A Tour of PostgreSQL Data Types

Jonathan S . Katz
Jim “Jimbo” Mlodgenski

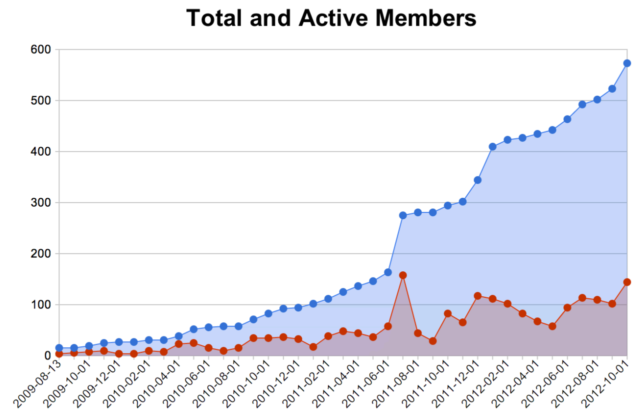
PGCon 2013 – May 21, 2013

Who We Are

- Jonathan S. Katz
  - CTO, VenueBook
  - Co-Organizer NYC PostgreSQL User Group (NYCPUG)
- Jim Mlodgenski
  - CEO, StormDB
  - Co-Organizer NYCPUG

A Brief Note on NYCPUG

- Active since 2010
- 700 members
- Monthly Meetups
- PGDay NYC 2013
  - March 22
  - 100 attendees
- Part of PG.US
- PGConf NYC 2014



Why Data Types

- Fundamental
  - 0 => 1
  - 00001111
- Building Blocks
  - 0x41424344
- Accessibility
  - 1094861636
  - ‘ABCD’

Why Data Types

- Primitive Data Types
  - Integers, floating points, booleans, characters
- Primitive Data Structures
  - Strings, arrays, linked lists, hash tables
- Data Structures++
  - Classes, structs, trees, matrices

Data Storage

- “Persistence of Memory”

Data Access and Retrieval

- Recall what we have stored
- Represent as it originally was
- Interface between disk <=> application

PostgreSQL

- Roots from “INGRES”
  - Image storage
- Data integrity = foremost concern
- Data representation
- Robustness
- Performance

The PostgreSQL Data Type Tour

- Data types
- Functions
- Features
- Indexes
- Use cases
- Extensions

The PostgreSQL Data Type Tour

- Assumptions
  - PostgreSQL 9.2+
    - Some looks at PostgreSQL 9.3beta1

Number Types

Name	Storage Size	Range
smallint	2 bytes	-32768 to +32767
integer	4 bytes	-2147483648 to +2147483647
bigint	4 bytes	-9223372036854775808 to 9223372036854775807
decimal	variable	up to 131072 digits before the decimal point; up to 16383 digits after the decimal point
numeric	variable	up to 131072 digits before the decimal point; up to 16383 digits after the decimal point
real	4 bytes	6 decimal digits precision
double	8 bytes	15 decimal digits precision

Integers

- smallint
  - Use only if disk space is a premium, e.g. embedded devices
- bigint
  - Slower than int
- int
  - For everything else…

numeric

- numeric
  - Provides scale and precision
    - *Scale* – count of decimal places
      - *987.123456 has a scale of 6*
    - *Precision* – total count of significant digits
      - *987.123456 has a precision of 9*
  - Declarations
    - numeric(precision, scale)
      - max declarable is (1000, 100)
    - numeric(precision)
      - essentially an integer
    - Numeric
      - Precision & scale up to limit (147455, 16383)
      - Contrary to SQL standard

numeric & NYC Sales Tax

```
SELECT 100 * (0.08875)::numeric;
---
8.875

SELECT 100 * (0.08875)::numeric(7,2);
---
9.0

SELECT (100 * 0.08875)::numeric(7,2);
---
8.88
```

numeric

- Storage
  - Determined by size of numeric type, no padding

```
SELECT pg_column_size('123'::numeric(7,2));
pg_column_size
----------------
              8

SELECT pg_column_size('123.45'::numeric(7,2));
pg_column_size
----------------
             10
```

Numbers – numeric

- 'NaN'
- decimal equivalent to numeric

Numbers – floating point

- IEEE 754
- Inexact
  – Unexpected behavior may occur
    - Overflow/underflow
    - Equality
- Constants
  – 'Nan', 'Infinity', '-Infinity'
- Types
  – real => 1E-37 <=> 1E+37
  – double precision => 1E-308 <=> 1E+308
  – float(1) <=> float(24) = real
  – float(25) <=> float(53) = double precision

numeric vs double precision

```
\timing

CREATE TABLE floats (x double precision);
CREATE TABLE numerics (x numeric(15, 15));

INSERT INTO floats
SELECT random() FROM generate_series(1,1000000);

INSERT INTO numerics
SELECT * FROM floats;

CREATE INDEX floats_idx ON floats (x);
CREATE INDEX numerics_idx ON numerics (x);

SELECT * FROM floats WHERE x >= 0.7;
-- avg 280ms

SELECT * FROM numerics WHERE x >= 0.7;
-- avg 120ms
```

numeric vs floating points

- generally it is better to use numeric
- floating point usage is application specific
  - reading data from a thermometer
  - IEEE 754 specific programs
  - too many rows for larger numeric data type
  - do not require precision
- understand ramifications before making choice

Number Functions

- ceil/ceiling, floor
- exp (exponential), ln, log
- greatest, least
- random, setseed
- round, truncate
- sign
- to_number
- degrees(radians), radians(degrees)
- cos, acos, sin, asin
- cot (cotangent), tan, atan
- atan2(x, y) = atan(x/y)

Serial Types

Name	Storage Size	Range
smallserial	2 bytes	1 to 32767
serial	4 bytes	1 to 2147483647
bigserial	8 bytes	1 to 9223372036854775807

Serial “Types”

- Not truly a data type, but a convenience

```
CREATE TABLE awesome (
   id serial
);
```

or

```
CREATE SEQUENCE awesome_id_seq;
CREATE TABLE awesome (
   id integer NOT NULL DEFAULT nextval(‘awesome_id_seq’)
);
ALTER SEQUENCE awesome_colname_seq OWNED BY awesome.id;
```

Serial Functions

- nextval
  - advances sequence and returns new value

```
SELECT nextval('sequence_name');
```

- setval
  - sets the current value of the sequence

```
SELECT setval('sequence_name', 2); -- nextval returns 3
SELECT setval('sequence_name', 2, true); -- nextval returns 3
SELECT setval('sequence_name', 2, false); -- nextval returns 2
```

- currval
  - returns current value of sequence if sequence has been manipulated in session

```
SELECT currval('sequence_name');
```

- lastval
  - returns current value of last sequence that has been manipulated in session

```
SELECT lastval();
```

Monetary Types

Name	Storage Size	Range
money	8 bytes	-92233720368547758.08 to 92233720368547758.07

Monetary Types: The Story

- Stores monetary amounts with precision based on ‘lc_monetary’ setting
- Output based on lc_monetary
  - ‘$1,000.00’

Monetary Types: The Reality

- Don’t use it
- Store money as
  - integer family of types
  - numeric

Character Types (or Strings)

Name	Description
varchar(n)	variable-length with limit
char(n)	fixed-length, blank padded
text	variable unlimited length

Character Types

- char(n) and varchar(n) mostly follow the ANSI standard
  - Will throw an error if given a string longer than n characters (not bytes)
  - Trailing spaces in char(n) are ignored in char(n) comparisons, and stripped when converting to other string types
  - Unlike many databases, char(n) is NOT stored as a fixed-sized field in Postgres. It is treated exactly the same as varchar(n) except for being padded

Character Types

- "varlena"
  - Called internally when creating any character type
- text
  - Preferred type in practice
  - Max ~1GB
- varchar(n)
  - Use only when you have to restrict length
  - CPU overhead (marginal)
- char(n)
  - avoid
  - unexpected behavior e.g. with "LIKE" expressions

Character Types & Encoding

- What do encoding, cache management, and concurrency all have in common?

