Créer ses propres types de données avec PostgreSQL

Stéphane Bortzmeyer
<stephane+blog@bortzmeyer.org>

Première rédaction de cet article le 18 juin 2010. Dernière mise à jour le 21 juin 2010

https://www.bortzmeyer.org/postgresql-creer-ses-types.html

Un des gros avantages du SGBD libre PostgreSQL est qu'on peut créer ses propres types de données. PostgreSQL a déjà une riche bibliothèque pré-définie <http://www.postgresql.org/docs/current/interactive/datatype.html>, incluant par exemple adresses IP, UUID et plein d'autres choses qui ont été très utiles à des programmes comme DNSwitness <http://www.dnswitness.net/>. Mais même cette vaste bibliothèque peut être insuffisante si on veut manipuler des objets d'un type un peu inhabituel. Dans ce cas, la solution est de créer ses propres types.

Commençons par un exemple très simple. Si on veut un type de données qui regroupe plusieurs valeurs <http://www.postgresql.org/docs/current/interactive/rowtypes.html> (un peu comme un `struct` en C), PostgreSQL dispose d'une instruction `CREATE TYPE`. J'emprunte à Xiaochun Wu un exemple :

```
CREATE TYPE lifetime AS (start_time TIMESTAMP, end_time TIMESTAMP);
CREATE TABLE device (id SERIAL UNIQUE NOT NULL, name TEXT, life lifetime);
...
INSERT INTO device (name, life) VALUES ('Foo', ('2006-01-01 12:34', now()));
INSERT INTO device (name, life) VALUES ('Bar', ('2008-03-05', '2010-02-01 12:00:00'));
...
SELECT * FROM device;
 id | name |                          life
----+------+-------------------------------------------------------
  1 | Foo  | ("2006-01-01 12:34:00","2009-07-28 16:21:00.544521")
  2 | Bar  | ("2008-03-05 00:00:00","2010-02-01 12:00:00")
...
SELECT * FROM device WHERE (life).end_time > '2010-01-01';
 id | name |                      life
----+------+-----------------------------------------------
  2 | Bar  | ("2008-03-05 00:00:00","2010-02-01 12:00:00")
```

(Notez les parenthèses autour de `life` pour éviter la confusion avec le nom d'une table.) Ainsi, on peut avoir des enregistrements en PostgreSQL. Ce n'est pas forcément une bonne idée, on peut penser que cela viole la première forme normale. Et je leur trouve des limites, par exemple je ne crois pas qu'il soit possible de mettre des contraintes d'intégrité comme le fait que `end_time ¿= start_time`. Mais de toute façon, dans cet article, je vais parler de types plus complexes, créés en programmant le serveur en C. **Un avertissement, toutefois** : C est un langage de bas niveau et relier le SGBD PostgreSQL à une bibliothèque que vous avez programmée dans ce langage, qui offre peu de garanties, peut entrainer des problèmes plus ou moins sérieux (allant jusqu'à la corruption de données, comme cela m'est arrivé au cours d'un essai avec PostgreSQL 8.3). Donc, à ne pas utiliser sur une base de production avant des tests détaillés et des sauvegardes sérieuses.

Le premier exemple va être très classique, c'est celui d'un type pour les nombres rationnels. Ces nombres sont ceux qui peuvent s'écrire sous forme de fraction de deux nombres entiers, par exemple 2/3 ou 1/5. On veut pouvoir entrer de tels nombres dans la base et que, lors de l'affichage, les deux termes de part et d'autre de la barre de fraction soient réduits par le PGCD. Suivons les instructions de la documentation <http://www.postgresql.org/docs/current/interactive/xtypes.html>.

D'abord, je développe une bibliothèque indépendante de PostgreSQL, avec des outils classiques (il existe certainement des bibliothèques de gestion des rationnels toutes faites mais c'est juste un exemple). J'écrirai ensuite le peu de code nécessaire pour que ce type de données soit ajouté à PostgreSQL. Pour la réduction avec le PGCD, j'emprunte un code C à Wikipédia, qui fournit la fonction `gcd` utilisée plus loin. J'écris un en-tête C qui donne l'API de ma petite bibliothèque :

```
typedef struct
{
  unsigned long numerator, denominator;
} rational;
...
bool are_equal (rational left, rational right);
bool is_lt (rational left, rational right);
...
/* Input-Output functions */
rational text_to_rational (const char *textr);
...
```

