Session: F17

DB2 9 New Datatypes and SQL Functions: Blessing or Curse?

Peter Vanroose
ABIS Training & Consulting

IDUG
The Worldwide DB2 User Community

Thursday 8 October 2009 • 11:00
Platform: DB2 9 for z/OS and LUW

Scope

- this presentation is aimed at you:

if you are using DB2

- as a writer of SQL
- as an application developer
- as a DBA
- as a database designer
- on platforms z/OS or Linux, Unix, MS-Windows

if you are already at, or soon going to version 9

- mix of "new features in v9", "SQL performance", "user experiences", "cross-platform development".

Objectives

- learn about datatypes DECFLOAT, BIGINT, BINARY and VARBINARY, and manipulate DECFLOAT data through QUANTIZE(), NORMALIZE_DECFLOAT(), COMPARE_DECFLOAT(), DECFLOAT_SORTKEY()
- using the new standard SQL functions (now in DB2 9): EXTRACT, POSITION, LOCATE, ...
- how to implement “cultural sort” (i.e., “locale” support) into an application by using COLLATION_KEY()
- feel at ease with Unicode, octets, codeunits32, and the text manipulation functions
- easier statistical data analysis (on LUW) by using the new aggregation and regression functions

ideas gathered from ...

- overview of DB2 9 new possibilities
  - *datatypes*: BIGINT, DECFLOAT, BINARY
  - SQL scalar *functions*
  - evolutions: DW; XML; application programming
- standardisation tendencies
  - Edgar Codd and *domains*
  - SQL ISO/ANSI *standards*
  - IEEE standards for *floating point* representation
  - Unicode
  - DB2 cross-platform convergence
  - *compatibility*: Oracle, MySQL, SQLServer...

Agenda

- datatypes and standardisation
- DB2 cross-platform uniformisation
- text: VARCHAR, BINARY; UNICODE
- numeric datatypes: BIGINT
- floating-point: the IEEE-754 standard
- numeric datatypes: DECFLOAT
- scalar functions for DECFLOAT
- SQL standard manipulators for text data
- cultural sort
- DW and extended statistical functionality

thoughts on datatypes

- Edgar Codd's heritage:
  - relational model ≠ physical implementation
  - domains: more than just datatypes:
    - semantics of a column; referential integrity
    - range and precision (granularity, quantisation)
  - “date”: example of non-elementary datatype
    - combination of 3 numbers (day, month, year)
    - Interface ('2009-10-08') ≠ internal representation
  - even *text* is non-trivial:
    - CCSIDs: mapping of byte value to character set
    - Unicode, esp. UTF-8
    - VARCHAR: length as prefix ↔ nul-terminated ↔ RRF

SQL standardisation & datatypes

- SQL:1999
  - CHAR(n), VARCHAR(n), CLOB(n), BLOB(n)
  - INT, SMALLINT, BIGINT, DECIMAL(n,p)
  - REAL, FLOAT(n), DOUBLE
  - DATE, TIME, TIMESTAMP, INTERVAL
- SQL:2003
  - BOOLEAN
- SQL:2008
  - BINARY, VARBINARY; BOOLEAN dropped
- for a next release?
  - DECFLOAT

Evolutions in DB2

- why?
  - helps application development
  - "education" - also DBA (cross-platform)
- how?
  - bring SQL syntactically closer to each other
- why historic differences z/OS LUW?
  - different programming languages:
    - COBOL, PL/I: DECIMAL, CHAR(n)
      INT, SMALLINT
    - C, Java: VARCHAR
      INT, SMALLINT, BIGINT
      REAL, DOUBLE ("FLOAT")

Importance of data representation

- (mis)matching data representation
  database ↔ program
  - (un)efficient communication
  - (no) unexpected surprises:
    - numeric precision; rounding effects
    - overflow; underflow
    - truncation; trailing blanks
    - CCSID interpretation
  - education; standardisation
- (cf. misunderstanding of NULL)

VARCHAR

- not so new ... but still ...
  - data representation:
    - DB2: 2-byte length + actual data
    - C: actual data, ended with '\0'
  - DB2 v8 allows “alter table alter column”
    - CHAR(64) → VARCHAR(1024)
    - consequences?
  - DB2 9 for z/OS: RRF (reordered row format)
- getting a feeling for:
  - (mis)matching data representation
  - strict separation:
    “concept” vs “physical implementation”

