Writing a user-defined type on PostgreSQL

Heikki Linnakangas

What is a data type?

- A data type encapsulates semantics and rules

Domains

- Easy to create
- Domain creates a “subtype” of an existing base type
- Shares operators with base type
- Can have additional constraints on what is accepted

```
CREATE DOMAIN countrycode AS text
CHECK (value ~ '^[a-z][a-z]$');
```

Enum types

- A list of values
- Typically used for states

```
CREATE TYPE mood AS ENUM ('sad', 'ok', 'happy');
```

Creating a new base type

- At a minimum you need
- Input and output functions
  - Converts from/to string to internal format
- Internal storage
  - fixed-size, or
  - variable length

Example

- Data type for representing colours
  - As a 24-bit RGB value
  - For convenience, stored in a 32-bit integer
  - String representation in hex:
    - #000000 – black
    - #FF0000 – red
    - #0000A0 – dark blue
    - #FFFFFF – white

Writing an extension in C

```
~/colour_type (master)$ ls -l
yhteensä 16
-rw-r--r-- 1 heikki heikki 919  3.2. 11:53 colour.c
-rw-r--r-- 1 heikki heikki 418  3.2. 11:47 colour.sql.in
-rw-r--r-- 1 heikki heikki 308  3.2. 11:31 Makefile
-rw-r--r-- 1 heikki heikki 118  3.2. 11:49 uninstall_colour.sql
```

I/O functions

```
/* colour_out */
PG_FUNCTION_INFO_V1(colour_out);
Datum
colour_out(PG_FUNCTION_ARGS)
{
    Int32   val = PG_GETARG_INT32(0);
    char   *result = palloc(8);

    snprintf(result, 8, "#%06X", val);
    PG_RETURN_CSTRING(result);
}
```

I/O functions

```
/* colour_in */
PG_FUNCTION_INFO_V1(colour_in);
Datum
colour_in(PG_FUNCTION_ARGS)
{
    const char *str = PG_GETARG_CSTRING(0);
    int32    result;

    if (str[0] != '#' || strspn(&str[1], "01234567890ABCDEF") != 6)
        ereport(ERROR,
                (errcode(ERRCODE_INVALID_TEXT_REPRESENTATION),
                 errmsg("invalid input syntax for colour: \"%s\"", str)));

    sscanf(str, "#%X", &result);
    PG_RETURN_INT32(result);
}
```

Register type with PostgreSQL

```
SET search_path = public;

CREATE OR REPLACE FUNCTION colour_in(cstring) RETURNS colour
AS 'MODULE_PATHNAME' LANGUAGE 'C' IMMUTABLE STRICT;

CREATE OR REPLACE FUNCTION colour_out(colour) RETURNS cstring
AS 'MODULE_PATHNAME' LANGUAGE 'C' IMMUTABLE STRICT;

CREATE TYPE colour (
INPUT = colour_in,
OUTPUT = colour_out,
LIKE = pg_catalog.int4
);
```

The type is ready!

```
postgres=# CREATE TABLE colour_names (
  name text,
  rgbvalue colour
);
CREATE TABLE
postgres=# INSERT INTO colour_names VALUES ('red', '#FF0000');
INSERT 0 1
postgres=# SELECT * FROM colour_names ;
 name | rgbvalue
------+----------
 red  | #FF0000
(1 row)
```

Operators

- A type needs operators
- Equality

```
postgres=# SELECT * FROM colour_names WHERE rgbvalue
= '#FF0000';
ERROR:  operator does not exist: colour = unknown
```

Equality operator

- We can borrow the implementation from built-in integer operator:

```
CREATE FUNCTION colour_eq (colour, colour)
RETURNS bool
LANGUAGE internal AS 'int4eq' IMMUTABLE;

CREATE OPERATOR = (
  PROCEDURE = colour_eq,
  LEFTARG = colour, RIGHTARG = colour,
  HASHES, MERGES);
```

Operators

- Ok, now it works:

```
postgres=# SELECT * FROM colour_names WHERE rgbvalue
= '#FF0000';
 name | rgbvalue
------+-----------
 red  | #FF0000
(1 row)
```

More functions

```
CREATE FUNCTION red(colour) RETURNS int4
LANGUAGE C AS 'MODULE_PATHNAME' IMMUTABLE;

CREATE FUNCTION green(colour) RETURNS int4
LANGUAGE C AS 'MODULE_PATHNAME' IMMUTABLE;

CREATE FUNCTION blue(colour) RETURNS int4
LANGUAGE C AS 'MODULE_PATHNAME' IMMUTABLE;
```

Luminence

```
CREATE FUNCTION luminence(colour)
RETURNS numeric AS
$$
SELECT 0.3 * red($1) + 0.59 * green($1) +
0.11 * blue($1)
$$
LANGUAGE SQL IMMUTABLE STRICT;
```

Summary so far

- We have created a type
  - With input and output functions
  - With equality operator
  - With functions for splitting a colour into components and calculating luminence

Ordering

```
postgres=# SELECT * FROM colour_names ORDER BY
rgbvalue;
ERROR:  could not identify an ordering operator
for type colour
LINE 1: SELECT * FROM colour_names ORDER BY
rgbvalue;
                                               ^

HINT:  Use an explicit ordering operator or
modify the query.
```

- We need to define an ordering operator!