String Functions

- ascii
- bit_length
- btrim
- char_length
- chr
- concat
- convert
- decode
- encode
- initcap
- length

- lower
- lpad
- ltrim
- md5
- octet_length
- overlay
- pg_client_encoding
- position
- quote_ident
- quote_literal
- repeat
- replace

- rpad
- rtrim
- split_part
- strpos
- substr
- substring
- to_ascii
- to_hex
- translate
- trim
- upper

Binary Data Types

Name	Storage Size	Description
bytea	1 to 4 bytes plus size of binary string	variable-length binary string

Binary Data Types

- Used to store “raw bytes”
- Different output formats:
  - Pre–9.0: PostgreSQL “escape”
  - 9.0+: hex
  - ‘bytea_output’ – config parameter to choose (default: ‘hex’)

Binary Data Types

- Should probably not store raw binary data in PostgreSQL
- If you must, keep in its own table and JOIN when needed

Date / Time Types

- PostgreSQL – second to none
- `timestamp with time zone`
- `timestamp without time zone`
- `date`
- `time with time zone`
- `time without time zone`
- `interval`

Date / Time Types

- PostgreSQL – second to none

Name	Size	Range	Resolution
timestamp without timezone	8 bytes	4713 BC to 294276 AD	1 microsecond / 14 digits
timestamp with timezone	8 bytes	4713 BC to 294276 AD	1 microsecond / 14 digits
date	4 bytes	4713 BC to 5874897 AD	1 day
time without timezone	8 bytes	00:00:00 to 24:00:00	1 microsecond / 14 digits
time with timezone	12 bytes	00:00:00+1459 to 24:00:00-1459	1 microsecond / 14 digits
interval	12 bytes	-178000000 years to 178000000 years	1 microsecond / 14 digits

Date / Time General Notes

- timestamp = timestamp without time zone
- timestamptz = timestamp with time zone
- time, timestamp, and interval have optional argument “p”
  - “precision” – number of fractional digits
  - p <@ [0,6], default is 6
  - SELECT CURRENT_TIMESTAMP::time(4);
- interval has other storage options (more later)

Date / Time Input

- PostgreSQL is very flexible
  - ISO 8601
  - SQL
  - POSTGRES
  - and more
- Day / Month / Year ordering – “datestyle” parameter
  - datestyle = ‘iso, mdy’
  - DMY, MDY, YMD

Date Input

Input	Description
2013-05-03	ISO 8601, May 3 with any datestyle (recommended format)
May 3, 2013	May 3 with any datestyle
5/3/2013	May 3 with MDY, March 5 with DMY
5/21/2013	May 21 with MDY, rejected with other formats
5/2/3	May 2, 2003 with MDY, February 5 2003 with DMY, February 3, 2005 with YMD
2013-May-3 May-3-2013 3-May-2013	All equivalent with any datestyle
32-May-3	May 3 with YMD, otherwise rejected
3-May-32	rejected with YMD, otherwise May 3
May-3-32	rejected with YMD, otherwise May 3
20130503	ISO 8601
2013.123	year and day of year, in this case May 3, 2013
J2456416	Julian date, in this case May 3, 2013

Time / Time with Time Zone Input

Input	Description
22:12:34.567 22:12:34 22:12 221234	ISO 8601
10:12 AM	same as 10:12
10:12 PM	same as 22:12
22:12-5 22:12-05:00 221200-05	ISO 8601, same as 10:12 PM EST
22:12 EST	time zone specified by identifier, in this case an abbreviation; same as 10:12 PM EST

Time Zone Input

Input	Description
EST	Abbreviation (Eastern Standard Time)
America/New_York	Full name
EST5EDT	POSIX style
-5:00 -500 -5	ISO 8601 style
zulu z	Military abbreviation for UTC

http://www.postgresql.org/docs/current/static/datatype-datetime.html#DATATYPE-TIMEZONES

```
SELECT CURRENT_TIMESTAMP AT TIME ZONE ’CST';
```

Timestamp Input

- Combine Date and Time and Time Zone inputs!

```
<date> <time> <timezone> (AD|BC)
```

- timestamp with time zone
  - internally stored as UTC
  - default representation is from “timezone” parameter or system default

- Caveat Emptor – which of these are equivalent?

```
TIMESTAMP '2013-05-21 10:00:00'
TIMESTAMP '2013-05-21 10:00:00-05'
TIMESTAMP WITH TIME ZONE '2013-05-21 10:00:00-05'
```

Be careful with your data type declarations!

Special Inputs

Input String	Valid Types	Description
epoch	date, timestamp	1970-01-01 00:00:00+00 (Unix system time zero)
infinity	date, timestamp	later than all other time stamps
-infinity	date, timestamp	earlier than all other time stamps
now	date, time, timestamp	current transaction's start time
today	date, timestamp	midnight today
tomorrow	date, timestamp	midnight tomorrow
yesterday	date, timestamp	midnight yesterday
allballs	time	00:00:00.00 UTC

Date / Time Output

Style Specification	Description	Example
ISO	ISO 8601, SQL standard	1997-12-17 07:37:16-08
SQL	traditional style	12/17/1997 07:37:16.00 PST
Postgres	original style	Wed Dec 17 07:37:16 1997 PST
German	regional style	17.12.1997 07:37:16.00 PST

Can adjust with:

- Command: SET <datestyle>;
- postgresql.conf – 'DateStyle' parameter
- environmental var: PGDATESTYLE

Intervals

- YEAR
- MONTH
- DAY
- HOUR
- MINUTE
- SECOND
- YEAR TO MONTH
- DAY TO HOUR
- DAY TO MINUTE
- DAY TO SECOND
- HOUR TO MINUTE
- HOUR TO SECOND
- MINUTE TO SECOND

```
SELECT CURRENT_DATE +
   '11 days 11 hour 11 month 11 year'::interval;
---------------------
 2025-01-03 11:00:00
```

Why Intervals Are Cool

```
SELECT avg(hours)
FROM sleep
WHERE
  day BETWEEN
   CURRENT_DATE - '7 day'::interval AND
   CURRENT_DATE;
```

