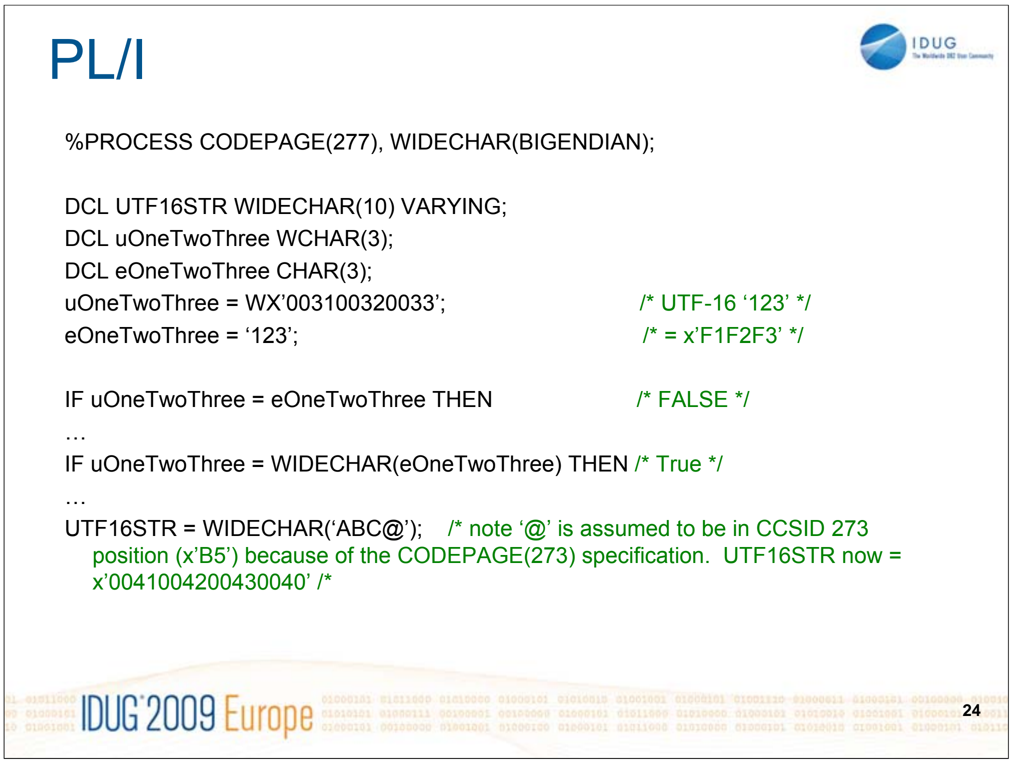
- National Decimal is the Unicode format of Zoned Decimal

```
05 Count-n Pic 9(4) Value 25 Usage National
```

- National Groups *

```
01 Alpha-Group-1.
    02 Group-1.
        04 Month PIC 99.
        04 DayOf PIC 99.
        04 Year PIC 9999.
    02 Group-2 GROUP-USAGE NATIONAL.
        04 Amount PIC 9(4).99.
```

Can be subordinated under an alpha group, but alpha groups cannot be subordinated under a national group.



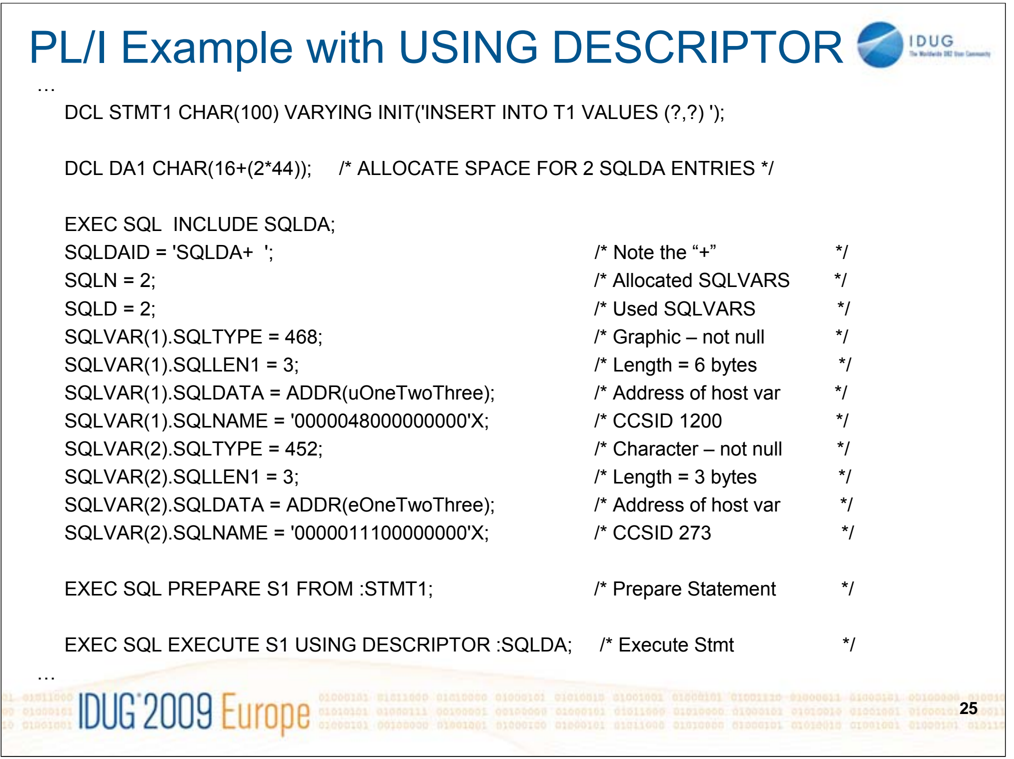
PL/I has some support for Unicode UTF-16 data.

PL/I uses the WIDECHAR datatype to support UTF-16.

PL/I supports UTF-16 data as either BIG or LITTLE ENDIAN. For compatibility with DB2, BIGENDIAN data should be used.

PL/I has basic conversion support via the WIDECHAR function. This function does not use the z/OS conversion services at this time.

PL/I does not have any support for a UTF-16 literal, but does offer a WX literal which can be used to specify a UTF-16 codepoint (like the UX constant would be used in DB2).



This code snippet shows the code needed to use a descriptor (SQLDA) with a dynamic INSERT statement.

This is one example of how to perform this sort of programming and is intended to demonstrate the techniques associated with using a descriptor and setting CCSID values with a dynamic INSERT statement.

COPROCESSOR – CCSID options

- COBOL
  - SQLCCSID (default)
  - NOSQLCCSID
- PL/I
  - CCSID0 (default)
  - NOCCSID0

Precompiler or Coprocessor?

- COBOL
  - COPROCESSOR is required for Unicode apps
- PL/I or C
  - Coprocessor is recommended for Unciode apps
- Long term direction of DB2 is the Coprocessor

Conversion Considerations – Application -vs- DB2

- In many cases, conversion can be handled by the application or DB2 (as shown in previous slides).

- Conversion by DB2
  - Pro
    - Less application code
  - Con
    - Performance impact may be greater than just conversion cost

- Conversion by Application
  - Pro
    - May perform better in some cases
  - Con
    - Requires more application code

DCLGEN

DBCSSYMBOL

Specifies the symbol used to denote a graphic data type in a COBOL PICTURE clause.

(G) Graphic data is denoted using G.

(N) Graphic data is denoted using N.

