Writing a user-defined datatype

Heikki Linnakangas

VMware Inc.

October 30, 2013
What is a datatype?

A datatype encapsulates semantics and rules. PostgreSQL offers many built-in datatypes, e.g:

- integer
- text
- timestamp
- point

Other datatypes can be derived from the base types:

- domains
- arrays
- ranges
This presentation

PART 1

▶ Creating a new base type from scratch
▶ Define basic functions and operators
▶ B-tree indexing support

PART 2

▶ Advanced indexing
Creating a new base type

PostgreSQL stores data as opaque Datums

▶ Fixed or variable length (varlen) chunk of memory
▶ Can be copied around the system and stored on disk

All other operations are defined by the data type author. Minimum:

▶ Input and output functions. These convert between string representation and the internal format.
Example

A datatype 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 –
Datum
colour_in(PG_FUNCTION_ARGS)
{
    const char *str = PG_GETARG_CSTRING(0);
    int32 result;

    sscanf(str, "#%X", &result);
    PG_RETURN_INT32(result);
}
Input function, with error checking

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);
}
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);
}
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)
```
CREATE TYPE syntax

CREATE TYPE name (  
    INPUT = input_function,  
    OUTPUT = output_function  
    [ , RECEIVE = receive_function ]  
    [ , SEND = send_function ]  
    [ , TYPMOD_IN = type_modifier_input_function ]  
    [ , TYPMOD_OUT = type_modifier_output_function ]  
    [ , ANALYZE = analyze_function ]  
    [ , INTERNALLENGTH = \{ internallength | VARIABLE \} ]  
    [ , PASSEDBYVALUE ]  
    [ , ALIGNMENT = alignment ]  
    [ , STORAGE = storage ]  
    [ , LIKE = like_type ]  
    [ , CATEGORY = category ]  
    [ , PREFERRED = preferred ]  
    [ , DEFAULT = default ]  
    [ , ELEMENT = element ]  
    [ , DELIMITER = delimiter ]  
    [ , COLLATABLE = collatable ]
)
Operators

A type needs operators:

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:

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

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

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

CREATE FUNCTION blue(colour) RETURNS int4
    LANGUAGE C AS 'MODULE_PATHNAME' IMMUTABLE STRICT;
Extracting the components

postgres=# select name, rgbvalue, red(rgbvalue), green(rgbvalue), blue(rgbvalue) from colour_names;

<table>
<thead>
<tr>
<th>name</th>
<th>rgbvalue</th>
<th>red</th>
<th>green</th>
<th>blue</th>
</tr>
</thead>
<tbody>
<tr>
<td>red</td>
<td>#FF0000</td>
<td>255</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>green</td>
<td>#00FF00</td>
<td>0</td>
<td>255</td>
<td>0</td>
</tr>
<tr>
<td>blue</td>
<td>#0000FF</td>
<td>0</td>
<td>0</td>
<td>255</td>
</tr>
<tr>
<td>white</td>
<td>#FFFFFF</td>
<td>255</td>
<td>255</td>
<td>255</td>
</tr>
<tr>
<td>black</td>
<td>#000000</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>light grey</td>
<td>#C0C0C0</td>
<td>192</td>
<td>192</td>
<td>192</td>
</tr>
<tr>
<td>lawn green</td>
<td>#87F717</td>
<td>135</td>
<td>247</td>
<td>23</td>
</tr>
<tr>
<td>dark grey</td>
<td>#808080</td>
<td>128</td>
<td>128</td>
<td>128</td>
</tr>
</tbody>
</table>

(8 rows)
Luminence

The human eye is more sensitive to green light.

CREATE FUNCTION luminence(colour) RETURNS numeric AS
$$
SELECT (0.30 * red($1) +
      0.59 * green($1) +
      0.11 * blue($1))
   / 255.0
$$
LANGUAGE SQL IMMUTABLE STRICT;
Luminence

```sql
postgres=# select name, rgbvalue,
    red(rgbvalue), green(rgbvalue), blue(rgbvalue),
    round( luminence(rgbvalue), 6) as luminence
from colour_names ;