Basic Operators

Operator	Example	Result
+	date '2001-09-28' + integer '7'	date '2001-10-05'
+	date '2001-09-28' + interval '1 hour'	timestamp '2001-09-28 01:00:00'
+	date '2001-09-28' + time '03:00'	timestamp '2001-09-28 03:00:00'
+	interval '1 day' + interval '1 hour'	interval '1 day 01:00:00'
+	timestamp '2001-09-28 01:00' + interval '23 hours'	timestamp '2001-09-29 00:00:00'
+	time '01:00' + interval '3 hours'	time '04:00:00'
-	- interval '23 hours'	interval '-23:00:00'
-	date '2001-10-01' - date '2001-09-28'	integer '3' (days)
-	date '2001-10-01' - integer '7'	date '2001-09-24'
-	date '2001-09-28' - interval '1 hour'	timestamp '2001-09-27 23:00:00'
-	time '05:00' - time '03:00'	interval '02:00:00'
-	time '05:00' - interval '2 hours'	time '03:00:00'
-	timestamp '2001-09-28 23:00' - interval '23 hours'	timestamp '2001-09-28 00:00:00'
-	interval '1 day' - interval '1 hour'	interval '1 day -01:00:00'
-	timestamp '2001-09-29 03:00' - timestamp '2001-09-27 12:00'	interval '1 day 15:00:00'
*	900 * interval '1 second'	interval '00:15:00'
*	21 * interval '1 day'	interval '21 days'
*	double precision '3.5' * interval '1 hour'	interval '03:30:00'
/	interval '1 hour' / double precision '1.5'	interval '00:40:00'

Selected Functions

- `age(timestamp, timestamp)`
- `age(timestamp)`
- `date_part(text, timestamp)`
  - Same as 'EXTRACT'
- `date_trunc(text, timestamp)`
- `justify_days(interval)`
- `justify_hours(interval)`
- `CURRENT_TIMESTAMP, CURRENT_DATE, CURRENT_TIME`

Boolean Data Types

- Postgres – second to none :-)

Name	Size
boolean	1 byte

- These are all equivalent
  - TRUE, ‘t’, ‘true’, ‘y’, ‘yes’, ‘on’, ‘1’
  - FALSE, ‘f’, ‘false’, ‘n’, ‘no’, ‘off’, ‘0’
  - all case-insensitive, preferred TRUE / FALSE

Boolean Data Type Notes

- bool = boolean
- **NEVER CREATE AN INDEX ON A BOOLEAN TYPE**

Enumerated Types

```
SELECT name, color_name
   FROM suspect s INNER JOIN eye_color e
                     ON e.color_id = s.color_id
```

The table eye_color is (fairly) static

Declaring an Enum

CREATE TYPE enum_eye_color AS ENUM ('blue', 'brown', 'gray', 'green');

ALTER TYPE enum_eye_color ADD VALUE 'amber' BEFORE 'blue';

The declaration order is used by ORDER BY

Using an Enum

```
CREATE TABLE suspect
 (name TEXT,
 eye_color enum_eye_color);

INSERT INTO suspect VALUES
('John Doe', 'brown');
```

Using an Enum

```
test1=# select * from suspect order by
  eye_color;
    name    | eye_color
------------+-----------
 Jack Smith | blue
 John Doe   | brown
```

Enum Alternatives

Lookup Table	

Constraint	CREATE TABLE suspect
	(name TEXT NOT NULL,
	 eye_color TEXT NOT NULL,
	 CONSTRAINT check_eye_color
	   CHECK (eye_color IN
	   ('blue','brown','gray','green')));

Domain	CREATE DOMAIN eye_color AS TEXT CONSTRAINT check_eye_color CHECK (VALUE IN ('blue','brown','gray','green')));

Stretch Break #1

Reading Material For Break: B-Tree Indexes

- "default" index in Postgres
- optimized for retrieving data on circular disk
  - can sometimes help with sorts
- supports <=, <, =, >, >=
  - BETWEEN, IN
  - IS NOT NULL, IS NULL
  - LIKE in specific case of 'plaintext%'
  - ~ in specific case of '^plaintext'
  - ILIKE and ~* if pattern starts with nonalpha characters
- one of many indexes in Postgres
  - some of these conditions change with other indexes…

Geometric Types

Name	**Size**	**Representation**	**Format**
point	16 bytes	point on a plane	(x,y)
lseg	32 bytes	finite line segment	((x1, y1), (x2, y2))
box	32 bytes	rectangular box	((x1, y1), (x2, y2))
path	16 + 16n bytes	closed path (similar to polygon, n = total points	((x1, y1), (x2, y2), …, (xn, yn))
path	16 + 16n bytes	open path, n = total points	[(x1, y1), (x2, y2), …, (xn, yn)]
polygon	40 bytes + 16n	polygon	((x1, y1), (x2, y2), …, (xn, yn))
circle	24 bytes	circle – center point and radius	<(x, y), r>

It Only Does Everything…

Operator	Description	Example
+	Translation	`box '((0,0),(1,1))' + point '(2.0,0)'`
-	Translation	`box '((0,0),(1,1))' - point '(2.0,0)'`
*	Scaling/rotation	`box '((0,0),(1,1))' * point '(2.0,0)'`
/	Scaling/rotation	`box '((0,0),(2,2))' / point '(2.0,0)'`
#	Point or box of intersection	`'((1,-1),(-1,1))' # '((1,1),(-1,-1))'`
#	Number of points in path or polygon	`# '((1,0),(0,1),(-1,0))'`
@-@	Length or circumference	`@-@ path '((0,0),(1,0))'`
@@	Center	`@@ circle '((0,0),10)'`
##	Closest point to first operand on second operand	`point '(0,0)' ## lseg '((2,0),(0,2))'`
<->	Distance between	`circle '((0,0),1)' <-> circle '((5,0),1)'`
&&	Overlaps? (One point in common makes this true.)	`box '((0,0),(1,1))' && box '((0,0),(2,2))'`
<<	Is strictly left of?	`circle '((0,0),1)' << circle '((5,0),1)'`
>>	Is strictly right of?	`circle '((5,0),1)' >> circle '((0,0),1)'`
&<	Does not extend to the right of?	`box '((0,0),(1,1))' &< box '((0,0),(2,2))'`
&>	Does not extend to the left of?	`box '((0,0),(3,3))' &> box '((0,0),(2,2))'`
<<|	Is strictly below?	`box '((0,0),(3,3))' <<| box '((3,4),(5,5))'`
|>>	Is strictly above?	`box '((3,4),(5,5))' |>> box '((0,0),(3,3))'`