DCLBIT

Specifies if DCLGEN should be sensitive to the declaration of FOR BIT DATA items

(NO) – backward compatible behavior

(YES) – Causes DCLGEN to create a DECLARE VARIABLE statement for columns in DB2 that were declared with the FOR BIT DATA clause

DCLGEN Example (wrong)

```
******************************************************************
* DCLGEN TABLE(ADMF001.T1)                                       *
*        LIBRARY(USER.DBRMLIB.DATA(T4))                          *
*        LANGUAGE(COBOL)                                         *
*        QUOTE                                                   *
* ... IS THE DCLGEN COMMAND THAT MADE THE FOLLOWING STATEMENTS   *
******************************************************************
     EXEC SQL DECLARE ADMF001.T1 TABLE
     ( NAME                            VARGRAPHIC(15),
       ADDRESS                         VARGRAPHIC(25),
       CITY                            VARGRAPHIC(20),
       STATE                           GRAPHIC(2),
       ZIP                             GRAPHIC(5),
       PASSWORD                        CHAR(8)
     ) END-EXEC.
******************************************************************
* COBOL DECLARATION FOR TABLE ADMF001.T1                         *
******************************************************************
 01  DCLT1.
     10 NAME.
        49 NAME-LEN           PIC S9(4) USAGE COMP.
        49 NAME-TEXT          PIC G(15) USAGE DISPLAY-1.
     10 ADDRESS.
        49 ADDRESS-LEN        PIC S9(4) USAGE COMP.
        49 ADDRESS-TEXT       PIC G(25) USAGE DISPLAY-1.
     10 CITY.
        49 CITY-LEN           PIC S9(4) USAGE COMP.
        49 CITY-TEXT          PIC G(20) USAGE DISPLAY-1.
     10 STATE                 PIC G(2) USAGE DISPLAY-1.
     10 ZIP                   PIC G(5) USAGE DISPLAY-1.
     10 PASSWORD              PIC X(8).
******************************************************************
* THE NUMBER OF COLUMNS DESCRIBED BY THIS DECLARATION IS 6       *
******************************************************************
```

DCLGEN Example (right)

```
******************************************************************
* DCLGEN TABLE(ADMF001.T1)                                       *
*        LIBRARY(USER.DBRMLIB.DATA(T1))                          *
*        LANGUAGE(COBOL)                                         *
*        QUOTE                                                   *
*        DBCSSYMBOL(N)                                           *
*        DCLBIT(YES)                                             *
* ... IS THE DCLGEN COMMAND THAT MADE THE FOLLOWING STATEMENTS   *
******************************************************************
     EXEC SQL DECLARE ADMF001.T1 TABLE
     ( NAME                           VARGRAPHIC(15),
       ADDRESS                        VARGRAPHIC(25),
       …
       PASSWORD                       CHAR(8)
     ) END-EXEC.
******************************************************************
* DECLARED VARIABLES FOR 'FOR BIT DATA' COLUMNS                  *
******************************************************************
     EXEC SQL DECLARE
      :PASSWORD
     VARIABLE FOR BIT DATA END-EXEC.
******************************************************************
* COBOL DECLARATION FOR TABLE ADMF001.T1                         *
******************************************************************
 01  DCLT1.
     10 NAME.
        49 NAME-LEN          PIC S9(4) USAGE COMP.
        49 NAME-TEXT         PIC N(15).
     10 ADDRESS.
        49 ADDRESS-LEN       PIC S9(4) USAGE COMP.
        49 ADDRESS-TEXT      PIC N(25).
     10 CITY.
        49 CITY-LEN          PIC S9(4) USAGE COMP.
        49 CITY-TEXT         PIC N(20).
     10 STATE                PIC N(2).
     10 ZIP                  PIC N(5).
     10 PASSWORD             PIC X(8).
******************************************************************
* THE NUMBER OF COLUMNS DESCRIBED BY THIS DECLARATION IS 6       *
******************************************************************
```

When to use UTF-8/UTF-16

Sample Schema - EBCDIC

```
CREATE TABLE DSN8XX0.EMP
 (EMPNO      CHAR(6)                        NOT NULL,
   FIRSTNME  VARCHAR(12)                    NOT NULL,
   MIDINIT   CHAR(1)                        NOT NULL,
   LASTNAME  VARCHAR(15)                    NOT NULL,
   WORKDEPT  CHAR(3)                                ,
   PHONENO   CHAR(4) CONSTRAINT NUMBER CHECK
     (PHONENO >= '0000' AND PHONENO <= '9999')      ,
   HIREDATE  DATE                                   ,
   JOB       CHAR(8)                                ,
   EDLEVEL   SMALLINT                               ,
   SEX       CHAR(1)                                ,
   BIRTHDATE DATE                                   ,
   SALARY    DECIMAL(9, 2)                          ,
   BONUS     DECIMAL(9, 2)                          ,
   COMM      DECIMAL(9, 2)                          ,
   PRIMARY KEY(EMPNO))
 CCSID EBCDIC;
```

Sample Schema – Unicode UTF-8 and SBCS

```
CREATE TABLE DSN8YY0.EMP
 (EMPNO      CHAR(6)  FOR SBCS DATA                 NOT NULL,
    FIRSTNME  VARCHAR(36)                           NOT NULL,
    MIDINIT   VARCHAR(3)                            NOT NULL,
    LASTNAME  VARCHAR(45)                           NOT NULL,
    WORKDEPT  CHAR(3)  FOR SBCS DATA                         ,
    PHONENO   CHAR(4)  FOR SBCS DATA CONSTRAINT NUMBER CHECK
      (PHONENO >= '0000' AND PHONENO <= '9999')              ,
    HIREDATE  DATE                                           ,
    JOB       CHAR(9)FOR SBCS DATA                           ,
    EDLEVEL   SMALLINT                                       ,
    SEX       CHAR(1)FOR SBCS DATA                           ,
    BIRTHDATE DATE                                           ,
    SALARY    DECIMAL(9, 2)                                  ,
    BONUS     DECIMAL(9, 2)                                  ,
    COMM      DECIMAL(9, 2)                                  ,
    PRIMARY KEY(EMPNO))
CCSID UNICODE;
```

Sample Schema – Unicode UTF-16

```
CREATE TABLE DSN8ZZ0.EMP
  (EMPNO       VARGRAPHIC(6)                      NOT NULL,
     FIRSTNME  VARGRAPHIC(12)                     NOT NULL,
     MIDINIT   VARGRAPHIC(1)                      NOT NULL,
     LASTNAME  VARGRAPHIC(15)                     NOT NULL,
     WORKDEPT  VARGRAPHIC(3)                              ,
     PHONENO   VARGRAPHIC(4) CONSTRAINT NUMBER CHECK
       (PHONENO >= '0000' AND PHONENO <= '9999')          ,
     HIREDATE  DATE                                       ,
     JOB       VARGRAPHIC(8)                              ,
     EDLEVEL   SMALLINT                                   ,
     SEX       VARGRAPHIC(1)                              ,
     BIRTHDATE DATE                                       ,
     SALARY    DECIMAL(9, 2)                              ,
     BONUS     DECIMAL(9, 2)                              ,
     COMM      DECIMAL(9, 2)                              ,
     PRIMARY KEY(EMPNO))
   CCSID UNICODE;
```

Sample Schema – Unicode UTF16 and SBCS

```
CREATE TABLE DSN8AA0.EMP
  (EMPNO     CHAR(6)  FOR SBCS DATA                  NOT NULL,
     FIRSTNME  VARGRAPHIC(12)                        NOT NULL,
     MIDINIT   VARGRAPHIC(1)                         NOT NULL,
     LASTNAME  VARGRAPHIC(15)                        NOT NULL,
     WORKDEPT  CHAR(3)  FOR SBCS DATA                        ,
     PHONENO   CHAR(4)  FOR SBCS DATA CONSTRAINT NUMBER CHECK
       (PHONENO >= '0000' AND PHONENO <= '9999')            ,
     HIREDATE  DATE                                         ,
     JOB       CHAR(8)  FOR SBCS DATA                       ,
     EDLEVEL   SMALLINT                                     ,
     SEX       CHAR(1)  FOR SBCS DATA                       ,
     BIRTHDATE DATE                                         ,
     SALARY    DECIMAL(9, 2)                                ,
     BONUS     DECIMAL(9, 2)                                ,
     COMM      DECIMAL(9, 2)                                ,
     PRIMARY KEY(EMPNO))
   CCSID UNICODE;
```