<table>
<thead>
<tr>
<th>name</th>
<th>rgbvalue</th>
<th>red</th>
<th>green</th>
<th>blue</th>
<th>luminence</th>
</tr>
</thead>
<tbody>
<tr>
<td>red</td>
<td>#FF0000</td>
<td>255</td>
<td>0</td>
<td>0</td>
<td>0.300000</td>
</tr>
<tr>
<td>green</td>
<td>#00FF00</td>
<td>0</td>
<td>255</td>
<td>0</td>
<td>0.590000</td>
</tr>
<tr>
<td>blue</td>
<td>#0000FF</td>
<td>0</td>
<td>0</td>
<td>255</td>
<td>0.110000</td>
</tr>
<tr>
<td>white</td>
<td>#FFFFFF</td>
<td>255</td>
<td>255</td>
<td>255</td>
<td>1.000000</td>
</tr>
<tr>
<td>black</td>
<td>#000000</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.000000</td>
</tr>
<tr>
<td>light grey</td>
<td>#C0C0C0</td>
<td>192</td>
<td>192</td>
<td>192</td>
<td>0.752941</td>
</tr>
<tr>
<td>lawn green</td>
<td>#87F717</td>
<td>135</td>
<td>247</td>
<td>23</td>
<td>0.740235</td>
</tr>
<tr>
<td>dark grey</td>
<td>#808080</td>
<td>128</td>
<td>128</td>
<td>128</td>
<td>0.501961</td>
</tr>
</tbody>
</table>
```

(8 rows)
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
postgres=# SELECT * FROM colour_names ORDER BY rgbvalue;
ERROR: could not identify an ordering operator for type colour
Ordering operator

What is an ordering operator?

- `<`
- `<=`
- `=` (we already did this)
- `>=`
- `>`

We’re going define these in terms of luminence
CREATE FUNCTION colour_lt (colour, colour) 
RETURNS bool AS $$
    SELECT luminence($1) < luminence($2);
$$ LANGUAGE SQL IMMUTABLE STRICT;
Implementing ordering functions

CREATE FUNCTION colour_le (colour, colour) RETURNS bool AS $$
    SELECT luminence($1) <= luminence($2);
$$ LANGUAGE SQL IMMUTABLE STRICT;

CREATE FUNCTION colour_ge (colour, colour) RETURNS bool AS $$
    SELECT luminence($1) >= luminence($2);
$$ LANGUAGE SQL IMMUTABLE STRICT;

CREATE FUNCTION colour_gt (colour, colour) RETURNS bool AS $$
    SELECT luminence($1) > luminence($2);
$$ LANGUAGE SQL IMMUTABLE STRICT;
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;
An operator class ties the individual operators together. Operator classes are defined for indexing support, but the B-tree operator class is a bit special.

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);
### Query Result:

```sql
postgres=# SELECT * FROM colour_names ORDER BY rgbvalue;

<table>
<thead>
<tr>
<th>name</th>
<th>rgbvalue</th>
</tr>
</thead>
<tbody>
<tr>
<td>white</td>
<td>#FFFFFF</td>
</tr>
<tr>
<td>light grey</td>
<td>#C0C0C0</td>
</tr>
<tr>
<td>lawn green</td>
<td>#87F717</td>
</tr>
<tr>
<td>green</td>
<td>#00FF00</td>
</tr>
<tr>
<td>dark grey</td>
<td>#808080</td>
</tr>
<tr>
<td>red</td>
<td>#FF0000</td>
</tr>
<tr>
<td>blue</td>
<td>#0000FF</td>
</tr>
<tr>
<td>black</td>
<td>#000000</td>
</tr>
</tbody>
</table>

(8 rows)
```
Indexing

We already created the B-tree operator class:

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

```
EXPLAIN SELECT * FROM colour_names
    WHERE rgbvalue='#000000'
    ORDER BY rgbvalue;
```

**QUERY PLAN**

```
Index Scan using colour_lum_index on colour_names
    (cost=0.13..8.20 rows=4 width=36)
    Index Cond: (rgbvalue = '#000000'::colour)
(2 rows)
```

`postgres=#`
Summary so far

We have created a type:

- With input and output functions
- With functions for splitting a colour into components and calculating luminence

Index support:

- Operators: $>$ $>$= $==$ $<=$ $<$
- A comparison function: `colour_cmp`
- A B-tree operator class to tie the above together
Wait, there’s more!

- Hash function and operator class
  - for hash index support
  - for hash joins and aggregates
- Casts
- Cross-datatype operators
- Binary I/O routines
- Analyze function
- typmod
  - VARCHAR(50)
  - NUMERIC(1,5)
Packaging

~/presentations/PGConfEU2013/src (master)$ ls -l
total 16
-rw-r--r-- 1 heikki heikki 2523 loka 25 11:11 colour--1.0.sql
-rw-r--r-- 1 heikki heikki 1618 loka 25 11:15 colour.c
-rw-r--r-- 1 heikki heikki 144 loka 25 11:10 colour.control
-rw-r--r-- 1 heikki heikki 185 loka 25 11:09 Makefile

Upload to PGXN
PART 2: advanced indexing

Ordering by luminence is nice..
But what about finding a colour that’s the closes match to given colour?
Distance function

$$\sqrt{(R_1 - R_2)^2 + (G_1 - G_2)^2 + (B_1 - B_2)^2}$$

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 IMMUTABLE STRICT;

CREATE OPERATOR <-> (  
  PROCEDURE = colour_diff,  
  LEFTARG=colour,  
  RIGHTARG=colour  
);
Order by distance

postgres=#
SELECT * FROM colour_names ORDER BY rgbvalue <-> '#00FF00';

<table>
<thead>
<tr>
<th>name</th>
<th>rgbvalue</th>
</tr>
</thead>
<tbody>
<tr>
<td>green</td>
<td>#00FF00</td>
</tr>
<tr>
<td>lawn green</td>
<td>#87F717</td>
</tr>
<tr>
<td>dark grey</td>
<td>#808080</td>
</tr>
<tr>
<td>black</td>
<td>#000000</td>
</tr>
<tr>
<td>light grey</td>
<td>#COC0C0</td>
</tr>
<tr>
<td>white</td>
<td>#FFFFFF</td>
</tr>
<tr>
<td>blue</td>
<td>#0000FF</td>
</tr>
<tr>
<td>red</td>
<td>#FF0000</td>
</tr>
</tbody>
</table>

(8 rows)
But can we index that?