It Only Does Everything Cont'd…

<table>
<tr><td>&<|</td><td>Does not extend above?</td><td>box '((0,0),(1,1))' &<| box '((0,0),(2,2))'</td></tr>
<tr><td>|&></td><td>Does not extend below?</td><td>box '((0,0),(3,3))' |&> box '((0,0),(2,2))'</td></tr>
<tr><td><^</td><td>Is below (allows touching)?</td><td>circle '((0,0),1)' <^ circle '((0,5),1)'</td></tr>
<tr><td>>^</td><td>Is above (allows touching)?</td><td>circle '((0,5),1)' >^ circle '((0,0),1)'</td></tr>
<tr><td>?#</td><td>Intersects?</td><td>lseg '((-1,0),(1,0))' ?# box '((-2,-2),(2,2))'</td></tr>
<tr><td>?-</td><td>Is horizontal?</td><td>?- lseg '((-1,0),(1,0))'</td></tr>
<tr><td>?-</td><td>Are horizontally aligned?</td><td>point '(1,0)' ?- point '(0,0)'</td></tr>
<tr><td>?|</td><td>Is vertical?</td><td>?| lseg '((-1,0),(1,0))'</td></tr>
<tr><td>?|</td><td>Are vertically aligned?</td><td>point '(0,1)' ?| point '(0,0)'</td></tr>
<tr><td>?-|</td><td>Is perpendicular?</td><td>lseg '((0,0),(0,1))' ?-| lseg '((0,0),(1,0))'</td></tr>
<tr><td>?||</td><td>Are parallel?</td><td>lseg '((-1,0),(1,0))' ?|| lseg '((-1,2),(1,2))'</td></tr>
<tr><td>@></td><td>Contains?</td><td>circle '((0,0),2)' @> point '(1,1)'</td></tr>
<tr><td><@</td><td>Contained in or on?</td><td>point '(1,1)' <@ circle '((0,0),2)'</td></tr>
<tr><td>~=</td><td>Same as?</td><td>polygon '((0,0),(1,1))' ~= polygon '((1,1),(0,0))'</td></tr>
</table>

It Only Does Everything Cont'd...

Function	Return Type	Description	Example
`area(`*object*`)`	`double precision`	area	`area(box '((0,0),(1,1))')`
`center(`*object*`)`	`point`	center	`center(box '((0,0),(1,2))')`
`diameter(circle)`	`double precision`	diameter of circle	`diameter(circle '((0,0),2.0)')`
`height(box)`	`double precision`	vertical size of box	`height(box '((0,0),(1,1))')`
`isclosed(path)`	`boolean`	a closed path?	`isclosed(path '((0,0),(1,1),(2,0))')`
`isopen(path)`	`boolean`	an open path?	`isopen(path '[(0,0),(1,1),(2,0)]')`
`length(`*object*`)`	`double precision`	length	`length(path '((-1,0),(1,0))')`
`npoints(path)`	`int`	number of points	`npoints(path '[(0,0),(1,1),(2,0)]')`
`npoints(polygon)`	`int`	number of points	`npoints(polygon '((1,1),(0,0))')`
`pclose(path)`	`path`	convert path to closed	`pclose(path '[(0,0),(1,1),(2,0)]')`
`popen(path)`	`path`	convert path to open	`popen(path '((0,0),(1,1),(2,0))')`
`radius(circle)`	`double precision`	radius of circle	`radius(circle '((0,0),2.0)')`
`width(box)`	`double precision`	horizontal size of box	`width(box '((0,0),(1,1))')`

Performance Considerations

- Size on disk
  - Consider I/O on retrievals
- Indexing
  - ~~B-tree~~
    - equality operators modified for ad-hoc purposes, e.g. area
  - Are we out of luck on performance?

Index Detour #1: Expression Indexes

- allows pre-computed values to be stored in an index
- useful for "on the fly" comparisons

```
SELECT * FROM receipts WHERE (subtotal + tax) <
  numeric(1000.00);
SELECT * FROM receipts WHERE upper(name) = 'JIM';
```

- fast for searches, costly on updates
- Easy to create

```
CREATE INDEX receipts_total_idx ON
  receipts ((subtotal + tax));
CREATE INDEX receipts_upper_idx ON
  receipts((upper(name)));
```

Back to Geometric Performance: Expression Indexes

```
CREATE TABLE houses (plot box);

INSERT INTO houses
SELECT box(
  point((500 * random())::int, (500 *
  random())::int),
  point((750 * random() + 500)::int, (750 *
  random() + 500)::int)
)
FROM generate_series(1, 1000000);
```

Area without Expression Index

```
EXPLAIN SELECT * FROM houses WHERE area(plot)
  BETWEEN 50000 AND 75000;
                                                QUERY
  ----
 Seq Scan on houses  (cost=0.00..27353.00
  rows=5000 width=32)
   Filter: ((area(plot) >= 50000::double
  precision) AND (area(plot) <= 75000::double
  precision)
```

Run time average 220ms

Area with Expression Index

```
CREATE INDEX houses_area_plot ON houses (area(plot));
EXPLAIN SELECT * FROM houses WHERE area(plot) BETWEEN
   50000 AND 75000;

Bitmap Heap Scan on houses  (cost=108.60..7160.30
   rows=5000 width=32)
    Recheck Cond: ((area(plot) >= 50000::double
   precision) AND (area(plot) <= 75000::double
   precision))
    ->  Bitmap Index Scan on houses_area_plot
   (cost=0.00..107.35 rows=5000 width=0)
          Index Cond: ((area(plot) >= 50000::double
   precision) AND (area(plot) <= 75000::double
   precision))
```

Average run time was 48ms

Index Detour #2: GiST

- "generalized search tree"
- balanced, tree-structured
- allows arbitrary indexing schemes
  – B-trees, R-trees
  – indexing on custom data types
- supports lots more operators
- can implement your own indexing scheme