A Basic Approach

Pick a Development Strategy

- Big Bang
  - Faster
  - Cheaper
  - Riskier

Tends to be used with OTS SW, or when offshore resources involved

- Incremental
  - Slower
  - More Expensive
  - More manageable

Tends to be used for RYO SW when in-house resources are used

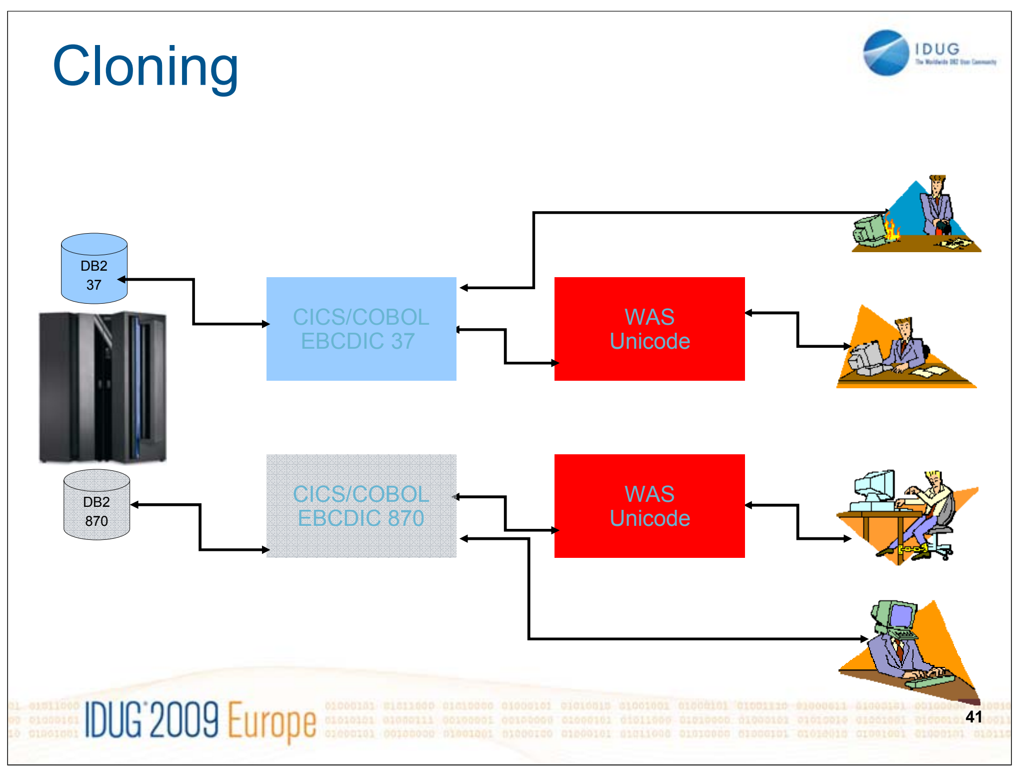
Combined

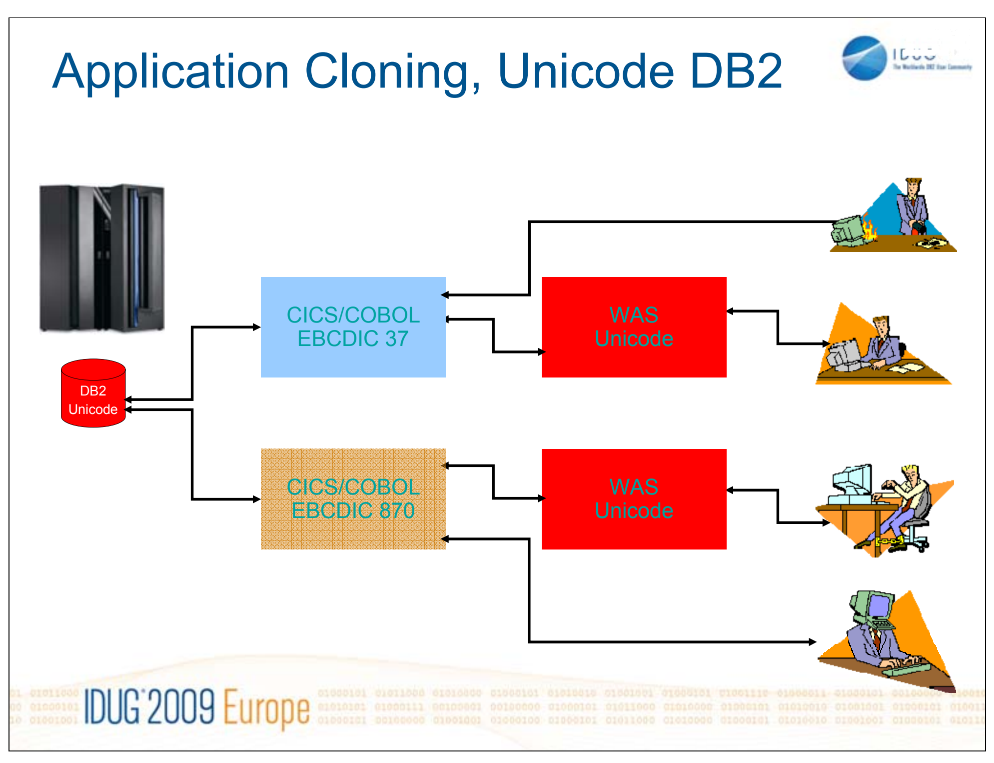
Pick a display strategy

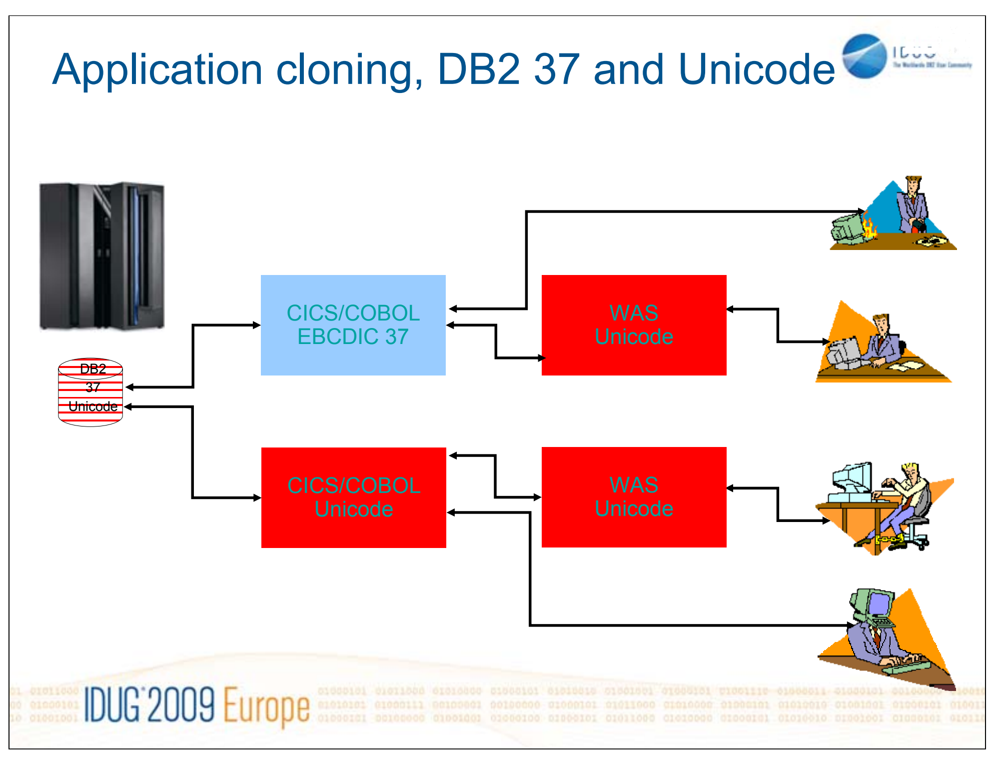
- 3270 Interfaces are not Unicode enabled
- What are you using today besides 3270
  - MQ
  - CICS Transaction Gateway
  - IMS Connect
  - ???
- Consider message “compression”
  - UTF-8 as transport can improve performance

