# La base de données Unicode en SQL

Stéphane Bortzmeyer
<stephane+blog@bortzmeyer.org>

Première rédaction de cet article le 7 septembre 2007. Dernière mise à jour le 30 octobre 2007

https://www.bortzmeyer.org/unicode-to-sql.html

---

Unicode est une norme de classification des caractères utilisés pour écrire des textes. L'imagination des humains dans ce domaine étant sans limites, Unicode est une norme de taille importante, et compliquée. Le cœur de la norme Unicode étant une liste de tous les caractères, pour toutes les écritures du monde, mettre cette liste dans une base de données relationnelle simplifie son accès, grâce au langage SQL.

Le consortium Unicode, qui gère la norme, la publie gratuitement sur le réseau sous le forme d'un ensemble de fichiers texte disponibles en <http://www.unicode.org/Public/UNIDATA/>. La documentation de ces fichiers est disponible dans le document Unicode Character Database <http://www.unicode.org/Public/UNIDATA/UCD.html> mais il est incomplet et pas mal de points nécessitent de lire la norme elle-même (disponible sur <http://www.unicode.org/standard/standard.html> mais hélas uniquement en PDF avec quelques menottes numériques ; c'est quand même nettement mieux qu'à l'ISO). Une bonne description de cette base se trouve également dans le livre de Jukka Korpela, "*Unicode explained*" <https://www.bortzmeyer.org/unicode-explained.html>.

Armé de ces fichiers texte et d'un petit programme (en ligne sur https://www.bortzmeyer.org/files/ucd2sql.lua) écrit en Lua, nous pouvons créer notre base. Détaillons les étapes.

Le schéma SQL n'est pas évident à concevoir, vue la richesse et le caractère parfois un peu désordonné d'Unicode. Comme il n'existe pas de normes ici, voici le schéma (très incomplet) que je propose (testé avec PostgreSQL, le SGBD que j'utilise) :

```
CREATE TABLE BidiClasses (id SERIAL PRIMARY KEY,
name TEXT NOT NULL UNIQUE,
        description TEXT);

CREATE TABLE Categories (id SERIAL PRIMARY KEY,
name TEXT NOT NULL UNIQUE,
        description TEXT);
```

```
CREATE TABLE Properties (id SERIAL PRIMARY KEY,
shorthand TEXT NOT NULL UNIQUE,
description TEXT NOT NULL UNIQUE);

CREATE TABLE Blocks (id SERIAL PRIMARY KEY,
name TEXT NOT NULL UNIQUE);

CREATE TABLE Han_Properties (id SERIAL PRIMARY KEY,
codepoint INTEGER,
Definition TEXT,
TotalStrokes INTEGER);
-- TODO: other properties of Unihan

CREATE TABLE Characters (id SERIAL PRIMARY KEY,
codepoint INTEGER NOT NULL UNIQUE,
        name TEXT NOT NULL, -- Warning: unlike what some people say, Unicode names
        -- are not UNIQUE (<control> characters, for instance, have no unique name)
        version TEXT, -- Version of Unicode where it was added
-- TODO: this is only Simple Casing. We must also handle SpecialCasing
-- where the uppercase can be a string. Use PotgreSQL's arrays?
        -- The REFERENCES Characters(codepoint) is commented out because
-- it is painful to enforce it (forward references in UnicodeData.txt
uppercase INTEGER, -- REFERENCES Characters(codepoint),
lowercase INTEGER, -- REFERENCES Characters(codepoint),
titlecase INTEGER, -- REFERENCES Characters(codepoint),
decomposition_type TEXT, -- If NULL, canonical decomposition. Otherwise,
-- indicates the type of the compatibility decomposition.
decomposition INTEGER[], -- May be an empty array if no decomposition.
-- TODO: is evaluating the length of an array in PostgreSQL fast? Otherwise,
        -- it would be better to have a boolean telling us if there is a decomposition.
        category TEXT REFERENCES Categories(name),
bidiclass TEXT REFERENCES BidiCLasses(name));

CREATE TABLE Characters_Properties (id SERIAL PRIMARY KEY,
codepoint INTEGER,
        -- TODO: This schema works only for boolean properties
property TEXT REFERENCES Properties(description));

-- Displays a codepoint in standard Unicode notation
-- TODO: pad with leading zeroes
CREATE FUNCTION To_U(INTEGER) RETURNS TEXT AS
    'SELECT ''U+'' || Upper(To_hex($1))'
    LANGUAGE 'SQL';

-- Be careful, it returns NULL, not 0, is there is no decomposition (I did not find a better way
-- in pure SQL)
CREATE FUNCTION Length_decomposition(INTEGER) RETURNS INTEGER AS
    'SELECT (array_upper(decomposition, 1) - array_lower(decomposition, 1)) + 1 FROM Characters WHERE codepo
    LANGUAGE 'SQL';

-- Complicated, yes. See http://www.varlena.com/GeneralBits/104.php
-- That's why it's in comments. Better to use something like:
--     SELECT * FROM Characters WHERE codepoint = x'0DC7'::INTEGER;
--CREATE FUNCTION To_D(TEXT) RETURNS INTEGER AS
--    $$
--    DECLARE
--       r RECORD;
--     BEGIN
--       FOR r IN EXECUTE 'SELECT x'''||$1||'''::integer AS hex' LOOP
--         RETURN r.hex;
--       END LOOP;
--     END
--    $$
--    LANGUAGE 'PLPGSQL' IMMUTABLE STRICT;
```

