Exploring PostgreSQL Datatypes

OpenSource Days 2013
Copenhagen, Denmark

Magnus Hagander
magnus@hagander.net

Magnus Hagander

- PostgreSQL
  - Core Team member
  - Committer
  - PostgreSQL Europe
- Redpill Linpro
  - Infrastructure services
  - Principal database consultant

PostgreSQL

- "The Worlds Most Advanced Open Source Database"
- RDBMS
- Lots of features
- Designed to use those features

PostgreSQL datatypes

- Pluggable type system
- Everything is a type!
  - >300 types by default
  - Table is type

Standard datatypes

- A few quick notes
  - text vs varchar
  - prefer int4/int8, not numeric
- But that's not why we're here

Advanced datatypes

- Plenty to choose from
- Internal and external
  - e.g. PostGIS

Advanced datatypes

- Date & time
- Range types
- json & hstore

Date & time

- Please don't use seconds-since-1970
- Instead use
  - timestamp with time zone
  - date
  - time

Timestamp with time zone

- Should be your go-to datatype for timestamps
- Does not mean it stores the timezone!
- Means it considers the timezone

```
CREATE TABLE tbl(t timestamp with time zone)
```

Timestamp with time zone

```
postgres=# SELECT t FROM tbl;
 2013-03-30 17:45:15+01

postgres=# SET timezone='America/Montreal';
SET

postgres=# SELECT t FROM tbl;
 2013-03-30 12:45:15-04
```

Timestamp with time zone

```
postgres=# SELECT t AT TIME ZONE 'Asia/Tokyo'
postgres-#  FROM tbl;
 2013-03-31 01:45:15
```

Timestamp math

```
postgres=# SELECT t + '3 hours' FROM tbl;
 2013-03-30 20:45:15+01

postgres=# SELECT t - now() FROM tbl;
 50 days 04:13:17.575963
```

Timestamp math vs timezone

```
postgres=# SELECT t + '10 hours' FROM tbl;
 2013-03-31 04:45:15+02
```

Getting the pieces out

```
postgres=# SELECT extract('year' FROM t) FROM tbl;
      2013

postgres=# SELECT extract('epoch' FROM t) FROM tbl;
 1364661915
```

Associated datatypes

- Use date for dates
  - Don't use timestamp and set time to zeroes!
- Use time for times
  - When you have no date, don't make one up!

Advanced datatypes

- That's pretty standard
  - "Everybody" has it
  - It just happens to be more convenient
- Let's look at some really cool stuff

Range types

- Store *any* type of range data
- Builtin and custom
  - integers and numerics
  - timestamps and dates
- Inclusive or exclusive
- Discrete or continuous

Range types - why?

- Simplify queries
- Advanced operators
- Indexes
- Constraints

Range-type simple example

- "On call schedule"
- Let's assume we have employees
  - Identified by employee_id
- Someone needs to be on call
- When there is a problem, find who's on call right now

Before range types

```
CREATE TABLE schedule (
  id  serial PRIMARY KEY,
  employee_id integer,
  starttime timestamp with time zone,
  endtime timestamp with time zone
);
```

Who's on call?

```
postgres=# SELECT employee_id FROM schedule WHERE
postgres-# now() BETWEEN starttime AND endtime;
  1
```

- Ok, that was easy
- What about *can I schedule X tomorrow between 16 and 17*

Is X free?

```
SELECT count(*) FROM schedule
WHERE
 employee_id = 1 AND (
  (
    starttime >= '2013-02-09 16:00' AND
    starttime <= '2013-02-09 17:00'
  ) OR (
    endtime >= '2013-02-09 16:00' AND
    endtime <= '2013-02-09 17:00'
  )
)
```

Is X free?

- That's not enough...
  - Contained
  - Completely covering
  - Start before, end in or after
  - Start in, end before or after
- Finding overlaps is complicated
- Gets worse with more factors

Range types!

- tstzrange = range type of timestamptz

```
CREATE TABLE schedule (
  id  serial PRIMARY KEY,
  employee_id integer,
  t tstzrange
);
```

Who's on call?

```
postgres=# SELECT employee_id FROM schedule_old WHERE
postgres-# now() BETWEEN starttime AND endtime;
  1

postgres=# SELECT employee_id FROM schedule WHERE
postgres-# t @> now();
  1
```

Is X free?

```
postgres=# SELECT count(*) FROM schedule WHERE employee_id=1 AND
postgres-# t && '[2013-02-09 16:00, 2013-02-09 17:00]'::tstzrange;
    1
```

Is X free?

```
postgres=# SELECT count(*) FROM schedule WHERE employee_id=1 AND
postgres-# t && '[2013-02-09 16:00, 2013-02-09 17:00]'::tstzrange;
    1

postgres=# SELECT count(*) FROM schedule WHERE employee_id=1 AND
postgres-# t && '[2013-02-09 17:00, 2013-02-09 18:00]'::tstzrange;
    1
```

Is X free?

```
postgres=# SELECT count(*) FROM schedule WHERE employee_id=1 AND
postgres-# t && '[2013-02-09 16:00, 2013-02-09 17:00]'::tstzrange;
     1

postgres=# SELECT count(*) FROM schedule WHERE employee_id=1 AND
postgres-# t && '[2013-02-09 17:00, 2013-02-09 18:00]'::tstzrange;
     1

postgres=# SELECT count(*) FROM schedule WHERE employee_id=1 AND
postgres-# t && '(2013-02-09 17:00, 2013-02-09 18:00]'::tstzrange;
     0
```

