Writing a user-defined datatype

Heikki Linnakangas / VMware

What is a datatype?

- A datatype encapsulates semantics and rules.
- Text representation, used in psql and elsewhere

Different kinds of datatypes

- PostgreSQL offers many built-in datatypes, e.g:
  - Integer, Text, Timestamp, Point, JSON
- Other datatypes can be derived from the base types
  - Domains, Arrays, Ranges, Enums
- Extensions
  - PostGIS, contrib modules, ISBN

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 internally treats data as opaque *Datums*
  - *Fixed* or *variable* length (varlena) chunk of memory
  - Can be copied around the system and stored on disk
- All other operations are defined by the datatype author.

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 – *(white)*

Let’s begin

A datatype needs ***input*** and ***output functions***, to get data in and out of the system.

- Must be written in C
- We’ll use hex strings like “#112233” as the text representation
- Internal representation is a fixed-length, 32-bit integer

Input function

```
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);
}
```

Output function

```
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);
}
```

Register type with PostgreSQL

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

CREATE 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 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:

```
SELECT * FROM colour_names
WHERE rgbvalue = '#FF0000';

ERROR: operator does not exist: colour = unknown
```

Equality operator =

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

```
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:

```
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 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;

    name     | rgbvalue | red | green | blue
-------------+----------+-----+-------+------
 red         | #FF0000  | 255 |     0 |    0
 green       | #00FF00  |   0 |   255 |    0
 blue        | #0000FF  |   0 |     0 |  255
 white       | #FFFFFF  | 255 |   255 |  255
 Black       | #000000  |   0 |     0 |    0
 light grey  | #C0C0C0  | 192 |   192 |  192
 lawn green  | #87F717  | 135 |   247 |   23
 dark grey   | #808080  | 128 |   128 |  128
(8 rows)
```

Luminance

The human eye is more sensitive to green light [1].

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

[1] https://en.wikipedia.org/wiki/Relative_luminance

Luminance

```
select name, rgbvalue,
  red(rgbvalue), green(rgbvalue), blue(rgbvalue),
  round( luminence(rgbvalue), 6) as luminance
from colour_names;

    name     | rgbvalue | red | green | blue | luminance
-------------+----------+-----+-------+------+-----------
 red         | #FF0000  | 255 |     0 |    0 | 0.300000
 green       | #00FF00  |   0 |   255 |    0 | 0.590000
 blue        | #0000FF  |   0 |     0 |  255 | 0.110000
 white       | #FFFFFF  | 255 |   255 |  255 | 1.000000
 black       | #000000  |   0 |     0 |    0 | 0.000000
 light grey  | #C0C0C0  | 192 |   192 |  192 | 0.752941
 lawn green  | #87F717  | 135 |   247 |   23 | 0.740235
 dark grey   | #808080  | 128 |   128 |  128 | 0.501961
(8 rows)
```

Summary so far

We have created a type with:

- input and output functions
- equality operator (=)
- functions for splitting a colour into components and calculating luminance

Ordering

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

ERROR: could not identify an ordering operator for type
colour
```

What is an *ordering operator*?

<

<=

= (we already did this)

>=

>

We’re going to define these in terms of luminance.

Implementing ordering functions

```
CREATE FUNCTION colour_lt (colour, colour)
RETURNS bool AS
$$
  SELECT luminance($1) < luminance($2);
$$
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:

```
-- Comparison function, required for b-tree operator class.
CREATE FUNCTION luminance_cmp(colour, colour) RETURNS integer
AS $$
  SELECT CASE WHEN $1 = $2 THEN 0
              WHEN luminance($1) < luminance($2) THEN 1
              ELSE -1
         END;
$$ LANGUAGE SQL IMMUTABLE STRICT;
```

Operator class

```
-- B-tree operator class
CREATE OPERATOR CLASS luminance_ops
  DEFAULT FOR TYPE colour USING btree AS
  OPERATOR 1 <,
  OPERATOR 2 <=,
  OPERATOR 3 =,
  OPERATOR 4 >=,
  OPERATOR 5 >,
  FUNCTION 1 luminance_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 the B-tree operator class:

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

Indexing

```
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)
```

Summary so far

We have created a ***type*** with:

- ***input*** and ***output*** functions
- ***function***s for splitting a colour into components and calculating luminance

Index support:

- ***operators***: > >= = <= <
- ***comparison*** function: `colour_cmp`
- ***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
- Analyze function

Packaging

```
~/colour-datatype/$ ls -l
total 20
-rw-r--r-- 1 heikki heikki 2868 Oct 25  2013 colour--1.0.sql
-rw-r--r-- 1 heikki heikki 1618 Oct 25  2013 colour.c
-rw-r--r-- 1 heikki heikki  144 Oct 25  2013 colour.control
-rw-r--r-- 1 heikki heikki  185 Jan 23 12:26 Makefile
```

When packaged as an extension, you can install it with:

```
CREATE EXTENSION colour;
```

PART 2: Advanced indexing

Ordering by luminance is nice…

How about finding a colour that’s the closest match to a given colour?

What does “closest colour” mean?

Distance:

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

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 IMMUTABLE STRICT;

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

Order by distance

```
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)
```

But can we index that?