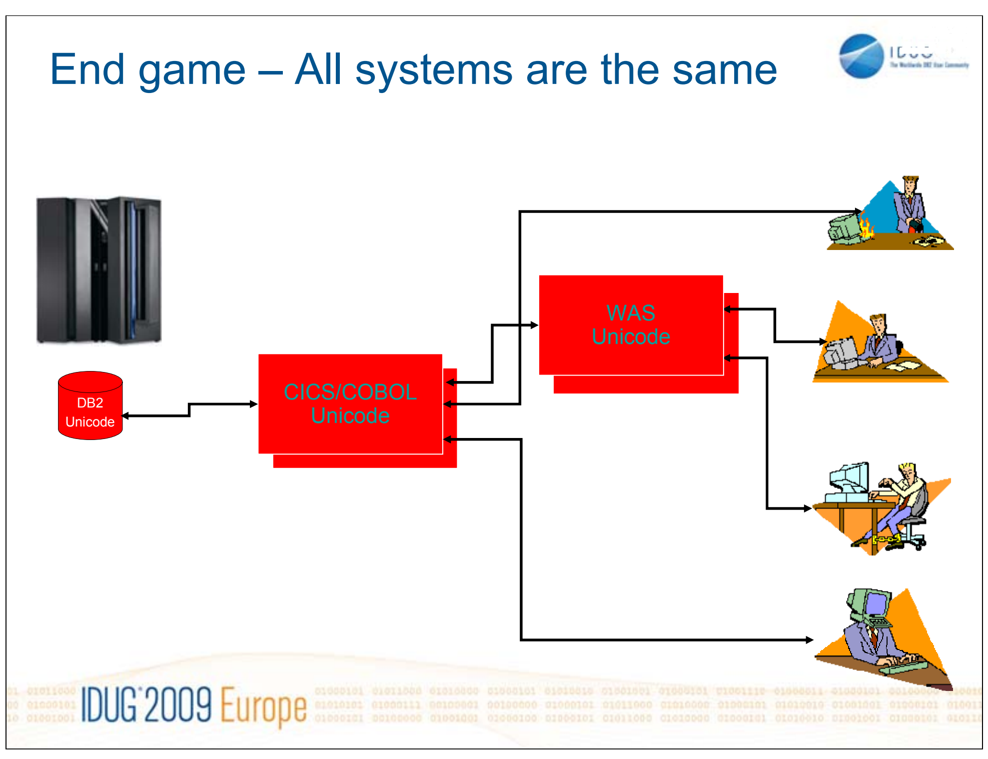
Keep it Simple

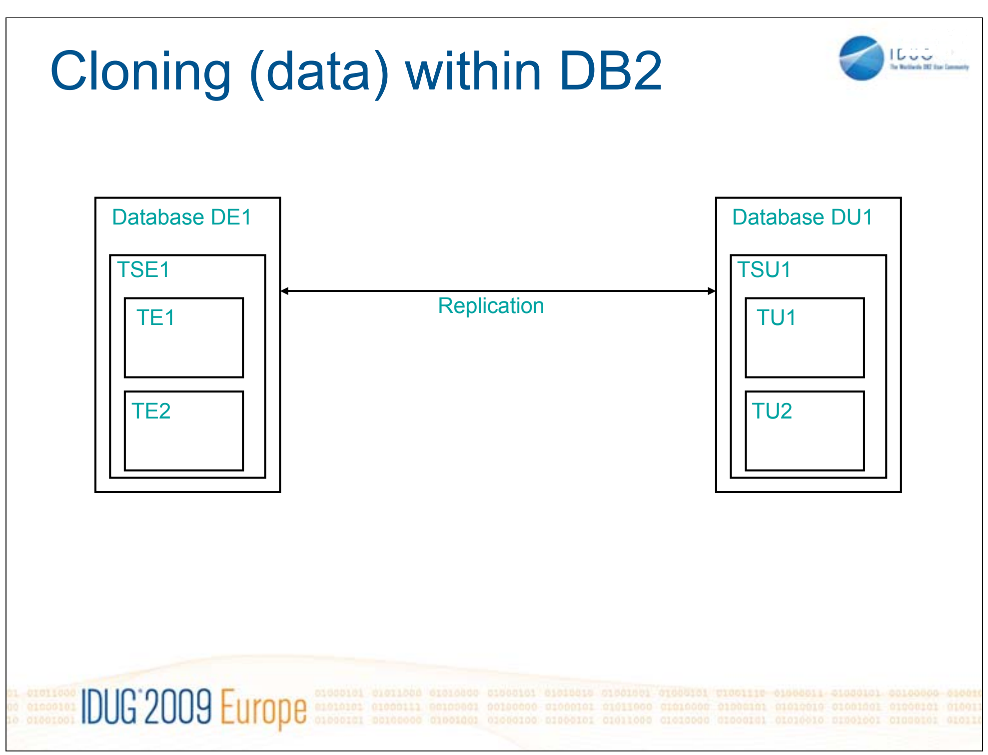
- Single or Multiple Encodings?
  - Most existing applications
    - One Encoding from end to end
    - One Encoding in DB, Second Encoding in App
  - Multiple encodings greatly increases complexity
    - Messaging technology typically is mono-encoding and string based – applications that are multi-encoding will have to convert to a single encoding on send/receive message
- Tactical -> Strategic approach

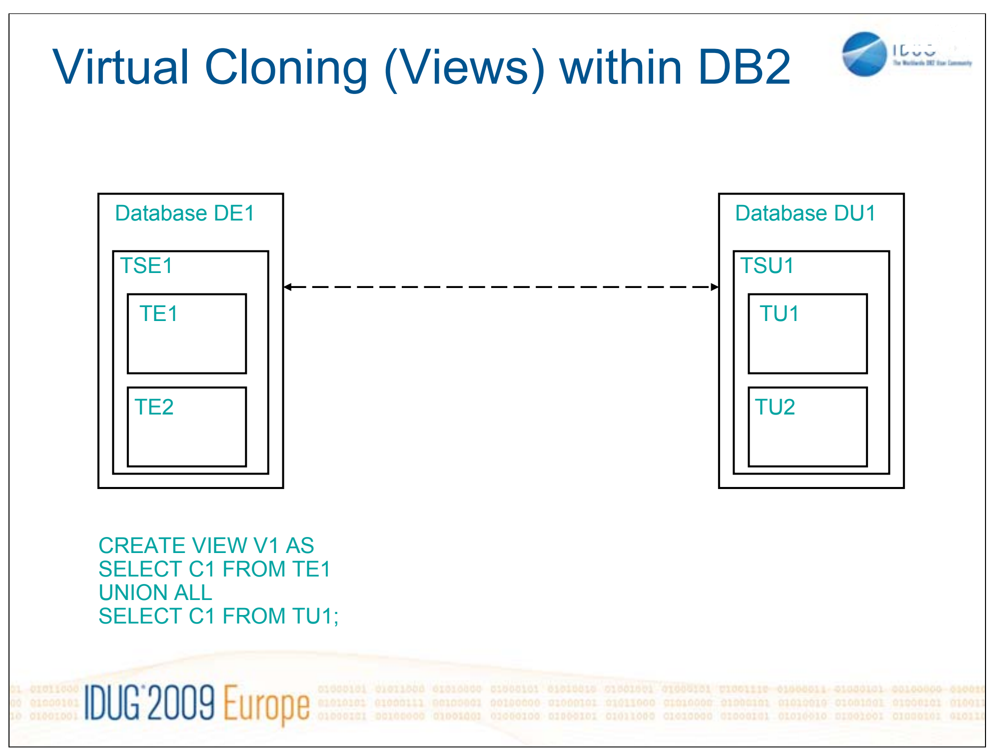

Cloning
IDUG
DB2
37
CICS/COBOL
EBCDIC 37
WAS
Unicode
DB2
870
CICS/COBOL
EBCDIC 870
WAS
Unicode
IDUG 2009 Europe
41


Application Cloning, Unicode DB2
DB2
Unicode
CICS/COBOL
EBCDIC 37
WAS
Unicode
CICS/COBOL
EBCDIC 870
WAS
Unicode
IDUG 2009 Europe


Application cloning, DB2 37 and Unicode
CICS/COBOL
EBCDIC 37
WAS
Unicode
DB2
37
Unicode
CICS/COBOL
Unicode
WAS
Unicode
IDUG 2009 Europe

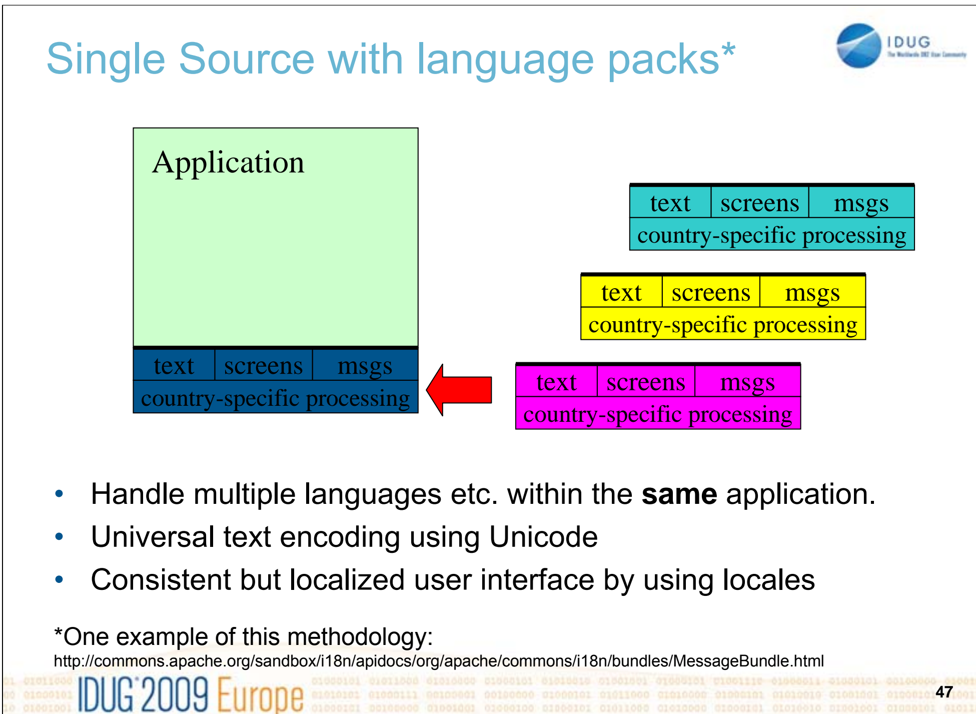

End game – All systems are the same
DB2
Unicode
CICS/COBOL
Unicode
WAS
Unicode
IDUG 2009 Europe