VARCHAR

- CHAR(n) vs VARCHAR(n):
  - not transparent to application!
    - DB2 data is VARCHAR, host variable is CHAR:
      ==> must RTRIM before INSERT/UPDATE
    - DB2 data is CHAR, host variable is VARCHAR:
      ==> must RTRIM during SELECT

(could be made transparent through a view)

  - advantage of VARCHAR:
    - less average storage ==> more rows per I/O
  - disadvantage of VARCHAR:
    - runtime computation of next field pos (→ RRF)
  - PADDED vs NOT PADDED indexes (z/OS)

BINARY and VARBINARY

- new datatypes in DB2 9 for z/OS
  - alias (but not quite): “CHAR(n) FOR BIT DATA”
  - better (?) name would have been: “BYTE(n)”
- guarantee of non-interpreted bytes
  - CHAR will auto-convert CCSIDs (since v8)
  - BINARY is compatible with BLOB
- usage: “code” field; pictures; x'FF'; TINYINT
- is the most elementary datatype!
  - e.g.: INT == BINARY(4) with interpretation
  - CHAR(30) = BINARY(30) w. interpretation
  - note: CHAR(30) ≠ 30 characters!

Unicode

- Avoids CCSID conversion problems
  - CCSIDs 87 and 500 (EBCDIC) have no ş š Ł ő
  - ISO 8859-2 (Latin-2) has no à è ê û æ å ð
  - CCSIDs 87 and 500: inconsistently map |!¢¬[ ]^
- More than 256 chars => 1 char ≠ 1 byte !
- Codepoints vs encoding (UTF-8, UTF-16)
- UTF-8:
  - 1-byte chars: digits, non-accented chars, punct
  - 2-byte chars: most “short” alphabets
  - 3-byte chars: e.g. €, Japanese, Chinese (simpl)
  - 4-byte chars: e.g. Gothic, Chinese (full)

Unicode: caveats

- DB2 v8 for z/OS: catalog in Unicode
- DB2 v8 for LUW: Unicode database
  - application declares its character encoding
  - bind time: all SQL is interpreted in this CCSID
  - runtime: host vars interpreted in this CCSID
  - DB2 converts where necessary
- ORDER BY: follows the table encoding
  - what about “virtual” tables / views?
  - application needs knowledge of data encoding
- v8 feature becomes potential nightmare!
  - e.g. LENGTH(s) → byte length or # characters?

BIGINT

- 64-bit (8-byte) signed integer
  - hardware-supported on 64-bit processors
  - return type of RID(tbl)
  - maps to “long int” / “bigint” in host languages
- ranges from -9×10^{18} to 9×10^{18}
  - (2^{63} = 9˙223˙372˙036˙854˙775˙808)
- already in DB2 v8 for LUW
- no longer need DECIMAL(18,0)

See other presentation at this IDUG:

E13 “DECFLOATing in a BIGINT pool of (VAR)BINARY data” (Steve Thomas)

Floating-point numbers

- non-integral numbers:
  - DECIMAL(n,p): fixed precision n, fixed scale p
  - FLOAT(n): REAL (32 bit) & DOUBLE (64 bit)
  - since v9: DECIMAL FLOAT (DECFLOAT)
- DECIMAL: advantages:
  - integer based arithmetic; fixed “window”
  - decimal representation => “no” rounding errors
- FLOAT: advantages:
  - each number stores its decimal point position

==> larger range than DEC, still fixed precision

- DECFLOAT: combines these advantages

floating point: representation

- general form: $(-1)^s \times c \times b^{1-p} \times b^q$
  - **b**: base (10 or 2), fixed; **p**: precision, fixed
  - **s**: sign; **c**: p-digit "coefficient"; **q**: exponent (int)
- examples: (b=10)
  - 123.40 ==> $(-1)^0 \times 1.2340 \times 10^2$
  - -999e6 ==> $(-1)^1 \times 9.99 \times 10^8$ = -999000000
  - 88.77e-6 ==> $(-1)^0 \times 8.877 \times 10^{-5}$ = 0.00008877
- exponent: q between -emax & emax ($=2^w-1$)
- storage needs 3 parts:

s (1bit), c (p digits), q (w+1 bits).