Et la mise en œuvre est plutôt simple, sauf pour les entrées/sorties :

```
bool
are_equal(rational left, rational right)
{
    return ((left.numerator == right.numerator) &&
            (left.denominator == right.denominator));
}
...
void
_canonicalize(rational * r)
{
    long            denom = gcd(r->numerator, r->denominator);
    if (denom != 1) {
        r->numerator = r->numerator / denom;
        r->denominator = r->denominator / denom;
    }
}
...
rational
text_to_rational(const char *textr)
{
    rational        r;
```

```
    char           *slash;
    unsigned int    i;
    INIT(r);
    slash = index(textr, '/');
    if (slash == NULL) {
        r.denominator = 1;
        for (i = 0; i < strlen(textr); i++) {
            if (!isdigit(textr[i])) {
                INIT(r);
                return (r);
            }
        }
        r.numerator = atol(textr);
...
    _canonicalize(&r);
    return r;
}
```

Notez la fonction de canonicalisation, qui réduit numérateur et dénominateur par le PGCD dès la conversion en rationnel.

Maintenant, la bibliothèque indépendante de PostgreSQL est écrite, on peut la tester. j'écris un petit programme de test, `test_rational_lib.c` et une règle dans le `Makefile` pour exécuter les tests :

```
% make test_lib
Tests that should succeed
./test_rational_lib 1 2
Working with 1 and 2
The two numbers are different
The first is smaller
./test_rational_lib 1/3 2/5
Working with 1/3 and 2/5
The two numbers are different
The first is smaller
./test_rational_lib 16/24 2/4
Working with 2/3 and 1/2
The two numbers are different
The first is larger
Tests that should fail
./test_rational_lib a 1 || true
Invalid syntax for rational number in "a"
./test_rational_lib 2.3 1 || true
Invalid syntax for rational number in "2.3"
```

Parfait, les syntaxes incorrectes comme "a" ou "2.3" sont bien rejetées. Et la canonicalisation fonctionne bien (voyez comment 16/24 a été canonicalisé en 2/3).

Maintenant, il faut permettre à PostgreSQL d'utiliser cette bibliothèque. Cela nécessite, suivant la documentation <http://www.postgresql.org/docs/current/interactive/xtypes.html>, des fonctions en C <http://www.postgresql.org/docs/current/interactive/xfunc-c.html> et pas mal de "*boilerplate*" (l'ancienne version de l'interface à PostgreSQL était plus simple de ce point de vue). Par exemple, la fonction d'égalité devient :

```
PG_FUNCTION_INFO_V1(rational_eq);
Datum
rational_eq(PG_FUNCTION_ARGS)
{
    bool            result;
    result =
        are_equal(*((rational *) PG_GETARG_POINTER(0)),
                  *((rational *) PG_GETARG_POINTER(0)));
    PG_RETURN_BOOL(result);
};
```

et appelle la `are_equal` de notre bibliothèque. Notez la manière compliquée (mais très générale) de récupérer les paramètres avec `PG_GETARG_POINTER` et le fait qu'il faille tout convertir en pointeurs.

Les fonctions d'entrée/sortie sont plus complexes car elles peuvent échouer (par exemple si le format d'entrée est incorrect, ce qui appelera `ereport`) et elles nécessitent de gérer la mémoire (d'où l'utilisation de la macro `MALLOC`, qui va être pointée vers un `malloc` spécifique à PostgreSQL, `palloc`) :

```
PG_FUNCTION_INFO_V1(rational_in);
Datum
rational_in(PG_FUNCTION_ARGS)
{
    char           *str = PG_GETARG_CSTRING(0);
    rational       *result, val;
    result = MALLOC(sizeof(rational));
    val = text_to_rational(str);
    if (!VALID(val)) {
        ereport(ERROR, (errcode(ERRCODE_INVALID_TEXT_REPRESENTATION),
                        errmsg("rational_in: invalid RATIONAL \"%s\"", str)));
    }
    memcpy(result, &val, sizeof(rational));
    PG_RETURN_POINTER(result);
}
```

Il faut aussi le code SQL qui va installer ce type (on va utiliser le `CREATE TYPE` vu précédemment) et ces fonctions :

```
CREATE TYPE rational (
        internallength = TYPE_LENGTH,
        input = rational_in,
        output = rational_out
);
...
CREATE OR REPLACE function rational_eq(rational, rational)
        returns bool
        as 'MODULE_PATHNAME'
        language 'c';
...
CREATE OPERATOR = (
        leftarg = rational,
        rightarg = rational,
        commutator = =,
        procedure = rational_eq
);
...
CREATE OR REPLACE function rational_in(cstring)
        returns rational
        as 'MODULE_PATHNAME'
        language 'c';
...
```

Chaque fonction accessible depuis SQL devra être ainsi déclarée. Même chose pour les opérateurs comme celui d'égalité déclaré ci-dessus.