Cloning (data) within DB2
Database DE1
TSE1
TE1
TE2
Replication
Database DU1
TSU1
TU1
TU2
IDUG 2009 Europe



Virtual Cloning (Views) within DB2
Database DE1
TSE1
TE1
TE2
Database DU1
TSU1
TU1
TU2
CREATE VIEW V1 AS
SELECT C1 FROM TE1
UNION ALL
SELECT C1 FROM TU1;
IDUG 2009 Europe

Single Source with language packs*



- Handle multiple languages etc. within the **same** application.
- Universal text encoding using Unicode
- Consistent but localized user interface by using locales

*One example of this methodology:
http://commons.apache.org/sandbox/i18n/apidocs/org/apache/commons/i18n/bundles/MessageBundle.html

Conversion between codepages

- Performance and Functional preference
  - No Conversion
  - Lossless Conversions
    - UTF-8 <-> UTF-16
    - Latin-1 <-> Latin-1 (e.g. 37 <-> 500)
    - ASCII/EBCDIC -> Unicode
      - Watch out for characters that may not be in the “right” spot due to RT conversions
  - Round Trip
    - Most useful in a 2-tier, homogenous environment
  - Enforced Subset
    - Unicode -> ASCII/EBCDIC
    - Latin-n -> Latin-m

Where loss “may” be acceptable

IDUG

- There may be some cases where data loss is acceptable. Examples of areas where data loss may be acceptable include:
  - Internal reports where names appear, but are not “required” – for example show me the top 10 customers by deposit. It may be that the name is a “nice to have” field in the report, but not necessary.
  - Output only devices where data is not updated – for example, it may be acceptable for a period of time for tellers to have “green screen” applications that show customer names that won’t display correctly as long as a phonetic pronunciation is available. Logic may need to be prevent the tellers from updating names, addresses and other information if the tellers device is not capable of correctly representing all data.

Romanization

- Romanization is often used, in conjunction with the original data to provide a phonetic pronunciation of someone's name – for example, from http://en.wikipedia.org/wiki/Arabic_transliteration
  - An exact equivalent of e.g. صدام حسين would be **ṣd m ḥsyn**, which is meaningless to an untrained reader. The "full transliteration" adds information not in the text, which has to be supplied by a speaker of Arabic, **ṣaddām ḥussayn**. Usually, newspapers and popular books use not a transliteration, but a transcription: instead of translating each written letter they try to reproduce the sound of the words according to the orthography rules of the target language: **Saddam Hussein.**
- In many cases an additional column is added to a table containing names to provide a phonetic representation of the name using a Latin-1 alphabet:
  - Андрей -> Andrei

ASCII_STR & EBCDIC_STR

- Enable Unicode data to be returned without substitution in ASCII or EBCDIC. Data not directly convertible is "escaped"

- Assuming T1.C1 contains –
  "Hi, my name is Андрей"

```
SELECT C1 FROM T1; - Returns (to 3270 CCSID 37
  screen)
'Hi, my name is ......'

SELECT ASCII_STR(C1)FROM T1; - Returns
'Hi, my name is \0410\043D\0434\0440\0435\0439'
```

UNICODE_STR or UNISTR

- Accepts “escaped” Unicode input and converts it to UTF-8 or UTF-16.
- Assuming T1.C1 contains “Андрей”

```
SELECT ASCII_STR(C1)FROM T1; - Returns
‘\0410\043D\0434\0440\0435\0439’ – Escaped UTF-16

SELECT HEX(UNISTR(ASCII_STR(C1))) FROM T1; - Returns
‘D090D0BDD0B4D180D0B5D0B9’ – HEX of UTF-8

SELECT HEX(UNISTR(ASCII_STR(C1),UTF16)) FROM T1; -
  Returns
‘0410043D0434044004350439’ – HEX of UTF-16
```

Cultural conventions

- Numeric
  - 1,234 or 1.234?
- Date/Time
  - 07/27/1965 – obvious
  - 03/11/2003 – March 11th, or November 3rd?
- Calendar
  - Gregorian, Islamic (lunar), Chinese (lunisolar)
  - Work week (M-F, S-Th, ???)

Collation – COLLATION_KEY

```
CREATE TABLE T1 (C1 VARCHAR(6) ) CCSID UNICODE;
INSERT INTO T1 VALUES('Cat');
INSERT INTO T1 VALUES('cat');
SELECT C1 FROM T1 ORDER BY C1 ;
                +--------+
                |   C1   |
                +--------+
              1_| Cat    |
              2_| cat    |
                +--------+

SELECT C1 FROM T1 ORDER BY COLLATION_KEY(C1,'UCA410_LEL_CL');
                +--------+
                |   C1   |
                +--------+
              1_| cat    |
              2_| Cat    |
                +--------+
```

COLLATION_KEY – Example:

```
CREATE TABLE T1 (C1 VARCHAR(6) ) CCSID UNICODE;
INSERT INTO T1 VALUES('cote',1);
INSERT INTO T1 VALUES('côté',2);
INSERT INTO T1 VALUES('côte',3);
INSERT INTO T1 VALUES('coté',4);
```

COLLATION_KEY – Example:

```
SELECT C1 FROM T1 ORDER BY C1;
                +---------+
                |   C1    |
                +---------+
              1_| cote    |
              2_| coté    |
              3_| côte    |
              4_| côté    |
                +---------+

SELECT C1 FROM T1 ORDER BY COLLATION_KEY(C1,'UCA410_LFR_FO');
                +---------+
                |   C1    |
                +---------+
              1_| cote    |
              2_| côte    |
              3_| coté    |
              4_| côté    |
                +---------+
```

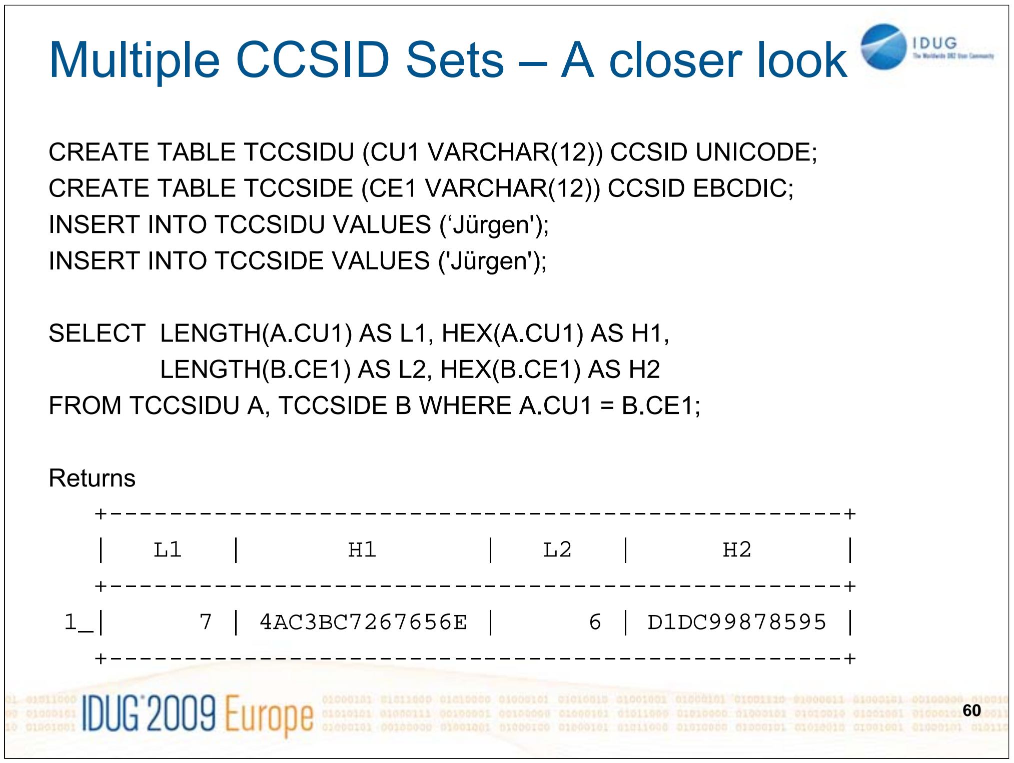
Timezone Issues

- Timezones
  - UTC
  - Local
  - Timestamp with Timezone (DB2 vNext)
- What time is it?
  - My phone always picks up local time
  - My Laptop is still on Pacific Time (UTC-7/8)