La table la plus importante est `Characters`, qui stockera les informations situées dans le fichier `UnicodeData.txt`. La fonction `To_U` est utile pour afficher les **points de code** Unicode (les index dans

la table Unicode), traditionnellement affichés en hexadécimal (par exemple, U+0152 pour le caractère « grand e dans l'o », [Caractère Unicode non montré [1] ]).

On peut alors créer le base, en encodage UTF-8, car certains fichiers contiennent de l'UTF-8 (PostgreSQL peut gérer de l'Unicode <https://www.bortzmeyer.org/postgresql-unicode.html>) :

```
createdb --encoding UTF-8 ucd
psql -f create.sql ucd
```

Ensuite, on fait tourner le programme Lua (disponible ici (en ligne sur https://www.bortzmeyer.org/files/ucd2sql.lua)) :

```
lua ucd2sql.lua /usr/share/unicode > ucd.sql
```

et on obtient un fichier de commandes SQL (INSERT, UPDATE, etc) qu'il n'y a plus qu'à soumettre à PostgreSQL :

```
psql -f ucd.sql ucd
```

Une fois la base chargée (cela prend du temps, Unicode est gros et le code SQL généré pas forcément très efficace), on peut se livrer à des études amusantes, bien plus faciles grâce à SQL. Les essais ci-dessous ont été faits avec les fichiers de la version 5.0.0 d'Unicode (mais le programme continue de marcher avec les versions ultérieures <https://www.bortzmeyer.org/unicode-6-0.html>).

Combien y a t-il de caractères dans Unicode ?

```
ucd=> SELECT count(*) AS Total FROM Characters;
 total
-------
 99089
```

À noter que le chiffre obtenu ne prend pas en compte les points de code réservés pour un usage privé ou bien les seizets d'indirection ("*surrogates*"). Cela donne une idée de la taille d'Unicode mais il faut se rappeler que les trois quarts sont des caractères Han :

```
ucd=> SELECT count(*) AS Total FROM Characters WHERE name LIKE 'CJK%';
 total
-------
 71357
```

1. Car trop difficile à faire afficher par LaTeX

On trouvera un autre comptage (pour une version plus ancienne d'Unicode) en "*The Number Of Characters In Unicode*" <http://www.i18nguy.com/unicode/char-count.html> ou bien en Break down of character statistics by Unicode version <http://babelstone.blogspot.com/2005/11/how-many-unicode-characters-are-there.html>.

Quand les caractères ont-ils été ajoutés à Unicode ?

```
ucd=> SELECT version,count(version) FROM Characters GROUP BY version ORDER BY version;
 version | count
---------+-------
 1.1     | 27577
 2.0     | 11373
 2.1     |     2
 3.0     | 10307
 3.1     | 44946
 3.2     |  1016
 4.0     |  1226
 4.1     |  1273
 5.0     |  1369
```

Le saut de la 3.1 (mai 2001) est dû à la première échappée hors du **Plan Multilingue de Base**, qui contenait uniquement les 65536 premiers caractères. Dépasser les 16 bits de ce plan a permis d'ajouter de nombreux caractères chinois. Tiens, quels sont les deux seuls caractères ajoutés dans la version 2.1, en novembre 1999 ?

```
ucd=> SELECT To_U(codepoint) AS Codepoint, name FROM Characters WHERE version = '2.1';
 codepoint |             name
-----------+------------------------------
 U+20AC    | EURO SIGN
 U+FFFC    | OBJECT REPLACEMENT CHARACTER
```