==> typically > 75% for c, < 20% for q.

floating point: IEEE-754

- standard from 1985 for b=2, p=23 or 52
  - supported by compilers and hardware (FPU)
  - p=23, emax=127: needs 1+23+1+7=32 bits
  - p=52, emax=1023: needs 1+52+1+10=64 bits
  - these are the datatypes REAL and DOUBLE
- advantage of b=2: fast multiply; compact
- disadvantage of b=2: I/O conversion dec
- largest possible REAL (*based on this*):

$1.11111111111111111111111 \times 10^{1111111}$ (binary)

$= (2^{24}-1)\times 2^{127-22} = 1.68\times 10^{7}\times 4.05\times 10^{31} = 6.806\times 10^{38}$

with precision of 6.92 decimal digits (23 bits)

floating point: IEEE-754

- IEEE-754 also defines +Inf, -Inf, NaN, sNaN:

-∞ = -Inf < any finite number < +Inf = +∞

+x / 0 = +Inf; -x / 0 = -Inf; 0 / 0 = NaN

+Inf + x= +Inf; x - +Inf = -Inf, ...

x / Inf = 0; Inf / Inf = NaN

all operations with NaN return NaN

- Overflow / underflow

due to limited emax (independent of precision p):

example (p=5, emax=7):

-1.1111e4 × 2.0000e4 = -Inf (with overflow)

1.1111e-5 / 1000 = 0 (with underflow)

floating point: IEEE-754

- IEEE-754 extended in aug. 2008:
  - b=2, p=112, emax=16383 ==> "long double"
  - b=10, p=16, emax=384 ==> DECFLOAT(16)
  - b=10, p=34, emax=6144 ==> DECFLOAT(34)
- "long double" was already intensively used
- "DECIMAL FLOAT" is relatively new

  see e.g. http://speleotrove.com/decimal/
- HW support: z9, z10, Power6 (IBM)
- SW support: gcc4.2; IBM compilers; DB2 9

DECFLOAT

- Datatype DECFLOAT(16):
  - 8 bytes of storage ***(or 9 ?!)***
  - 16 digits precision ==> 53.1 bits needed
  - largest: 9.999....9 × 10^{384} ==> exp needs 9.5 bits
  - smallest positive: 10^{-383}
- Datatype DECFLOAT(34):
  - 16 bytes of storage
  - 34 digits precision ==> 113 bits needed
  - largest: 9.999....9 × 10^{6144} ==> exp needs 14 bits
  - smallest positive: 10^{-6143}
- Some minor details left out here ...

DECFLOAT: constructors

- Table with DECFLOAT(34) column:

```
CREATE TABLE t ( p INT NOT NULL, d DECFLOAT ) ;
INSERT INTO t(p,d) VALUES (1, 123.45) ;
INSERT INTO t(p,d) VALUES (2, -1000) ;
INSERT INTO t(p,d) VALUES (3, decfloat('111e99')) ;
INSERT INTO t(p,d) VALUES (4, CAST('-111e-99' AS decfloat)) ;
INSERT INTO t(p,d) VALUES (5, decfloat('-Inf')) ;
SELECT * FROM t ORDER BY d ;
```

```
P      D
------ --------------------
     5            -Infinity
     2                -1000
     4            -1.11E-97
     1               123.45
     3            1.11E+101
```

DECFLOAT: computations

- (notice truncation and rounding effects)

```
SELECT * FROM t WHERE d < -1111 ;
SELECT p, d + 1.23 FROM t ;
SELECT p, d * decfloat('1.000e-2') FROM t WHERE d >= -1000 ;
```

```
5             -Infinity

1                                              124.68
2                                             -998.77
3 1.110000000000000000000000000000000E+101
4      1.230000000000000000000000000000000
5                                           -Infinity

1             1.2345000
2             -10.00000
3           1.11000E+99
4         -1.11000E-99
```

DECFLOAT: rounding modes

- CURRENT DECFLOAT ROUNDING MODE (special register)
  - ROUND_HALF_UP: the natural "default"

1.432 → 1.4   1.750 → 1.8   -1.750 → -1.8

  - ROUND_DOWN: truncation

1.432 → 1.4   1.750 → 1.7   -1.750 → -1.7

  - ROUND_FLOOR: "towards -∞"

1.432 → 1.4   1.750 → 1.7   -1.750 → -1.8

  - ROUND_CEILING: "towards +∞"