Christopher J. Crone

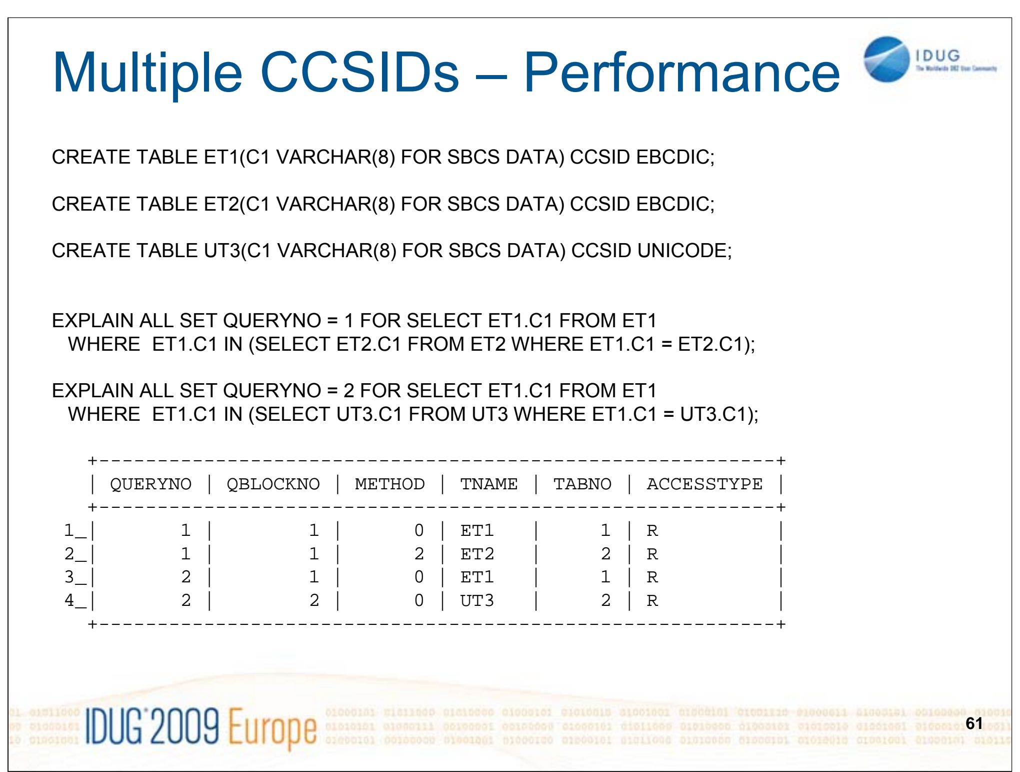
IBM – DB2 for z/OS Development
cjc@us.ibm.com

Reference Information



In this example, you can see that up until the data is returned to the application, the data from TCCSIDU is Unicode, and the data from TCCSIDE is EBCDIC. At the point the data is returned to the application (copied from DB2 to the applications host variables), conversion will be performed to the CCSID specified by the application (either implicitly or explicitly). In this case, since we are returning the HEX representation of the data, we really don’t convert the data, we convert the hex representation of the data (which will always be composed of characters between 0-9 and A-F).

The result is the same for all Join methods. For Sort Merge Join (SMJ), we convert the TCCSIDE.CE1 to Unicode before sorting. In the sort workfile, we carry both the EBCIDC version of the string (to return to the application), and the Unicode version of the string (for comparison processing in the Join). As a result, the size of workfiles can increase when EBCDIC and Unicode tables are joined (as compared to joining two EBCDIC or two Unicode tables). Additionally, it is possible that SQLCODE -136 or -670 may be issued.



In this example, we have three tables ET1 and ET2 are EBCDIC tables, and UT3 is a Unicode table.

The two queries are identical, except the second query has UT3 where the first query has ET2.

Note that table UT3 isn't really even using Unicode data (it is defined as "FOR SBCS DATA" which means that the data will be 7-bit ASCII).

When two EBCDIC tables are involved, as in QUERYNO=1, we are able to transform the query into a join. In the second example, the transformation to a join did not take place, as a result the second query may perform slower than the first query.

This is a very simple example that demonstrates the ramifications to performance of multiple CCSID statements.

Functions & Routines

- Functions – default is byte or double byte based
  - LENGTH, SUBSTR, POSSTR, LOCATE
    - Byte Oriented for SBCS and Mixed (UTF-8)
    - Double-Byte Character Oriented for DBCS (UTF-16)
  - V8 has new character based functions (see PQ88784)
- Cast Functions
  - UTF-16/UTF-8 accepted anywhere char is accepted (char, date, integer...)
    - SELECT DATE(graphic column) FROM T1;
    - SELECT INTEGER(graphic column) FROM T1;
- Routines
  - UDFs, UDTFs, and SPs will all be enabled to allow Unicode parameters
  - Parameters will be converted as necessary between char (UTF-8) and graphic (UTF-16)
  - Date/Time/Timestamp passed as UTF-8 (ISO Format)

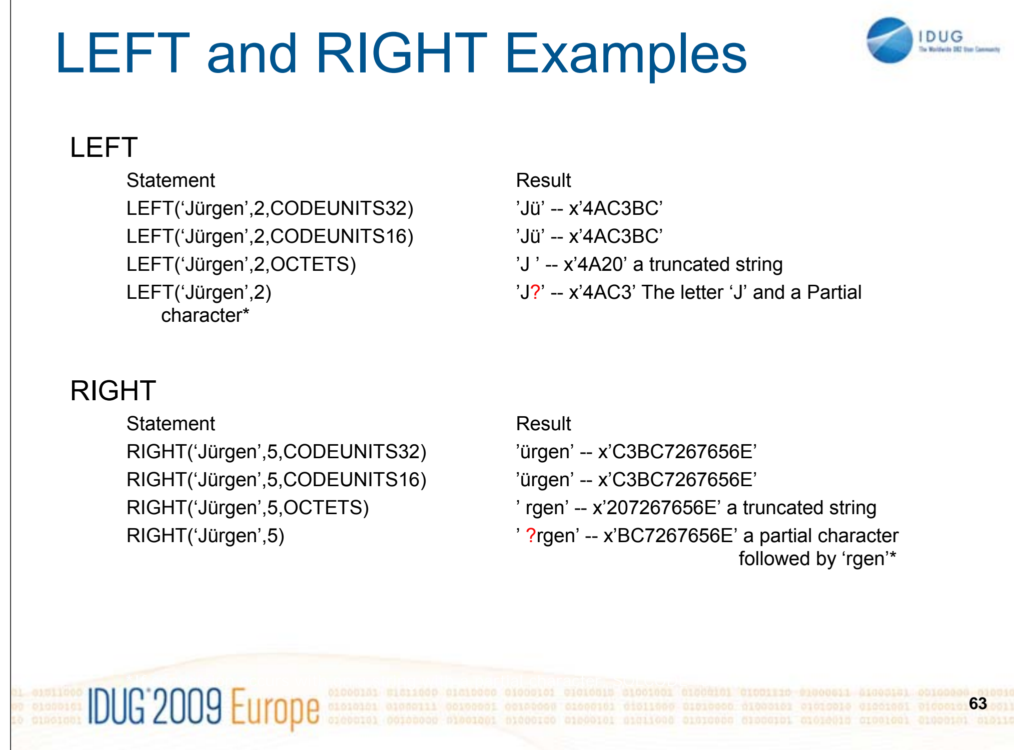
All Built In Functions (BIFs) have been extended to support Unicode

Some BIFs, such as LENGTH, SUBSTR, POSSTR, and LOCATE are byte oriented for UTF-8 and Double-Byte character oriented for UTF-16

Many new functions were added in V7, the CCSID_ENCODING function has been added to help users determine the encoding, ASCII, EBCDIC, or UNICODE of a particular CCSID

UTF-16 data is accepted in casting type functions such as DATE or INTEGER

Result CCSIDs for functions that return character strings will return UTF-8/CCSID 1208



LEFT and RIGHT operate in a similar manner.