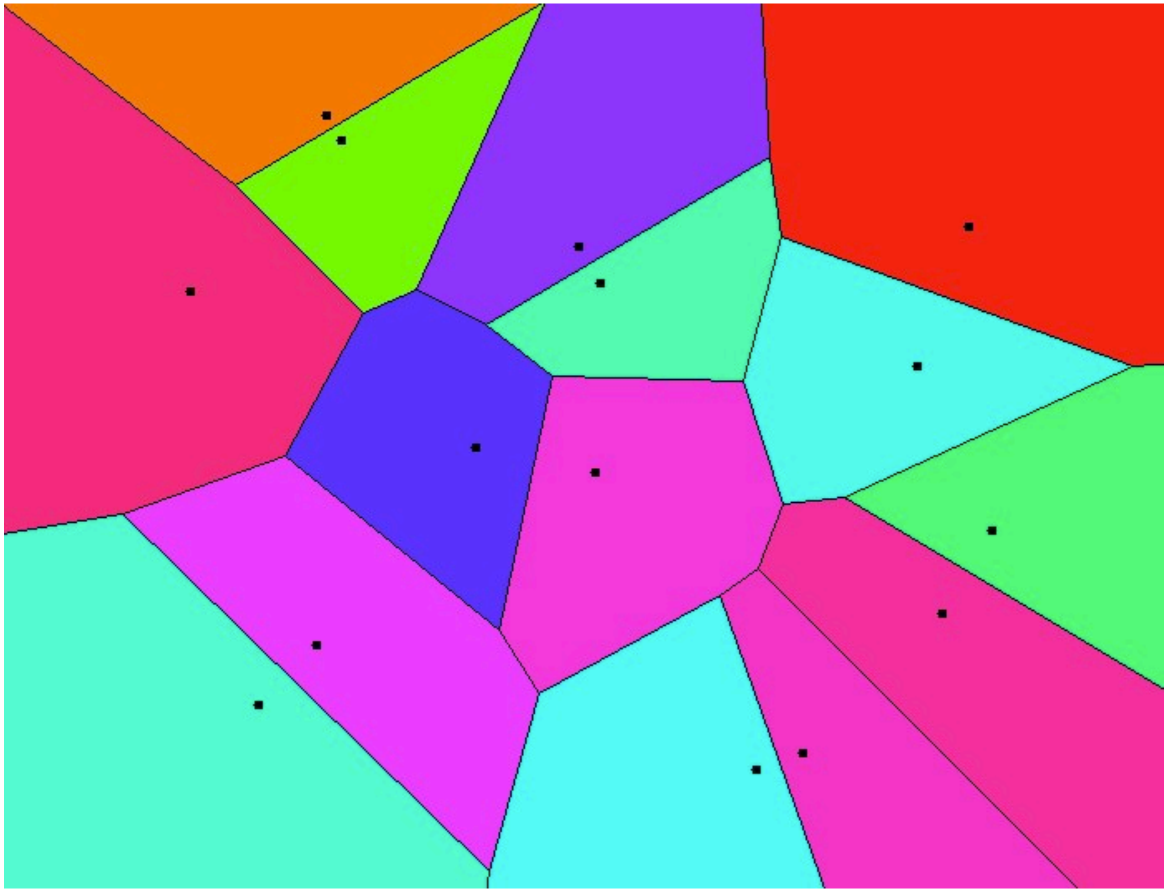
Index Detour #2: GiST

- GiST operators:
  - <<
  - &<
  - &>
  - >>
  - <<|
  - &<|
  - |&>
  - |>>
  - @>
  - <@
  - ~=
  - &&

Major Detour: K-Nearest Neighbor

- PostgreSQL 9.1+
- Given a collection of *n* objects
- When trying to classify an unknown object
  - compute the distance between all known objects
  - find the *k* ($k \geq 1$) closest objects to the unknown object

K=1 Example



Voronoi Diagram of order 1 can be used to make k=1 NN queries

KNN-GiST: A Very Special GiST Index

- (almost back to geometric types!)
- Let n = size of a table
- Can index data that provides a “<->” (distance) operator
- “k” = LIMIT clause

Geometry

```
CREATE INDEX geoloc_coord_idx ON geoloc USING gist(coord);
```

```
EXPLAIN ANALYZE
  SELECT
    coord,
    coord <-> point(500,500)
  FROM geoloc
  ORDER BY coord <-> point(500,500)
  LIMIT 10;
```

Results

Limit (cost=80958.28..80958.31 rows=10 width=20) (actual time=1035.313..1035.316 rows=10 loops=1)

-> **Sort** (cost=80958.28..85958.28 rows=2000000 width=20) (actual time=1035.312..1035.314 rows=10 loops=1)

Sort Key: ((coord <-> '(500,500)'::point))

Sort Method: top-N heapsort Memory: 25kB

-> **Seq Scan** on geoloc (cost=0.00..37739.00 rows=2000000 width=20) (actual time=0.029..569.501 rows=2000000 loops=1)

Total runtime: 1035.349 ms

Limit (cost=0.00..0.81 rows=10 width=20) (actual time=0.576..1.255 rows=10 loops=1)

-> Index Scan using geoloc_coord_idx on geoloc (cost=0.00..162068.96 rows=2000000 width=20) (actual time=0.575..1.251 rows=10 loops=1)

Order By: (coord <-> '(500,500)'::point)

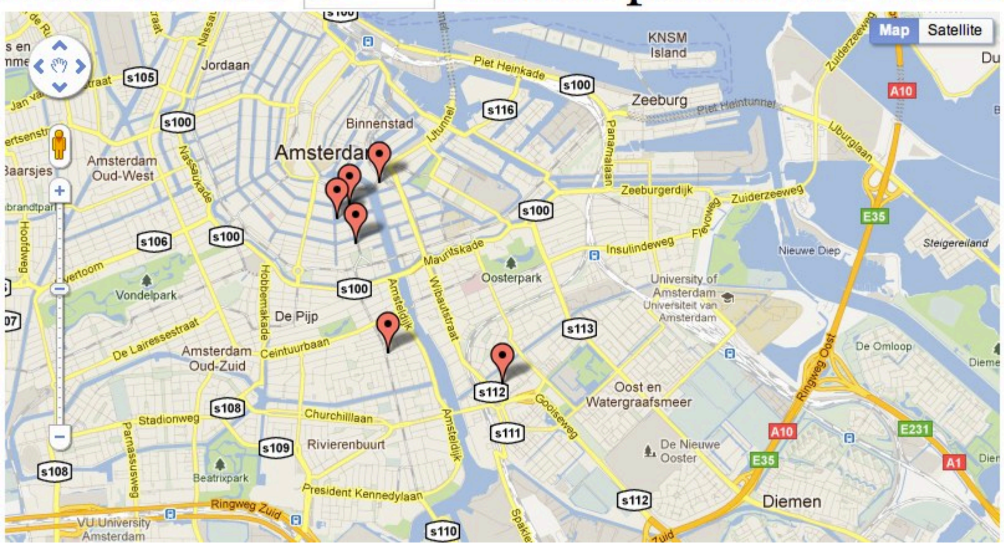
Total runtime: 1.391 ms

Application Examples

- Proximity map search – fast!

KNN - Amsterdam Edition

Find me the 5 closest places for coffee.



Geometric Type Index Summary

- ~~B-tree~~
- Expression indexes on geometric functions
- GiST indexes
  - Support special geometric operators
  - KNN-GiST

Addicted to Geometry? GIS?

- PostGIS
  - http://postgis.refractions.net/
  - OpenGIS, WKB, WKT
  - PostGIS EWKB, EWKT
  - SQL-MM Part 3
  - PostGIS Geography Type
  - …and takes advantage of GiST

Network Address Types

Name	Storage Size	Description
cidr	7 or 19 bytes	IPv4 or IPv6 networks
inet	7 or 19 bytes	IPv4 or IPv6 hosts or networks
macaddr	6 bytes	MAC addresses