1.432 → 1.5   1.750 → 1.8   -1.750 → -1.7

  - ROUND_HALF_EVEN: last digit even

1.432 → 1.4   1.750 → 1.8   1.85 → 1.8

DECFLOAT: child diseases

- Several APARs related to DECFLOAT

DB2 9.5 for LUW - 5 may 2008
IZ09711: POSSIBLE STACK CORRUPTION CONVERTING DECFLOAT(16) TO DOUBLE
The conditions under which this problem can arise are fairly specific:
* only conversions from DECFLOAT(16) to double are affected;
* one must be carrying out an affected operation on a DECFLOAT(16) value;
* the DECFLOAT(16) value being converted must contain 16 digits; and
* the DECFLOAT(16) value, x, being converted must be in the range 1E-6 < ABS(x) < 1.
Problem conclusion
First fixed in DB2 UDB Version 9.5, FixPak 1
===
IZ12232: NEW DECFLOAT COLUMN IN 9.5 NOT IN RESULTSETS FOR CLI FUNCTIONS
When a table is created with the new DECFLOAT column in 9.5, the column information is not in the result set from SQLColumns, although columns values can still be selected.
===
SPUFI issues
===
...

- Still some misunderstanding → education

DECFLOAT: current use

- XML
  - DB2 9 has new datatype “XML”
  - DB2 9 can extract fragments from XML documents (by using Xpath)
  - XML data is textual

    even for numeric data: '123.45'   '1.2345e2'
  - Conversion errors are unacceptable

    real('0.2') → 0.1999996 (binary 1.100110011...e11)

    decfloat('0.2') → 2.000000...000e-1
- “float”-like programming interface in SQL

DECFLOAT: scalar functions

- NORMALIZE_DECFLOAT(f)
  - returns a decfloat in “normalized” form

    e.g. normalize_decfloat('56.7800e+8') → '5.678e+9'
- QUANTIZE(f, q)
  - returns truncated/rounded or denormalized form

    e.g. quantize('56.7800e+8','100e7') → '568e+7'

    e.g. quantize('56.7800e+8','100.00000e7') → '567.80000e+7'
  - rounding mode is obeyed!
  - returns a DECFLOAT(34)

    unless both arguments are DECFLOAT(16)

DECFLOAT: scalar functions

- COMPARE_DECFLOAT(f1, f2)
  - returns:
    - 0   if f1 & f2 are “physically” equal
    - 1   if f1 < f2
    - 2   if f1 > f2
    - 3   if f1 & f2 are not comparable (f1 or f2 is NaN)

  e.g.   compare_decfloat('Inf', 3.141592) → 2
- TOTALORDER(f1, f2) → use: ORDER BY & WHERE
  - returns:
    - 0   if f1 & f2 are “logically” equal
    - -1   if f1 < f2 (where 'Inf' < 'NaN')
    - 1   if f1 > f2

e.g.   totalorder(decfloat('3.1415920'), 3.141592) → -1

string handling operators

“units”: OCTETS or CODEUNITS32

(*meaning: in bytes or in characters*)

no units allowed when s is BINARY

- CHARACTER_LENGTH(s, units)
  - cf. LENGTH(s): in bytes!
- LOCATE_IN_STRING(s,patt [,pos[,n], units])
  - cf. POSSTR(s, patt), POSITION(patt, s[, units]), LOCATE(patt, s [, startpos] [, units])
- SUBSTRING(s, startpos [, len], units)
  - cf. SUBSTR(s, startpos [, len])

string handling operators

SQL ANSI/ISO standard functions:

- CHARACTER_LENGTH(s [USING units])

OCTET_LENGTH(s)

- POSITION(patt IN s [USING units])

SUBSTRING(s FROM startpos [FOR len] [USING units])

where "units" is OCTETS or CHARACTERS)

See other presentation at this IDUG:

A08 "Functioning with DB2" (Chris Crone)

string handling operators

- LPAD(s, n [, pad-char])
  - prepend s with n spaces
- RPAD(s, n [, pad-char])
  - append s with n spaces
- INSERT(s, pos, len, repl [, units]) [already in v8]
  - at start position *pos*, replace *len* bytes by *repl*
- OVERLAY(s, repl, pos[, len], units)
- OVERLAY(s PLACING repl FROM pos [FOR len] [USING units])

encoding-related functions

- UNICODE(c)
  - returns the Unicode “code point” (int) of char c
- ASCII(c) [already in DB2 v8 for LUW]
  - returns the ordinal position of char c in ASCII
- ASCII_CHR(n) CHR(n)
  - returns the character at position n in ASCII
- EBCDIC_CHR(n)
  - returns the character at position n in EBCDIC
- UNICODE_STR(s)
  - returns the Unicode “translation” of string s
- ASCII_STR(s), EBCDIC_STR(s)

cultural sort

- COLLATION_KEY(s, collation_name)
  - to be used for “locale” support (when sorting)
  - return value only useful in mutual compare
  - example:

```
SELECT name
FROM   clients
WHERE  COLLATION_KEY(name,'UCA400R1_AS_LNL_S1_NO')
       BETWEEN COLLATION_KEY('VAAA','UCA400R1_AS_LNL_S1_NO')
           AND COLLATION_KEY('VZZZ','UCA400R1_AS_LNL_S1_NO')
ORDER BY COLLATION_KEY(name,'UCA400R1_AS_LNL_S1_NO')
```

`A: punctuation; L: locale; S: case&accents; N: normalisation`

  - UCA collation_name: encodes elements like

case-(in)sensitive; ignore-whitespace; ignore-accents; ignore-punctuation; country-specific alphabet; ...

http://www.ibm.com/developerworks/db2/library/techarticle/dm-0602doole/

SOUNDEX and related

- SOUNDEX(s) [already in DB2 v8 for LUW]
  - “sounds as”
  - returns a 4-character code (1 letter, 3 digits)
  - equal soundex-code: similar pronunciation
- DIFFERENCE(s1,s2) [already in DB2 v8 for LUW]
  - similarity of soundex-codes (0, 1, 2, 3, or 4)
    4: very similar
    0: very dissimilar
- See presentations at earlier IDUG

e.g. “V9 SQL for the Application Developer and DBA – Richer and Faster!”,
Suresh Sane, IDUG October 2008, Warsaw.

Unicode related

- NORMALIZE_STRING(s, form)
  - s: Unicode string
  - form: one of NFC, NFD, NFKC, NFKD

NFD: canonical decomposition

example: é → e + ´

NFC: canonical decomposition + composition

example: é → e + ´ → é

NFKD: compatibility decomposition

NFKC: compat. decomposition + composition

See other presentation at this IDUG:

E12 “I need Unicode - now what?” (Chris Crone)

Date & time related

- EXTRACT(year FROM dt) [already in DB2 v8]
  - this is the new SQL standard; replaces year()
  - also: “month”, “day”, “hour”, “minute”, “second”
- TIMESTAMPADD(interval-unit, n, ts)
  - *interval-unit:* 1: µs; 2: s; 4: min; 8: h; ...; 256: y
  - cf. TIMESTAMPDIFF(interval-unit, ts1-ts2)
- VARCHAR_FORMAT(ts, format)
  - Format: e.g. `'DD/MM/YYYY HH24:MI:SS'`
  - Alias (LUW only): TO_CHAR()
- TIMESTAMP_FORMAT(string, format)
  - Alias (LUW only): TO_DATE()

new statistical functionality

- important for e.g. Data Warehousing
- statistical analysis of data
  - data aggregation:

new aggregate functions

  - ranking:

rank() over (order by ...)

  - windowing:

sum() over (...)

aggregate functions

- COUNT, SUM, AVG, MIN, MAX
  - already from day 1...
- STDDEV(col), STDDEV_SAMP(col)
  - standard deviation (already in v8)
- VARIANCE, VARIANCE_SAMP
  - square of stddev (already in v8)
- CORRELATION(col1, col2)
  - between -1 and 1
- COVARIANCE, COVARIANCE_SAMP
- XMLAGG(xmlexpr ORDER BY expr)
  - aggregate concat! (already in v8)

conclusions

- new datatypes: blessing or curse?
  ==> **opportunity**!
- text: CHAR / VARCHAR / BINARY
  ==> make the right choice
  ==> careful with encoding
  ==> Unicode
- numeric: consider using DECFLOAT
  ==> where appropriate
  ==> INT / DECIMAL / FLOAT
- new functions: blessing or curse?
  ==> **standardisation**!

F17

DB2 9 New Datatypes and SQL Functions: Blessing or Curse?

Peter Vanroose

ABIS TRAINING & CONSULTING

pvanroose@abis.be

http://www.abis.be/