In these examples, CODEUNITS32 and CODEUNITS16 return the same results.

When OCTETS is specified, an a partial character would result, the PAD character is used.

When no units are specified, the behavior is similar to that of OCTETS (byte based), but partial characters can result.

SUBSTRING

Function ...	Returns ...
SUBSTRING(‘Jürgen',1,2,CODEUNITS32)	’Jü’ -- x’4AC3BC’
SUBSTRING(‘Jürgen',1,2,CODEUNITS16)	’Jü’ -- x’4AC3BC’
SUBSTRING(‘Jürgen',1,2,OCTETS)	’J ’ -- x’4A20’ - a truncated string
SUBSTR(‘Jürgen',1,2)	‘J?’ – x’4AC3’ – a partial character
SUBSTRING(‘Jürgen',8,CODEUNITS16)	a zero-length string
SUBSTRING(‘Jürgen',8,4,OCTETS)	a zero-length string
SUBSTRING(’ABCDEFG’,-2,2,OCTETS)	a zero-length string
SUBSTRING(’ABCDEFG’,-2,4,OCTETS)	’A’
SUBSTRING(’ABCDEFG’,-2,5,OCTETS)	’AB’
SUBSTRING(’ABCDEFG’,-2,OCTETS)	’ABCDEFG’
SUBSTRING(’ABCDEFG’,0,1,OCTETS)	a zero-length string

Let C be the value of the *string-expression*, let LC be the length in characters of C, and let S be the value of the *start*. v If *length* is

The first and second examples return the same value because even though ‘ü’ is a multi-byte character in UTF-8, it is one CODEUNITS32 or CODEUNITS16 codepoint.

In the third example OCTETS is specified and since a partial character would have resulted, truncation occurs on the partial character and a pad character is returned in place of the partial character.

In the fourth example, we are using SUBSTR, not SUBSTRING. SUBSTR is byte based and will return partial characters. As mentioned previously if conversion occurs on a string with a partial character, -330 will result.

specified, let L be the value of <string length> and let E be S+L. Otherwise, let E be the larger of LC + 1 and S. v If either C, S, or L is the null value, the result of the function is the null value. v If E is less than S, an exception condition is raised: data exception — substring error. v Otherwise: – If S is greater than LC or if E is less than 1 (one), the result of the function is a zero-length string. – Otherwise: - Let S1 be the larger of S and 1 (one). Let E1 be the smaller of E and LC+1. Let L1 be E1–S1. - The result of the function is a character string that contains the L1 characters of C starting at character number S1 in the same order that the characters appear in C.

Unicode value \u1D400 - 'A' MATHEMATICAL BOLD CAPITAL A

UTF-8	UTF-16	UTF-32
X'F09D9080'	X'D835DC00'	X'0001D400'

Assume that C1 is a VARCHAR(10) column, encoded in Unicode UTF-8, and that table T1 contains one row with the value of the mathematical bold capital A (X'F09D9080').

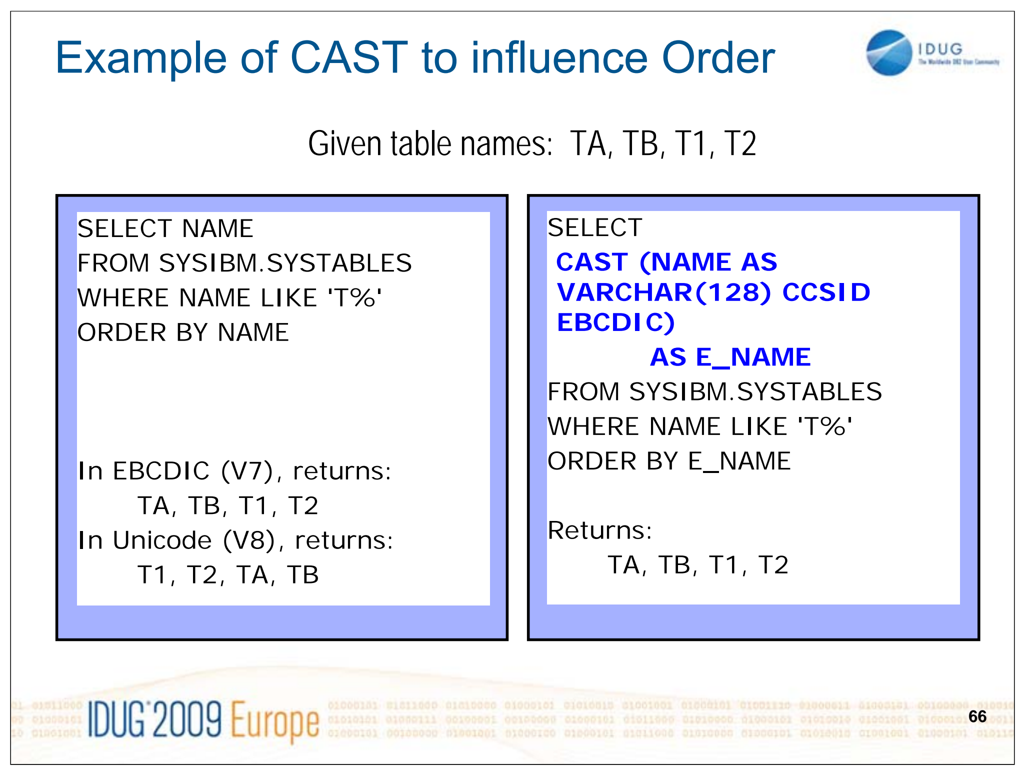
The following similar queries return different answers:

```
SELECT CHARACTER_LENGTH(C1,CODEUNITS32) FROM T1;  -- Returns 1
SELECT CHARACTER_LENGTH(C1,CODEUNITS16) FROM T1;  -- Returns 2
SELECT CHARACTER_LENGTH(C1,OCTETS) FROM T1;       -- Returns 4
```

The following similar queries return different answers:

```
SELECT HEX(SUBSTRING(C1,1,1,CODEUNITS32)) FROM T1;  -- Returns X'F09D9080'
SELECT HEX(SUBSTRING(C1,1,1,CODEUNITS16)) FROM T1;  -- Returns X'20'
SELECT HEX(SUBSTRING(C1,1,2,CODEUNITS16)) FROM T1;  -- Returns X'F09D9080'
SELECT HEX(SUBSTRING(C1,1,1,OCTETS)) FROM T1;       -- Returns X'20' (blank)
SELECT HEX(SUBSTR(C1,1,1)) FROM T1;                 -- Returns X'F0'
```


For many characters, specifying CODEUNITS16 and CODEUNITS32 on a BIF does not matter. However, when supplementary characters (sometimes also called surrogate characters) are used, the results can be different.



The new CAST specification can be used to influence ordering.

The above example shows how SQL could be recoded to cause data to be returned in the same order as DB2 V7 would.

CAST – Changing a CCSID

- If both the *length* and CCSID clauses are specified, the data is first cast to the specified CCSID, and then the *length* is applied.

  Example:

  CAST ('Jürgen' as VARCHAR(6) CCSID UNICODE)
  Returns 'Jürge'

- If either CODEUNITS32 or CODEUNITS16 is specified, the specification of *length* applies to the units specified.

  Example:

  CAST ('Jürgen' as VARCHAR(6 CODEUNITS16) CCSID UNICODE)
  Returns 'Jürgen'

When a length is specified as part of a CAST that also specifies a CCSID, the length is applied after the data has been converted to the target CCSID, unless CODEUNITS16 or CODEUNITS32 is specified. If CODEUNITS16 or CODEUNITES32 is specified, then the length applies in the specified units, then the data is cast to the final CCSID.