Network Address Types

- `inet (IPv4 & IPv6)`
  - `SELECT '192.168.1.1'::inet;`
  - `SELECT '192.168.1.1/32'::inet;`
  - `SELECT '192.168.1.1/24'::inet;`
- `cidr (IPv4 & IPv6)`
  - `SELECT '192.168.1.1'::cidr;`
  - `SELECT '192.168.1.1/32'::cidr;`
  - ~~`SELECT '192.168.1.1/24'::cidr;`~~
- `macaddr`
  - `SELECT '08:00:2b:01:02:03'::macaddr`

inet + cidr =

Operator	Description	Example
<	is less than	inet '192.168.1.5' < inet '192.168.1.6'
<=	is less than or equal	inet '192.168.1.5' <= inet '192.168.1.5'
=	equals	inet '192.168.1.5' = inet '192.168.1.5'
>=	is greater or equal	inet '192.168.1.5' >= inet '192.168.1.5'
>	is greater than	inet '192.168.1.5' > inet '192.168.1.4'
<>	is not equal	inet '192.168.1.5' <> inet '192.168.1.4'
<<	is contained within	inet '192.168.1.5' << inet '192.168.1/24'
<<=	is contained within or equals	inet '192.168.1/24' <<= inet '192.168.1/24'
>>	contains	inet '192.168.1/24' >> inet '192.168.1.5'
>>=	contains or equals	inet '192.168.1/24' >>= inet '192.168.1/24'
~	bitwise NOT	~ inet '192.168.1.6'
&	bitwise AND	inet '192.168.1.6' & inet '0.0.0.255'
|	bitwise OR	inet '192.168.1.6' | inet '0.0.0.255'
+	addition	inet '192.168.1.6' + 25
-	subtraction	inet '192.168.1.43' - 36
-	subtraction	inet '192.168.1.43' - inet '192.168.1.19'

…even more functions

Function	Return Type	Description	Example
`abbrev(inet)`	`text`	abbreviated display format as text	`abbrev(inet '10.1.0.0/16')`
`abbrev(cidr)`	`text`	abbreviated display format as text	`abbrev(cidr '10.1.0.0/16')`
`broadcast(inet)`	`inet`	broadcast address for network	`broadcast('192.168.1.5/24')`
`family(inet)`	`int`	extract family of address; `4` for IPv4, `6` for IPv6	`family('::1')`
`host(inet)`	`text`	extract IP address as text	`host('192.168.1.5/24')`
`hostmask(inet)`	`inet`	construct host mask for network	`hostmask('192.168.23.20/30')`
`masklen(inet)`	`int`	extract netmask length	`masklen('192.168.1.5/24')`
`netmask(inet)`	`inet`	construct netmask for network	`netmask('192.168.1.5/24')`
`network(inet)`	`cidr`	extract network part of address	`network('192.168.1.5/24')`
`set_masklen(inet, int)`	`inet`	set netmask length for `inet` value	`set_masklen('192.168.1.5/24', 16)`
`set_masklen(cidr, int)`	`cidr`	set netmask length for `cidr` value	`set_masklen('192.168.1.0/24'::cidr, 16)`
`text(inet)`	`text`	extract IP address and netmask length as text	`text(inet '192.168.1.5')`

Bit Strings

Name	Storage Size	Description
bit(n)	y + ceil(n / 8) bytes	stores exactly n 0s and 1s y = 5 or 8
bit varying(n)	y + ceil(n / 8) bytes	stores up to n 0s and 1s y = 5 or 8
bit varying	variable	stores unlimited number of 0s and 1s

Bit Strings

```
SELECT B’1001001010101000’;
SELECT ‘1’::bit(3); -- ‘100’;
CREATE TABLE bits (
  a bit(3),
  b bit varying(5),
  c bit varying
);
```

Bit Strings

```
SELECT B'101' || B'010'; -- 101010
SELECT B'1011' & B'0101'; -- 0001
SELECT B'1011' | B'0101'; -- 1111
SELECT B'1011' # B'0101'; -- 1110
SELECT ~B'1011'; -- 0100
SELECT B'1011' << 2; -- 1100
SELECT B'1011' >> 2; -- 0010
```

Full Text Search

- built-in to PostgreSQL
- uses “tsearch2” algorithm
- appropriate data types + indexes for retrieval

Full Text Search

- tsvector
  - a sorted list of normalized lexemes

```
SELECT 'PGDay NYC 2013 is a conference run by the local NYC
  PostgreSQL User Group'::tsvector;

tsvector
----------
'2013' 'Group' 'NYC' 'PGDay' 'PostgreSQL' 'User' 'a' 'by'
  'conference' 'is' 'local' 'run' 'the
```

Full Text Search

```
SELECT tsvector('now:1 i:2 have:3
  learned:4 my:5 data:6 types:7
  and:8 i:9 can:10 teach:11 the:12
  world:13 now:14');

  tsvector
-------------
 'and':8 'can':10 'data':6 'have':3
  'i':2,9 'learned':4 'my':5 'now':
  1,14 'teach':11 'the':12 'types':
  7 'world':13
```

Full Text Search

```
SELECT tsvector('now:1C i:2 have:
  3 learned:4B my:5 data:6A
  types:7B');

  tsvector
-------------
 'data':6A 'have':3 'i':2
  'learned':4B 'my':5 'now':1C
  'types':7B
```

Full Text Search

- use 'to_tsvector' to normalize text
- used for indexing on actual full text search applications

```
SELECT to_tsvector('PGDay NYC 2013 is a
  conference run by the local NYC PostgreSQL
  User Group');

to_tsvector
------------------
 '2013':3 'confer':6 'group':14 'local':10
  'nyc':2,11 'pgday':1 'postgresql':12 'run':7
  'user':13
```

Full Text Search

- tsquery
  - lexemes that are to be searched for
  - operators: &, |, !

```
SELECT 'PostgreSQL & conference'::tsquery;

tsquery
------------------------------
 'PostgreSQL' & 'conference’
```

Full Text Search

```
SELECT 'PostgreSQL & (conference | 2013)'::tsquery;

 tsquery
----------------
 'PostgreSQL' & ( 'conference' | '2013' )

SELECT 'PostgreSQL & !conference | 2013'::tsquery;

 tsquery
----------------
 'PostgreSQL' & !'conference' | '2013'
```

Full Text Search

- prefix matching

```
SELECT 'pg:* & conference'::tsquery;

tsquery
----------------------
 'pg:*' & 'conference'
```

- be aware of stemming

```
S
ELECT to_tsvector('postgraduate') @@ to_tsquery('postgres:*');
```

Full Text Search Functions

- get_current_ts_config
- length
- numnode
- plainto_tsquery
- querytree
- setweight
- strip
- to_tsquery
- to_tsvector
- ts_headline
- ts_rank
- ts_rank_cd
- ts_rewrite
- tsvector_update_trigger
- tsvector_update_trigger_column

Full Text Searching

- A lot of functions and "weird" operators involved

```
SELECT title
FROM conferences
WHERE to_tsvector(title) @@
  to_tsquery('postgres:*');
```

Index Detour #3: GIN

- Generalized Inverted Index
  - search for composite values in composite items (huh?)
  - provides general access methods for implementor to provide logic
  - stores data by "keys"
    - rows referenced by multiple keys
  - exact vs partial match
  - fast on reads, slow on writes
- Supported on
  - full text search
  - btree_gin
  - hstore
  - pg_trgm
  - one-dimensional arrays on built-in types

Full Text Searching

- Can use GiST or GIN
  - Size: GIN 2-3x larger
  - Read performance: GIN 2-3x faster
  - Index creation: GiST 2-3x faster
  - Update: GiST moderately to 10x faster
    - (FASTUPDATE on GIN)
  - "100,000 lexemes"

```
CREATE INDEX full_text_search_idx ON conferences
  USING gin(to_tsvector('title'));
```