Range definitions

- (and) indicates exclusive range
- [and] indicates inclusive range
- Leave out to make infinite, e.g.
  - '(2,)'::int4range
  - '[now,]'::tstzrange

Discrete and continuous

- Discrete ranges "have next and prev", e.g.

```
postgres=# SELECT '(2,5)'::int4range;
 [3,5)

postgres=# select int4range(2,5,'()');
 [3,5)
```

- Continuous ranges don't, e.g.

```
postgres=# SELECT '(2,5)'::numrange;
 (2,5)
```

Indexing

- Fully supported by GiST indexes

```
postgres=# CREATE INDEX schedule_t_idx ON schedule USING gist (t);
CREATE INDEX
```

- Supports operators for:
  - Equals (=)
  - Overlaps (&&)
  - Containment (<@, @>)
  - Adjacent (-|-)
  - Does-not-extend-to-side-of (<&, &>)

Constraints

- Exclusion constraints supported
  - "Generalized UNIQUE"

```
postgres=# ALTER TABLE schedule ADD CONSTRAINT duplicate_booking
postgres-# EXCLUDE USING gist (t WITH &&);
ALTER TABLE

postgres=# INSERT INTO schedule (employee_id, t) VALUES
postgres-# (1, '[2013-02-08 13:30,2013-02-08 14:00]');
ERROR:  conflicting key value violates exclusion
  constraint "duplicate_booking"
DETAIL:  Key (t)=(["2013-02-08 13:30:00+00","2013-02-08 14:00:00+00"])
  conflicts with existing key (t)=(["2013-02-08 13:00:00+00",
    "2013-02-08 17:00:00+00"]).
```

Moving on

- Range types fit the traditional model
  - Basic RDBMS ideas
- What about non-relational?
  - Supposedly the future?
  - Combine with relational!

JSON

- JavaScript Object Notation
- Text-based data
- Schemaless
- Hierarchical
- PostgreSQL has native support (since 9.2)!

JSON in PostgreSQL

```
CREATE TABLE jsontable (
  id serial PRIMARY KEY,
  j json
);
```

Storing JSON

```
postgres=# INSERT INTO jsontable (j) VALUES ('{
postgres'#  "id":"mha",
postgres'#  "name":"Magnus Hagander",
postgres'#  "country": "Sweden"
postgres'# }');
INSERT 0 1
```

- Validates json syntax
- Maintains formatting

Mapping JSON

```
postgres=# SELECT row_to_json(schedule)
postgres-# FROM schedule WHERE id=1;
 {"id":1,"employee_id":1,"t":"[\"2013-02-08 13:00:00+00\",
   \"2013-02-08 17:00:00+00\")"}
```

Mapping JSON

```
postgres=# SELECT row_to_json(schedule)
postgres-# FROM schedule WHERE id=1;
 {"id":1,"employee_id":1,"t":"[\"2013-02-08 13:00:00+00\",
   \"2013-02-08 17:00:00+00\")"}

postgres=# SELECT row_to_json(t) FROM (
postgres-#  SELECT id, employee_id
postgres-#  FROM schedule) t;
 {"id":2,"employee_id":1}
 {"id":3,"employee_id":2}
 {"id":1,"employee_id":1}
```

Using JSON

- That's really all there is to JSON
  - At least in 9.2
- For full power, use with pl/v8
  - Extraction and combination
  - Indexing (using expression indexes)
  - Much more

More nonrelational

- Why have only one, when you can have two?

hstore

- Generic key-value store
- Fully indexable!
- Typeless
- No nesting

Installing hstore

```
postgres=# CREATE EXTENSION hstore;
CREATE EXTENSION
```

Defining hstore columns

```
postgres=# CREATE TABLE items (
postgres(#  itemid serial NOT NULL PRIMARY KEY,
postgres(#  itemname text NOT NULL,
postgres(#  tags hstore);
CREATE TABLE
```

Creating hstore values

```
postgres=# INSERT INTO items (itemname, tags)
postgres-# VALUES ('item1', 'color => red, category => stuff');
INSERT 0 1
```

Query by hstore

```
postgres=# SELECT itemnname FROM items
postgres-# WHERE tags->'color' = 'red';
 item1
```

Indexed access

- Create normal expression index on column

```
CREATE INDEX foo ON
  ((items->'color'))
```

- Requires one index per key
- That's what we wanted to avoid...

Dynamic GiST indexing

- Create index covering all keys

```
CREATE INDEX hstoreidx
 ON items
 USING gist(tags)
```

- Available for multiple operators
  - All types of containment
- Must use these operators

Querying with GiST

```
postgres=# EXPLAIN
postgres-# SELECT itemname FROM items
postgres-# WHERE tags @> 'color=>red';

 Index Scan using hstoreidx on items  (cost=0.12..8.14 rows=1 width=32)
   Index Cond: (tags @> '"color"=>"red"'::hstore)
```

Querying for tag presence

```
postgres=# EXPLAIN
postgres-# SELECT itemname FROM items
postgres-# WHERE tags ? 'color';

 Index Scan using hstoreidx on items  (cost=0.12..8.14 rows=1 width=32)
   Index Cond: (tags ? 'color'::text)
```

Downsides of hstore

- Values are not typed
  - Just strings
- No hierarchy
- No key compression
- Still slower than "normal columns"
  - But very useful with sparse data!

Can I have both?

- You'd really want both
- Hierarchical hstore with full indexing
- With a nice JSON API
- Not yet...

Thank you!

Magnus Hagander
magnus@hagander.net
@magnushagander