```
postgres=# explain SELECT * FROM colour_names
    ORDER BY rgbvalue <-> '#00FF00';
```

**QUERY PLAN**

```
Sort (cost=1.46..1.48 rows=8 width=36)
    Sort Key: (sqrt((((((red(rgbvalue) - 0))::double precision ^ 2::double precision) + (((green(rgbvalue) - 255))::double precision ^ 2::double precision)) + (((blue(rgbvalue) - 0))::double precision ^ 2::double precision))))
-> Seq Scan on colour_names (cost=0.00..1.38 rows=8 width=36)
(3 rows)
```

Oh, a seqscan. With a billion colours, that could be slow..
Advanced index types

PostgreSQL offers three kinds of generalized index types:

- GIN
- GiST (Generalized Search Tree)
- SP-GiST (Space-partitioned GiST)

PostgreSQL provides:

- 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 - extract each word from text, index the words
- Arrays - index the array elements
- Word similarity (pg_trgm) - extract trigrams from text, index trigrams
GiST

General tree structure
  ▶ Extremely flexible
  ▶ You define the layout

Used for:
  ▶ Full-text search
  ▶ Trigrams
  ▶ Hierarchical labels, ltree contrib module
  ▶ B-tree emulation
  ▶ Points (R-tree)
B-tree refresher

Five operators:

- <
- <=
- =
- >
- >=

One support function;

- colour_cmp() - returns -1, 0 or 1
GiST

GiST has 8 support functions:

- consistent - when searching, decide which child nodes to visit
- union - create a new inner node from a set of entries
- compress - converts a data item to internal format, for storing
- decompress - the reverse of compress
- penalty - used to decide where to insert new tuple
- picksplit - when page becomes full, how to split tuples on new pages?
- same - returns true if index entries are equal
- distance - returns the distance of an index entry from query (optional)
R-Tree
R-Tree using GiST

Support functions:

- consistent - Return true if point falls in the bounding box
- union - Expand bounding box to cover the new point
- penalty - Return distance of given point from bounding box
- picksplit - Divide points minimizing overlap
- same - trivial equality check
- distance - distance of given point from bounding box or point
- compress/decompress - do nothing
R-Tree for colours using GiST

- Treat colours as 3d points.
- In internal nodes, store a bounding box
- In leaf nodes, store the colour itself
Space-Partitioned GiST (SP-GiST)

New index type in PostgreSQL 9.2
Like GiST, but SP-GiST totally partitions the key space. * No overlapping pages.
Can be used to implement e.g:

- prefix tries for text
- Quad-tree for points
- KD-tree for points
KD-tree

(2dbaum.svg, Wikimedia Commons / Public Domain)
Implementing SP-GiST operator class for colours

- KD-tree.
- Each colour is a point in 3-D space. Each component, Red, Green, Blue, is one dimension.
SP-GiST support functions

SP-GiST requires 5 support functions:

- **config** - Returns static information about the implementation.
- **choose** - How to insert a new value into an inner tuple?
- **picksplit** - How to create a new inner tuple over a set of leaf tuples.
- **inner_consistent** - Returns set of nodes (branches) to follow during tree search.
- **leaf_consistent** - Returns true if a leaf tuple satisfies a query.
Advanced indexes summary

PostgreSQL offers three kinds of generalized index types:

- GIN (Generalized Inverted Index)
- GiST (Generalized Search Tree)
- SP-GiST (Space-partitioned GiST)

PostgreSQL provides:

- WAL-logging
- Concurrency
- Isolation
- Durability
- Transactions
The end

You’re the expert in your problem domain!
You define the semantics!
PostgreSQL handles the rest!