More on Full Text Search

- Lecture in itself
- http://www.postgresql.org/docs/current/static/textsearch.html

UUID

- Universally Unique Identifiers
- 16 bytes on disk
- Acceptable Formats
  - A0EEBC99-9C0B-4EF8-BB6D-6BB9BD380A11
  - {a0eebc99-9c0b-4ef8-bb6d-6bb9bd380a11}
  - a0eebc999c0b4ef8bb6d6bb9bd380a11
  - a0ee-bc99-9c0b-4ef8-bb6d-6bb9-bd38-0a11
  - {a0eebc99-9c0b4ef8-bb6d6bb9-bd380a11}

UUID Functions

```
CREATE EXTENSION "uuid-ossp";
```

- uuid_generate_v1
- uuid_generate_v1mc
- uuid_generate_v3
- uuid_generate_v4
- uuid_generate_v5

XML

- ensures that XML is valid
- no comparison methods
- caveat emptor: encoding
  - e.g. 'xpath'

XML

- Ensures the value is well formed XML

```
postgres=# SELECT xml '<PUG>NYC</PUG>';
      xml
----------------
 <PUG>NYC</PUG>

postgres=# SELECT xml '<PUG>NYC';
ERROR:  invalid XML content at character 12
DETAIL:  line 1: Premature end of data in tag
  PUG line 1
<PUG>NYC
```

XML Functions

- xml_is_well_formed
- xpath_string
- xpath_number
- xpath_bool
- xpath_nodeset
- xpath_nodeset
- xpath_nodeset
- xpath_list
- xpath_list

Embedded XML Fragments

```
CREATE TABLE Journey(
   JourneyId INTEGER,
   LX XML
);

INSERT INTO Journey
VALUES (1, '<LX>
             <LEG LAT=”52” LONG=”0”>
             <LEG LAT=”44” LONG=”5” >
            </LX>');

UPDATE Journey
   SET LegX = '<LX>'||
                xpath_string(LegX, ''/lx/leg'') ||
                '<LEG LAT=”56” LONG=”10” >
               </LX>'
WHERE JourneyId = 1;


SELECT xpath_number(LegX, 'fn:count(/lx/leg)') as num_legs FROM Journey WHERE
   JourneyId = 1;
```

Arrays

```
CREATE TABLE person (
    full_name text,
    sports text[],
    cars text[][],
    numbers int[3],
    incomes int ARRAY[4],
    phrases text ARRAY
);
```

- PostgreSQL does not enforce size restrictions
  - 9.3 and below

Arrays

```
SELECT ARRAY[1,2,3];
SELECT ARRAY[ARRAY[1,2], ARRAY[3,4]];
SELECT '{1,2,3}';
SELECT '{{1,2},{3,4}}';
```

Arrays

- arrays are 1-indexed

```
SELECT (ARRAY[1,2,3])[1]; -- returns 1
SELECT (ARRAY[1,2,3])[0]; -- returns NULL
SELECT (ARRAY[1,2,3])[1:2]; -- returns {1,2}
SELECT (ARRAY[1,2,3])[2:3]; -- returns {2,3}
SELECT (ARRAY[1,2,3])[2:3][2]; -- returns {2,3}
SELECT ((ARRAY[1,2,3])[2:3])[2]; -- returns 3
```

INSERT with ARRAY

```
INSERT INTO person
  VALUES ('Rocky Bama',
  '{“baseball”,”basketball”}',
  '{{"Toyota","Prius"},{"Chevy","Tahoe"}}');

INSERT INTO person
  VALUES ('Rocky Bama',
  ARRAY['baseball','basketball'],
  ARRAY[['Toyota','Prius'],
  ['Chevy','Tahoe']]);
```

SELECT and ARRAY

```
SELECT cars FROM person;

cars
---------------------------------
 {{Toyota,Prius},{Chevy,Tahoe}}
```

SELECT and ARRAY

```
SELECT *
FROM person
WHERE sports[1] = 'baseball';
```

SELECT and ARRAY

```
SELECT full_name
FROM person
WHERE 'baseball' = ANY (sports);
```

- "true" if any entry in sports for a person is 'baseball'

SELECT and ARRAY (2)

```
SELECT full_name
FROM person
WHERE 'baseball' = ALL
  (sports);
```

- "true" only if every entry in sports for a tuple in person is 'baseball'

UPDATE and ARRAY

```
UPDATE person SET sports[2] = 'tennis';

UPDATE person SET sports[2:3] = '{"hockey",
   "soccer"}';

UPDATE person SET sports = ARRAY['foozball',
   'billiards'];
```

Array Operators

- <, <=, =, >= >, <>
  - compares each array elements
  - B-tree index = yes!
- @>, <@

```
SELECT ARRAY[1,2,3] @> ARRAY[1,2];
SELECT ARRAY[1,2] <@ ARRAY[1,2,3];
```

- &&

```
SELECT ARRAY[1,2,3] && ARRAY[3,4,5];
```

- ||

```
SELECT ARRAY[1,2,3] || ARRAY[3,4,5];
SELECT ARRAY[ARRAY[1,2], ARRAY[3,4]] || ARRAY[5,6];
SELECT ARRAY[1,2,3] || 4;
```

- can use GIN index on one dimensional arrays

Array Functions

- modification

```
SELECT array_append(ARRAY[1,2,3], 4);
SELECT array_prepend(1, ARRAY[2,3,4]);
SELECT array_cat(ARRAY[1,2], ARRAY[3,4]);
SELECT array_remove(ARRAY[1,2,1,3], 1);
SELECT array_replace(ARRAY[1,2,1,3], 1, -4)
```

- size

```
SELECT array_length(ARRAY[1,2,3,4], 1); -- 4
SELECT array_ndims(ARRAY[ARRAY[1,2], ARRAY[3,4]]);
-- 2
SELECT array_dims(ARRAY[ARRAY[1,2], ARRAY[3,4]]);
-- [1:2][1:2]
```

Array Functions

- bounds

```
SELECT array_lower(ARRAY[2,3,4], 1);
SELECT array_upper(ARRAY[2,3,4], 1);
```

- join

```
SELECT array_to_string(ARRAY[1,2,NULL,4], ',', '*');
-- 1,2,*,4
```

- expand

```
SELECT unnest(ARRAY[1,2,3]);
 unnest
--------
      1
      2
      3
```

array_agg

- useful for variable-length lists or "unknown # of columns"
  - e.g. "find all speakers for a talk"

```
SELECT
   t.title
   array_agg(s.full_name)
FROM talk t
JOIN speakers_talks st ON st.talk_id = t.id
JOIN speaker s ON s.id = st.speaker_id
GROUP BY t.title;

 title                    | array_agg
--------------------------+-----------
 Data Types               | {Jonathan, Jim}
 Administration           | {Bruce}
 User Groups              | {Josh, Jonathan, Magnus}
```