`MODULE_PATHNAME` et `TYPE_LENGTH` seront remplacés, via sed, au moment de la compilation.

Il reste à compiler la bibliothèque sous forme d'une bibliothèque dynamique. La partie pertinente du `Makefile` est :

```
INCLUDES_PGSQL=$(shell pg_config --includedir-server)
LIBRARIES=rational.o gcd.o
MALLOC=palloc
CC_SHARED_OPTIONS=-fpic -shared
...
rational.so: rational-glue.c ${LIBRARIES}
        ${CC} ${CFLAGS} ${MALLOCFLAGS} -I${INCLUDES_PGSQL} \
                ${CC_SHARED_OPTIONS} ${LIBRARIES} -o $@ $<
```

Une fois la compilation de `rationale.so` effectuée :

```
% make rational.so
gcc -Wall -Wextra -g -O0  -c \
                 -o rational.o rational.c
gcc -Wall -Wextra -g -O0  -c -o gcd.o gcd.c
gcc -Wall -Wextra -g -O0  -DMALLOC=palloc  -I/usr/include/postgresql/8.3/server  \
               -fpic -shared  rational.o gcd.o -o rational.so rational-glue.c
```

il faut copier ce fichier dans le répertoire où PostgreSQL le trouvera :

```
# cp rational.so $(pg_config --pkglibdir)
```

Et on peut alors lancer les commandes SQL d'installation :

```
% sudo -u postgres make DATABASE=essais install
```

Parfait, il ne reste plus qu'à essayer :

```
essais=> CREATE TABLE Data (name TEXT, value rational);
CREATE TABLE
essais=>
essais=> INSERT INTO Data VALUES ('Raw integer', '1');
INSERT 0 1
essais=> INSERT INTO Data VALUES ('Canonicalizes to a raw integer', '4/2');
INSERT 0 1
essais=> INSERT INTO Data VALUES ('Already canonical', '4/3');
INSERT 0 1
essais=> INSERT INTO Data VALUES ('Not yet canonical', '16/22');
INSERT 0 1
essais=> INSERT INTO Data VALUES ('Less than zero', '1/3');
INSERT 0 1
essais=> INSERT INTO Data VALUES ('Less than zero, not canonical', '2/6');
INSERT 0 1
essais=> -- Should fail
essais=> INSERT INTO Data VALUES ('Dot', '1.3');
ERROR:  rational_in: invalid RATIONAL "1.3"
essais=> INSERT INTO Data VALUES ('Nothing after the slash', '2/');
ERROR:  rational_in: invalid RATIONAL "2"
essais=> INSERT INTO Data VALUES ('Letters', 'Test');
ERROR:  rational_in: invalid RATIONAL "Test"
essais=>
essais=> SELECT * FROM Data ORDER BY value;
              name               | value
---------------------------------+-------
 Less than zero                  | 1/3
 Less than zero, not canonical   | 1/3
 Not yet canonical               | 8/11
 Raw integer                     | 1
 Already canonical               | 4/3
 Canonicalizes to a raw integer  | 2
(6 rows)
```

Et tout marche bien.

Une archive complète de tous les fichiers utilisés se trouve en (en ligne sur `https://www.bortzmeyer.org/files/postgresql-rationale.tar.gz`).