On peut aussi se servir de cette base pour extraire facilement tous les caractères d'une écriture donnée, ici le Tifinagh :

```
ucd=> SELECT To_U(codepoint) AS U_Codepoint, name FROM Characters WHERE name LIKE 'TIFINAGH %' ORDER BY code
 u_codepoint |                    name
-------------+-------------------------------------------
 U+2D30      | TIFINAGH LETTER YA
 U+2D31      | TIFINAGH LETTER YAB
 U+2D32      | TIFINAGH LETTER YABH
 U+2D33      | TIFINAGH LETTER YAG
 U+2D34      | TIFINAGH LETTER YAGHH
 U+2D35      | TIFINAGH LETTER BERBER ACADEMY YAJ
 U+2D36      | TIFINAGH LETTER YAJ
 U+2D37      | TIFINAGH LETTER YAD
 U+2D38      | TIFINAGH LETTER YADH
...
```

On peut aussi trouver la majuscule d'un caractère :

```
ucd=> SELECT To_U(r.codepoint), r.name
ucd->    FROM Characters r, Characters l
ucd->    WHERE r.codepoint = l.uppercase AND l.codepoint = x'61'::INTEGER;
 to_u |          name
------+------------------------
 U+41 | LATIN CAPITAL LETTER A
```

La majuscule de 'a' est 'A'. Évidemment, on n'avait pas besoin d'une base de données pour cela. Cette lettre fait partie du jeu ASCII, pour lequel on peut passer de la minuscule à la majuscule uniquement en retirant 32 au point de code. Mais Unicode est bien plus large et compliqué qu'ASCII et un algorithme aussi trivial ne marche pas dans tous les cas. Par exemple :

```
ucd=> SELECT To_U(r.codepoint), r.name AS Uppercase,
             To_U(l.codepoint), l.name AS Lowercase
         FROM Characters r, Characters l
         WHERE r.codepoint = l.uppercase AND
               l.name LIKE 'GREEK%SIGMA';
 to_u  |         uppercase          | to_u  |           lowercase
-------+----------------------------+-------+--------------------------------
 U+3A3 | GREEK CAPITAL LETTER SIGMA | U+3C2 | GREEK SMALL LETTER FINAL SIGMA
 U+3A3 | GREEK CAPITAL LETTER SIGMA | U+3C3 | GREEK SMALL LETTER SIGMA
(2 rows)
```

En effet, [Caractère Unicode non montré ] et [Caractère Unicode non montré ] ont la même majuscule, [Caractère Unicode non montré ].

Et tous les caractères ayant la propriété « Est un caractère d'espacement » ?

```
ucd=> SELECT To_U(Characters.codepoint) AS Codepoint,substr(name,1,40) AS Name,property FROM Characters, Charact
codepoint |           name             |  property
----------+----------------------------+-------------
   U+9    | <control>                  | White_Space
   U+A    | <control>                  | White_Space
   U+B    | <control>                  | White_Space
   U+C    | <control>                  | White_Space
   U+D    | <control>                  | White_Space
   U+20   | SPACE                      | White_Space
   U+85   | <control>                  | White_Space
   U+A0   | NO-BREAK SPACE             | White_Space
   U+1680 | OGHAM SPACE MARK           | White_Space
   U+180E | MONGOLIAN VOWEL SEPARATOR  | White_Space
   U+2000 | EN QUAD                    | White_Space
   U+2001 | EM QUAD                    | White_Space
   U+2002 | EN SPACE                   | White_Space
   U+2003 | EM SPACE                   | White_Space
   U+2004 | THREE-PER-EM SPACE         | White_Space
   U+2005 | FOUR-PER-EM SPACE          | White_Space
   U+2006 | SIX-PER-EM SPACE           | White_Space
   U+2007 | FIGURE SPACE               | White_Space
   U+2008 | PUNCTUATION SPACE          | White_Space
   U+2009 | THIN SPACE                 | White_Space
   U+200A | HAIR SPACE                 | White_Space
   U+2028 | LINE SEPARATOR             | White_Space
   U+2029 | PARAGRAPH SEPARATOR        | White_Space
   U+202F | NARROW NO-BREAK SPACE      | White_Space
   U+205F | MEDIUM MATHEMATICAL SPACE  | White_Space
   U+3000 | IDEOGRAPHIC SPACE          | White_Space
```