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

                             QUERY PLAN

------------------------------------------------------------------------------------
------------------------------------------------------------------------------------
--------------------------------------------------------------------------
 Sort  (cost=135.80..138.97 rows=1270 width=44)
   Sort Key: (sqrt((((((red(rgbvalue) - 0))::double precision ^ '2'::double prec
ision) + (((green(rgbvalue) - 255))::double precision ^ '2'::double precision))
+ (((blue(rgbvalue) - 0))::double precision ^ '2'::double precision))))
   ->  Seq Scan on colour_names  (cost=0.00..70.33 rows=1270 width=44)
(3 rows)
```

Oh, a *Seq Scan*. With a billion rows, that could be slow...

Advanced index types

PostgreSQL offers several generalized index access methods:

- B-tree
- Hash
- GiST
- GIN
- GP-GiST
- BRIN

GIN: Generalized Inverted Index

Splits input into multiple parts, and indexes the parts.

- Full text search – extract *words* from text, index each word
- 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:

- Geographical data (e.g. R-tree)
- Full-text search, Trigrams

B-tree refresher

Five ***operators***:

< <= = > >=

One ***support function***:

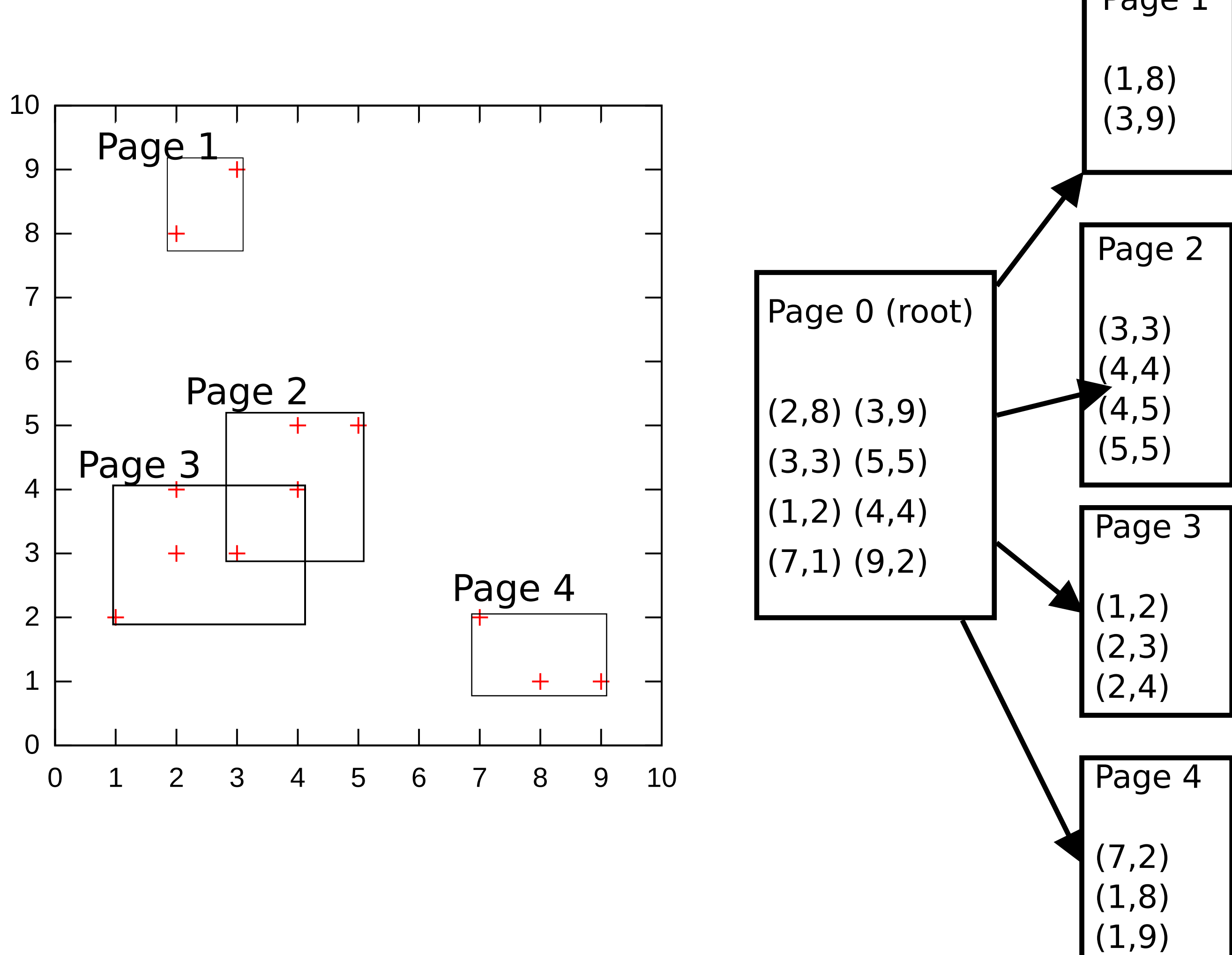
colour_cmp() - returns -1, 0 or 1

GiST

GiST has ***eight*** support functions:

- ***Consistent*** – when searching, determine which nodes need to be visited
- ***Union*** – create a new inner node from a set of entries
- ***Compress, Decompress*** – convert a data item to internal format for storing (and back)
- ***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 entries are equal
- ***Distance*** – returns the distance of an index entry from query (optional)

R-tree



R-tree using GiST

Support functions:

- ***Consistent*** – Returns 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 heck
- ***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 (cube)
- In leaf nodes, store the colour itself

Implementation is left as an exercise for the audience :-)

You **are the expert in your problem domain!**

PostgreSQL **provides:**

- WAL-logging, replication
- Transactions
- ACID: Atomicity, Concurrency, Isolation, Durability
- Backup & Restore, Point-in-time Recovery

Thank you!

For more information:

- PostgreSQL documentation
- Read the PostgreSQL source (Really! It’s quite readable!)

The code presented in this presentation is available at:
https://github.com/hlinnaka/colour-datatype/