Voyons maintenant un type un peu plus compliqué, `domainname`, qui modélise un nom de domaine. Au lieu de simples entiers, on va utiliser des chaînes de caractères. Avec le protocole DNS, ces noms sont insensibles à la casse. On voudrait pouvoir disposer de fonctions comme extraire le TLD ou trouver le nombre de composants. Voici quelques exemples de l'usage qu'on veut faire :

```
test=> CREATE TABLE Registry (id SERIAL UNIQUE NOT NULL,
     created TIMESTAMP NOT NULL DEFAULT now(),
     fqdn Domainname UNIQUE NOT NULL);
test=> INSERT INTO Registry (fqdn) VALUES ('example.net');
INSERT 0 1
test=> INSERT INTO Registry (fqdn) VALUES ('bortzmeyer.fr');
INSERT 0 1
test=> INSERT INTO Registry (fqdn) VALUES ('dnsmezzo.fr');
INSERT 0 1
test=> INSERT INTO Registry (fqdn) VALUES ('www.foobar.example');
INSERT 0 1
test=> SELECT * FROM Registry WHERE dn_tld(fqdn) = 'fr' ORDER BY fqdn;
 id |          created           |     fqdn
----+----------------------------+---------------
  2 | 2010-06-18 22:45:05.061786 | bortzmeyer.fr
  3 | 2010-06-18 22:45:09.542158 | dnsmezzo.fr
(2 rows)
test=> SELECT fqdn, length(fqdn), dn_nlabels(fqdn) AS nlabels
     FROM Registry
     ORDER BY fqdn;
        fqdn        | length | nlabels
--------------------+--------+---------
 bortzmeyer.fr      |     13 |       2
 dnsmezzo.fr        |     11 |       2
 example.net        |     11 |       2
 www.foobar.example |     18 |       3
(4 rows)
```

Pour cela, il faut d'abord créer une structure de données C :

```
typedef struct {
    unsigned short  _nlabels;
    size_t          _length;
    char            _orig_name[MAX_LENGTH];
    char            _lcase_name[MAX_LENGTH];
    char            _tld[MAX_LENGTH];
    char            _reg_domain[MAX_LENGTH];
} domainname;
```

Pour des raisons de performance (on sacrifie par contre de l'espace disque), on calculera toutes les valeurs utiles lors de l'insertion d'un nom. Ainsi, récupérer le TLD, par exemple, ne nécessitera pas de calculs :

```
char           *
tld(char *result, const domainname d)
{
    result[0] = '\0';
    strncat(result, d._tld, strlen(d._tld));
    return result;
}
```

Par contre, l'insertion d'un nom est évidemment plus complexe (fonction `text_to_domain()`).

La fonction `_reg_domain()` mérite un peu plus d'explications. L'idée est de trouver le nom enregistré auprès d'un registre. Par exemple, pour le nom `www.foobar.example`, le nom enregistré est sans doute `foobar.example`. Je dis « sans doute » car le TLD `.example` n'existe pas et on ne peut pas connaître sa politique d'enregistrement. Certains enregistrent au deuxième niveau (`.eu`, `.com`, etc), d'autres au troisième (`.uk`, `.jp`, etc), et d'autres encore à différents niveaux (`.fr`, etc). Les algorithmes triviaux comme « le domaine enregistré est formé des deux derniers composants » sont donc inappropriés. Une solution serait d'utiliser la liste (non-officielle et pas forcément à jour) <http://publicsuffix.org/> et de produire le code C automatiquement à partir de cette liste. Cela n'a pas encore été mis en œuvre, pour l'instant, la fonction `_reg_domain()` traite à la main le cas de `.fr` et applique l'algorithme trivial pour les autres.

Une fois la bibliothèque C compilée, on peut la tester :

```
% ./test-domain-name www.example.fr WWW.example.fr
The canonical version of "www.example.fr" is "www.example.fr"; its TLD is "fr".
        Its registered domain is "example.fr", it has 3 labels
www.example.fr and WWW.example.fr are equivalent domain names
www.example.fr and WWW.example.fr are NOT absolutely identical

% ./test-domain-name FOO.BAR.QUIZ.BAZ.EXAMPLE foobar.net
The canonical version of "FOO.BAR.QUIZ.BAZ.EXAMPLE" is "foo.bar.quiz.baz.example"; its TLD is "example".
        Its registered domain is "baz.example", it has 5 labels
FOO.BAR.QUIZ.BAZ.EXAMPLE and foobar.net are NOT equivalent domain names
FOO.BAR.QUIZ.BAZ.EXAMPLE and foobar.net are NOT absolutely identical
```

Reste à le faire utiliser par PostgreSQL. Pas de problème particulier, à part la question de l'espace de stockage utilisé :

```
CREATE TYPE domainname (
        internallength = 1032,
        input = domainname_in,
        output = domainname_out
);
```

La taille a été ici mise en dur dans le fichier (après calcul). Cela veut dire que tout nom de domaine, quelle que soit sa longueur, va consommer 1032 octets, ce qui est un gros gaspillage. Il existe des solutions en PostgreSQL pour manipuler des objets de taille variable mais qui sont plus complexes. Voir :

- La documentation sur le « toasting » <http://www.postgresql.org/docs/current/interactive/storage-toast.html>,
- Deux articles sur les types de longueur variable : « Utiliser le principe du type varlena <http://www.d-sites.com/2009/12/07/postgresql-utiliser-le-principe-du-type-varlena/> » et « parse_url <http://www.d-sites.com/projets/postgresql-parse_url/v2/> » (une bibliothèque d'analyse d'URL, qui utilise cette technique).