Cette base donne également accès à certaines propriétés des caractères, par exemple leur **directionnalité** (de droite à gauche ou de gauche à droite, sans compter les nombreuses variantes) :

---

https://www.bortzmeyer.org/unicode-to-sql.html

```
ucd=> SELECT count(codepoint) AS codepoint,Bidiclasses.description
ucd->    FROM Characters, Bidiclasses WHERE Bidiclasses.name = Characters.bidiclass
ucd->    GROUP BY bidiclass, Bidiclasses.description ORDER BY count(codepoint) DESC;
 codepoint |       description
-----------+-------------------------
     93356 | Left_To_Right
      3148 | Other_Neutral
      1010 | Arabic_Letter
       892 | Nonspacing_Mark
       260 | Right_To_Left
       176 | Boundary_Neutral
       120 | European_Number
        55 | European_Terminator
        18 | White_Space
        15 | Common_Separator
        12 | European_Separator
        12 | Arabic_Number
         7 | Paragraph_Separator
         3 | Segment_Separator
         1 | Right_To_Left_Override
         1 | Left_To_Right_Embedding
         1 | Right_To_Left_Embedding
         1 | Left_To_Right_Override
         1 | Pop_Directional_Format
```

On y voit que la majorité des caractères appartiennent à des écritures allant de gauche à droite.

On peut aussi regarder la décomposition d'un caractère, ici le « e accent grave » (nous utilisons les tableaux - non standards - de PostgreSQL pour cela) :

```
ucd=> SELECT To_u(codepoint),name,decomposition FROM Characters WHERE codepoint=x'0E8'::INTEGER;
 to_u |              name               | decomposition
------+---------------------------------+---------------
 U+E8 | LATIN SMALL LETTER E WITH GRAVE | {101,768}
(1 row)
```

On voit que "è" se décompose en 101 (en décimal, donc U+0065) et 768 (U+0300, l'accent combinant). La décomposition en Unicode étant récursive, les algorithmes de canonicalisation comme NFC ou NFD ne peuvent pas être programmés en SQL. Voici donc un exemple de programme (en ligne sur `https://www.bortzmeyer.org/files/unicode-decompose.py`) écrit en Python qui utilise cette base de données pour mettre en œuvre un sous-ensemble de NFD ou NKFD. Il sert par exemple pour le RFC 5051 [2].

Et, pour terminer cet article, que sait-on sur les caractères chinois ? Combien décrivent une tortue ?

```
SELECT To_U(Characters.codepoint) AS UCodepoint, name, definition
   FROM Characters, Han_Properties WHERE
   Characters.codepoint = Han_properties.codepoint AND
   definition ILIKE '%turtle%' ORDER BY Characters.codepoint;
 ucodepoint |               name               |  definition
...
 U+4E80     | CJK IDEOGRAPH-4E80               | turtle or tortoise; cuckold
 U+872E     | CJK IDEOGRAPH-872E               | a fabulous creature like a turtle; a toad
 U+8835     | CJK IDEOGRAPH-8835               | large turtles
 U+9C32     | CJK IDEOGRAPH-9C32               | huge sea turtle
 U+9C49     | CJK IDEOGRAPH-9C49               | turtle
...
[29 en tout]
```

2. Pour voir le RFC de numéro NNN, `https://www.ietf.org/rfc/rfcNNN.txt`, par exemple `https://www.ietf.org/rfc/rfc5051.txt`

Une version XML de la base Unicode est, depuis peu, disponible au même endroit, dans les répertoires nommés `ucdxml`. Elle est documentée dans l'annexe 42 <http://www.unicode.org/reports/tr42/>. Le caractère [Caractère Unicode non montré ] est noté :

```
<char cp="0100" age="1.1" na="LATIN CAPITAL LETTER A WITH MACRON" JSN="" gc="Lu" ccc="0" dt="can" dm="0041 030
```

Je ne l'ai pas encore essayée.

Si on veut explorer en détail la base Unicode, il existe d'autres alternatives, décrites dans l'article Naviguer dans Unicode <https://www.bortzmeyer.org/naviguer-dans-unicode.html>.