ISPF

- DISPLAY CCSID ccsid_number [LINE start_line end_line] [COLS start_col end_col]
  - CCSID
    - n
    - UTF8
    - UTF16
    - UTF32
    - UCS2
    - EBCDIC
    - UNICODE
- FIND C'xxxxxx' UTF8
- OA07685 (OA08496 fixes a problem with FIND)
- z/OS 1.4, 1.5, and 1.6

DB2 Connect and the Euro

- DB2 Connect uses CODEPAGEs not CCSIDs. This can cause problems because (for instance):
  - CODEPAGE 1252 maps the EURO
  - CCSID 1252 does not map the EURO
    - CCSID 5348 maps the EURO
- DB2 for z/OS allows specification of EURO and Non-EURO CCSIDs
  - For Example
    - 37 and 1140
- DB2 for z/OS does some conversions, DB2 Connect/LUW does some conversions. We are not consistent.
- DB2 LUW V8 fixpack 11 offers some help
  - New Registry Variable
    - DB2CONNECT_ENABLE_EURO_CODEPAGE
  - Tells DB2 Connect/LUW to use EURO or Non-Euro Codepage when communicating over DRDA.

In a DRDA client – server transaction, each side performs ½ of the conversions. The server converts the data it receives from the client to the CCSID that it needs (for instance, the CCSID of the table on an INSERT statement), and the client converts the data it receives to the CCSID it needs (for instance, when a SELECT is issued).

LUW ODBC/CLI DisableUnicode

db2cli.ini keyword syntax:

DisableUnicode = <not set> | 0 | 1

If an application is Unicode as indicated by:

- SQL_ATTR_ANSI_APP connection attribute is set to SQL_AA_FALSE
- Connection occurred with SQLConnectW()

Then the DisableUnicode keyword can be used to effect three different connection behaviors:

- DisableUnicode is not set in the db2cli.ini file:
  - If the target database supports Unicode, DB2 CLI will connect in Unicode code pages (1208 and 1200). Otherwise, DB2 CLI will connect in the application code page.
- DisableUnicode=0 is set:
  - DB2 CLI always connects in Unicode, whether or not the target database supports Unicode.
- DisableUnicode=1 is set:
  - DB2 CLI always connects in the application code page, whether or not the target database supports Unicode.

Java

- charOutputSize – (t2) sp inout/out parm size needs to take into account expansion
- getUnicodeString/setUnicodeString
  - UnicodeString - A class for handling an input stream whose bytes should be interpreted as Unicode
- sqlj.runtime.UnicodeStream class
- SQLJ/JDBC T4 sends UTF-8
- SQLJ/JDBC T2
  - The conversions that the JDBC/SQLJ driver for z/OS and the IBM DB2 Driver for JDBC and SQLJ with type 2 connectivity support are also limited to those that are supported by the underlying JRE implementation. Those drivers use CCSID information from the database server if it is available. The drivers convert input parameter data to the CCSID of the database server before sending the data. If target CCSID information is not available, the drivers send the data as Unicode UTF-8.

JDBC T2 Setup

- SQLJ/JDBC T2
  - The conversions that the JDBC/SQLJ driver for z/OS and the IBM DB2 Driver for JDBC and SQLJ with type 2 connectivity support are also limited to those that are supported by the underlying JRE implementation.
  - The T2 drivers use CCSID information from the database server if it is available. The drivers convert input parameter data to the CCSID of the database server before sending the data.
  - If target CCSID information is not available, the drivers send the data as Unicode UTF-8.

T2 JDBC Driver Prep

Putting DB2JccConfiguration.properties in a JAR file:*

1. Rename DB2JccConfiguration.properties to another name, such as EBCDICVersion.properties.
2. Run the iconv shell utility on the z/OS UNIX System Services command line to convert the file contents to Unicode. For example, to convert EBCDICVersion.properties to a Unicode file named DB2JccConfiguration.properties, issue this command:
   - iconv -f ibm-1047 -t utf-8 EBCDICVersion.properties \ > DB2JccConfiguration.properties
3. Execute the jar command to add the Unicode file to the JAR file. In the JAR file, the configuration properties file must be named DB2JccConfiguration.properties. For example:
   - jar -cvf jdbcProperties.jar DB2JccConfiguration.properties

*Assuming a 1047 EBCDIC codepage for USS

JDBC JCC Driver

db2.jcc.sendCharInputsUTF8

- Specifies whether the IBM DB2 Driver for JDBC and SQLJ converts character input data to the CCSID of the DB2 for z/OS database server, or sends the data in UTF-8 encoding for conversion by the database server (z/OS only).

Possible values are:

- **no, false, or 2** Specifies that the IBM DB2 Driver for JDBC and SQLJ converts character input data to the target encoding before the data is sent to the DB2 for z/OS database server. This is the default.
- **yes, true, or 1** Specifies that the IBM DB2 Driver for JDBC and SQLJ sends character input data to the DB2 for z/OS database server in UTF-8 encoding. The database server converts the data from UTF-8 encoding to the target CCSID.
- Specify yes, true, or 1 only if conversion to the target CCSID by the SDK for Java causes character conversion problems. The most common problem occurs when you use IBM DB2 Driver for JDBC and SQLJ type 2 connectivity to insert a Unicode line feed character (U+000A) into a table column that has CCSID 37, and then retrieve that data from a non-z/OS client. If the SDK for Java does the conversion during insertion of the character into the column, the line feed character is converted to the EBCDIC new line character X'15'. However, during retrieval, some SDKs for Java on operating systems other than z/OS convert the X'15' character to the Unicode next line character (U+0085) instead of the line feed character (U+000A). The next line character causes unexpected behavior for some XML parsers. If you set db2.jcc.sendCharInputsUTF8 to yes, the DB2 for z/OS database server converts the U+000A character to the EBCDIC line feed character X'25' during insertion into the column, so the character is always retrieved as a line feed character.
- For example, any of the following settings for db2.jcc.sendCharInputsUTF8 causes the IBM DB2 Driver for JDBC and SQLJ to convert input character strings to UTF-8, rather than the target encoding, before sending the data to the database server: db2.jcc.sendCharInputsUTF8=yes db2.jcc.sendCharInputsUTF8=true db2.jcc.sendCharInputsUTF8=1

Conversion of data to the target CCSID on the database server might cause the IBM DB2 Driver for JDBC and SQLJ to use more memory than conversion by the driver. The driver allocates memory for conversion of character data from the source encoding to the encoding of the data that it sends to the database server. The amount of space that the driver allocates for character data that is sent to a table column is based on the maximum possible length of the data. UTF-8 data can require up to three bytes for each character. Therefore, if the driver sends UTF-8 data to the database server, the driver needs to allocate three times the maximum number of characters in the input data. If the driver does the conversion, and the target CCSID is a single-byte CCSID, the driver needs to allocate only the maximum number of characters in the input data.

For example, any of the following settings for db2.jcc.sendCharInputsUTF8 causes the IBM DB2 Driver for JDBC and SQLJ to convert input character strings to UTF-8, rather than the target encoding, before sending the data to the database server: db2.jcc.sendCharInputsUTF8=yes db2.jcc.sendCharInputsUTF8=true db2.jcc.sendCharInputsUTF8=1

z/OS ODBC/CLI Determining Encoding

- ASCII data is placed into application variables when both of the following conditions are true:
  - CURRENTAPPENSCH = ASCII is specified in the initialization file.
  - The *fCType* argument specifies SQL_C_CHAR or SQL_C_DBCHAR in the SQLBindCol() call.
- EBCDIC data is placed into application variables when both of the following conditions are true:
  - CURRENTAPPENSCH = EBCDIC is specified in the initialization file, or the CURRENTAPPENSCH keyword is not specified in the initialization file.
  - The *fCType* argument specifies SQL_C_CHAR or SQL_C_DBCHAR in the SQLBindCol() call.
- Unicode UTF-8 data is placed into application variables when both of the following conditions are true:
  - CURRENTAPPENSCH = UNICODE is specified in the initialization file.
  - The *fCType* argument specifies SQL_C_CHAR in the SQLBindCol() call.
- Unicode UCS-2 data is placed into application variables when both of the following conditions are true:
  - CURRENTAPPENSCH = UNICODE is specified in the initialization file.
  - The *fCType* argument specifies SQL_C_WCHAR in the SQLBindCol() call.

SYSDUMMYx Tables

- The tables that you need to create are:
  - SYSIBM.SYSDUMMYU
  - SYSIBM.SYSDUMMYA
  - SYSIBM.SYSDUMMYE
- These tables ensure that character conversion does not occur when data is stored in DBCLOB or CLOB columns.