Ordering operator

- What is an ordering operator?
  - <
  - <=
  - = (we already did this)
  - >=
  - >

- We're going to define order of colours in terms of luminence

Implementing functions

- CREATE FUNCTION **colour_lt** (colour, colour) RETURNS bool AS $$ SELECT luminence($1) **<** luminence($2); $$ LANGUAGE SQL IMMUTABLE;
- CREATE FUNCTION **colour_le** (colour, colour) RETURNS bool AS $$ SELECT luminence($1) **<=** luminence($2); $$ LANGUAGE SQL IMMUTABLE;
- CREATE FUNCTION **colour_ge** (colour, colour) RETURNS bool AS $$ SELECT luminence($1) **>=** luminence($2); $$ LANGUAGE SQL IMMUTABLE;
- CREATE FUNCTION **colour_gt** (colour, colour) RETURNS bool AS $$ SELECT luminence($1) **>** luminence($2); $$ LANGUAGE SQL IMMUTABLE;

Create operators

- CREATE OPERATOR < (LEFTARG=colour, RIGHTARG=colour, PROCEDURE=colour_lt);
- CREATE OPERATOR <= (LEFTARG=colour, RIGHTARG=colour, PROCEDURE=colour_le);
- CREATE OPERATOR >= (LEFTARG=colour, RIGHTARG=colour, PROCEDURE=colour_ge);
- CREATE OPERATOR > (LEFTARG=colour, RIGHTARG=colour, PROCEDURE=colour_gt);

One more thing..

- We'll also need a comparison function that returns -1, 0, or 1 depending on which argument is greater

```
CREATE FUNCTION luminence_cmp(colour, colour)
RETURNS integer AS $$
 SELECT CASE WHEN $1 = $2 THEN 0
  WHEN luminence($1) < luminence($2) THEN 1
    ELSE -1 END;
$$ LANGUAGE SQL IMMUTABLE;
```

Operator class

- Ok, we're ready to create an operator class!

```
CREATE OPERATOR CLASS luminence_ops
DEFAULT FOR TYPE colour USING btree AS
OPERATOR  1 <,
OPERATOR  2 <=,
OPERATOR  3 =,
OPERATOR  4 >=,
OPERATOR  5 >,
FUNCTION  1 luminence_cmp(colour, colour);
```

Ready to order!

```
postgres=# SELECT * FROM colour_names ORDER BY rgbvalue;
    name    | rgbvalue
------------+----------
 white      | #FFFFFF
 light grey | #C0C0C0
 lawn green | #87F717
 green      | #00FF00
 dark grey  | #808080
 red        | #FF0000
 blue       | #0000FF
 black      | #000000
(8 rows)
```

Indexing

- We already created a b-tree operator class

```
CREATE INDEX colour_lum_index ON colour_names (rgbvalue);

postgres=# explain  SELECT * FROM colour_names WHERE
rgbvalue = '#000000';
                              QUERY PLAN
-----------------------------------------------------------------------------
 Index Scan using colour_lum_index on colour_names
(cost=0.00..8.32 rows=4 width=36)
   Index Cond: (rgbvalue = '#000000'::colour)
(2 rows)
```

Plain ordering is dull

- Ordering by luminence is nice
- But what about finding a colour that's the closest match to given colour?

Distance function

```
CREATE FUNCTION colour_diff (colour, colour)
RETURNS float AS $$
SELECT sqrt((red($1) - red($2))^2 + (green($1) -
green($2))^2 + (blue($1) - blue($2))^2)
$$ LANGUAGE SQL;

CREATE OPERATOR <-> (PROCEDURE = colour_diff,
LEFTARG=colour, RIGHTARG=colour);
```

Using the distance operator

```
postgres=# SELECT * FROM colour_names ORDER BY rgbvalue <->
'#00FF00';
    name    | rgbvalue
------------+----------
 green      | #00FF00
 lawn green | #87F717
 dark grey  | #808080
 black      | #000000
 light grey | #C0C0C0
 white      | #FFFFFF
 blue       | #0000FF
 red        | #FF0000
(8 rows)
```

GiST

- Generalized Search Tree
- You have to write support functions
  - Like we did for b-tree

GiST support functions

EnterpriseDB®
The Enterprise Postgres Company

- GiST needs 8 support functions:
  - Consistent
  - Union
  - Compress
  - Decompress
  - Penalty
  - Picksplit
  - Same
  - Distance (optional)

GiST support functions

- Consistent
  - Is search key consistent with stored key
  - For example, does rectangle X contain point Y

GiST support functions

- Union
  - Given two keys, return a key that represents the union of the two keys
  - For example, given a rectangle and a point, return a new bounding box rectangle that contains both

GiST support functions

- Compress
  - Given an input key, compress it into compact form that can be stored on disk
- Decompress
  - The opposite

GiST support functions

- Penalty
  - How bad would it be to insert key X to page Y
  - Usually defined using a distance function

GiST support functions

- Picksplit
  - Given a bunch of keys, what's the best way to split them into two sets of roughly same size?

GiST support functions

- Same
  - = equals

GiST support functions

- Distance (optional)
  - How far is key X from key Y
  - For example, how far is a point from a bounding box
  - Enables k nearest neighbors search
  - New in PostgreSQL 9.1

GiST Summary

- Implement the support functions:
  - Consistent, Union, Compress, Decompress, Penalty, Picksplit, Same, Distance (optional)
- Handles
  - WAL-logging
  - Concurrency
  - Isolation
  - Durability
  - Transactions

GIN

- Generalized Inverted Index
- Splits input key into multiple parts, and indexes the parts
- For example
  - Full text search
  - Arrays
  - Word similarity (pg_trgm)

Wait, there's more!

- Binary I/O routines
- Casts
- Cross-datatype operators
- Hash function

Summary

- You're the expert in your problem domain
- You define the semantics
- PostgreSQL handles the rest