On peut ensuite définir les fonctions, une qui manipule les chaînes de caractère C et une de plus haut niveau qui manipule des `TEXT` de PostgreSQL :

```
CREATE OR REPLACE function domainname_tld(domainname)
        returns cstring
   as '/home/stephane/AFNIC/ReD/Devel/DNSwitness/DNSmezzo/domain-name-type/domain-name.so'
        language 'c';

CREATE OR REPLACE function dn_tld(domainname)
        RETURNS text
   as 'SELECT domainname_tld($1)::TEXT'
        LANGUAGE 'sql';
```

Peut-on utiliser des index sur ce type, si on enregistre beaucoup de noms de domaines et qu'on veut accélérer l'accès ? C'est possible, via les `OPERATOR CLASS`. On dit à PostgreSQL quels opérateurs vont avec le type (opérateurs maison ou non, symboles habituels ou pas) et lesquels l'index doit utiliser (on peut même faire des choses très complexes <http://www.postgresql.org/docs/current/interactive/xindex.html>). Ici, on crée une série d'opérateurs (je n'en montre que deux) et la classe :

```
CREATE OPERATOR > (
        leftarg = domainname,
        rightarg = domainname,
        commutator = >,
        negator = <=,
        procedure = domainname_gt
);

CREATE OPERATOR >= (
        leftarg = domainname,
        rightarg = domainname,
        commutator = >=,
        negator = <,
        procedure = domainname_ge
);

CREATE OPERATOR CLASS domainname_ops
    DEFAULT FOR TYPE domainname USING btree AS
        OPERATOR        1       < ,
        OPERATOR        2       <= ,
        OPERATOR        3       = ,
        OPERATOR        4       >= ,
        OPERATOR        5       > ,
        FUNCTION 1 domainname_cmp(domainname, domainname);
```

Ainsi, on peut désormais indexer les noms de domaine.

Et sur le résultat des fonctions, les index marcheront aussi ? Si la fonction était ordinaire, comme ci-dessus, PostgreSQL refuserait :

```
essais=> CREATE INDEX idx_tld ON Registry(dn_tld(fqdn));
ERROR:  functions in index expression must be marked IMMUTABLE
```

Mais, justement, en indiquant que la fonction est `IMMUTABLE`, cela marche :

```
CREATE OR REPLACE function dn_tld(domainname)
RETURNS text
   as 'SELECT domainname_tld($1)::TEXT'
LANGUAGE 'sql' IMMUTABLE;
```

Attention, PostgreSQL ne teste pas que la fonction est vraiment `IMMUTABLE` (c'est-à-dire qu'elle renvoie toujours la même valeur, pour un argument donné). Mais, ici, c'est bien le cas, et `EXPLAIN` va bien montrer que l'index est utilisé :

```
test=> EXPLAIN SELECT * FROM Registry WHERE dn_tld(fqdn) = 'as';
                                QUERY PLAN
--------------------------------------------------------------------------
 Index Scan using idx_tld on registry  (cost=0.25..8.52 rows=1 width=1044)
   Index Cond: (dn_tld(fqdn) = 'as'::text)
(2 rows)
```

Une archive complète de tous les fichiers utilisés se trouve en (en ligne sur `https://www.bortzmeyer.org/files/postgresql-domain-name-type.tar.gz`). Comme pour le code des nombres rationnels, tout a été testé sur Debian et NetBSD, avec PostgreSQL 8.3 et 9.0.

Merci à Dimitri Fontaine pour ses conseils et son code. Quelques articles parlant de ce sujet ou des logiciels l'appliquant :

— Bibliothèque mettant en œuvre un type « préfixe » pour la téléphonie <`http://prefix.projects.postgresql.org/`>, avec son code <`http://github.com/dimitri/prefix`>. Une documentation sur la technique utilisée <`http://wiki.postgresql.org/wiki/Image:Prato_2008_prefix.pdf`> est disponible.