JSON

- Added in 9.2
- Ensures the value is valid JSON

```
SELECT '[{"PUG": "NYC"}]'::json;
      json
------------------
 [{"PUG": "NYC"}]

SELECT '[{"PUG": "NYC"]'::json;
ERROR:  invalid input syntax for type json at character
  8
DETAIL:  Expected "," or "}", but found "]".
CONTEXT:  JSON data, line 1: [{"PUG": "NYC"]
```

JSON

- Enhanced functionality added in 9.3

Operator	Description	Example
->	return JSON array element OR JSON object field	'[1,2,3]'::json -> 0; '{"a": 1, "b": 2, "c": 3}'::json -> 'b';
->>	return JSON array element OR JSON object field AS text	['1,2,3]'::json ->> 0; '{"a": 1, "b": 2, "c": 3}'::json ->> 'b';
#>	return JSON object using path	'{"a": 1, "b": 2, "c": [1,2,3]}'::json #> '{c, 0}';
#>>	return JSON object using path AS text	'{"a": 1, "b": 2, "c": [1,2,3]}'::json #> '{c, 0}';

JSON

- array_to_json

```
SELECT array_to_json(ARRAY[ARRAY[1,2], ARRAY[3,4]])
--------
[[1,2],[3,4]]
```

- row_to_json

```
SELECT row_to_json(ROW(1,2,3));
--------
{"f1":1,"f2":2,"f3":3}

SELECT row_to_json(x) FROM x LIMIT 1;
--------
{"a":56,"b":42,"c":63}
```

JSON

- json_extract_path, json_extract_path_text
  - LIKE (#>, #>>) but with list of args

```
SELECT json_extract_path(
  '{"a": 1, "b": 2, "c": [1,2,3]}'::json,
  'c', '0');
```

JSON

- to_json
- json_each, json_each_text

```
SELECT * FROM json_each('{"a": 1, "b":
  [2,3,4], "c": "wow"}'::json);

 key |  value
-----+---------
 a   | 1
 b   | [2,3,4]
 c   | "wow"
```

JSON

- json_object_keys

```
SELECT * FROM json_object_keys('{"a": 1,
  "b": [2,3,4], "c": { "e":
  "wow" }}'::json);
----------
 a
 b
 c
```

JSON

- json_populate_record

```
CREATE TABLE stuff (a int, b text, c int[]);

SELECT *
FROM json_populate_record(NULL::stuff, '{"a": 1, "b": "wow"}');

 a |  b  | c
---+-----+---
 1 | wow |

SELECT *
FROM json_populate_record(NULL::stuff, '{"a": 1, "b": "wow", "c":
   [4,5,6]}');
ERROR:  cannot call json_populate_record on a nested object
```

JSON

- json_populate_recordset

```
SELECT *
FROM json_populate_recordset(NULL::stuff, '[{"a": 1,
  "b": "wow"},
    {"a": 2, "b": "cool"}
   ]'
);

 a |  b   | c
---+------+---
 1 | wow  |
 2 | cool |
```

JSON

- json_agg

```
SELECT b, json_agg(stuff)
FROM stuff
GROUP BY b;

  b   |              json_agg
------+----------------------------------
 neat | [{"a":4,"b":"neat","c":[4,5,6]}]
 wow  | [{"a":1,"b":"wow","c":[1,2,3]}, +
      |  {"a":3,"b":"wow","c":[7,8,9]}]
 cool | [{"a":2,"b":"cool","c":[4,5,6]}]
```

Stretch Break #2

Stretch Break #2

- If you don't want to stretch, try this puzzle:

With this table:

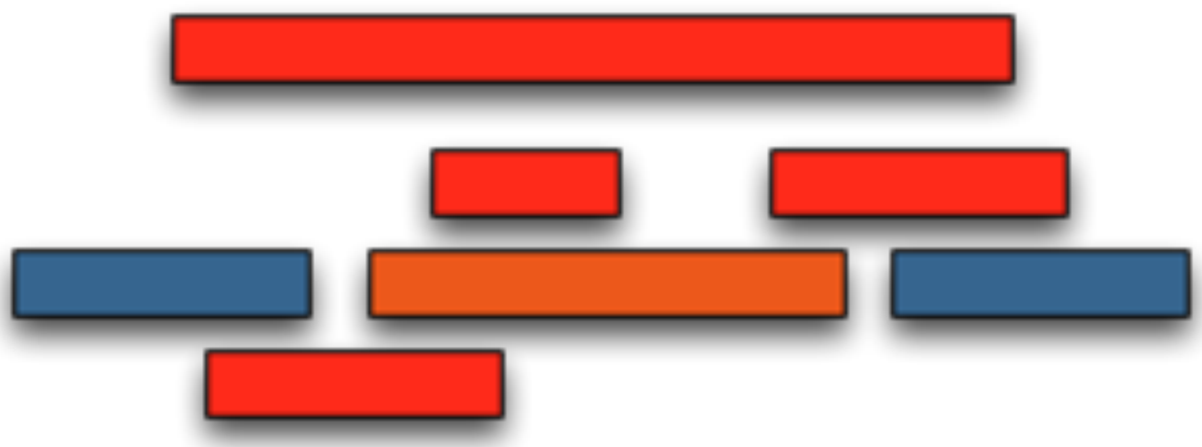
name (text)	low (int)	high (int)
a	20	30
b	25	35
c	10	15
d	17	24
e	40	50
f	26	36

Write a query that finds all the names whose range of values are between 18 and 26

Ranges

- Scheduling
- Probability
- Measurements
- Financial applications
- Clinical trial data
- Intersections of ordered data

Why Range Overlaps Are Difficult



Before Postgres 9.2

- OVERLAPS

```
SELECT
   ('2013-01-08`::date, '2013-01-10'::date) OVERLAPS
   ('2013-01-09'::date, '2013-01-12'::date);
```

- Limitations:
  - Only date/time
  - Start <= x <= End

Postgres 9.2+

- `INT4RANGE (integer)`
- `INT8RANGE (bigint)`
- `NUMRANGE (numeric)`
- `TSRANGE (timestamp without time zone)`
- `TSTZRANGE (timestamp with time zone)`
- `DATERANGE (date)`

Range Type Size

- Size on disk = 2 * (data type) + 1
  - sometimes magic if bounds are equal

```
SELECT pg_column_size(daterange(CURRENT_DATE, CURRENT_DATE));
----------------
              9

SELECT pg_column_size(daterange(CURRENT_DATE,CURRENT_DATE +
   1));
----------------
             17
```

Range Bounds

- Ranges can be inclusive, exclusive or both
  - `[2,4] => 2 ≤ x ≤ 4`
  - `[2,4) => 2 ≤ x < 4`
  - `(2,4] => 2 < x ≤ 4`
  - `(2,4) => 2 < x < 4`

- Can also be empty

Infinite Ranges

- Ranges can be infinite
  - `[2,) => 2 ≤ x < ∞`
  - `(,2] => -∞ < x ≤ 2`
- CAVEAT EMPTOR
  - “infinity” has special meaning with timestamp ranges
  - `[CURRENT_TIMESTAMP,) = [CURRENT_TIMESTAMP,]`
  - `[CURRENT_TIMESTAMP, 'infinity') <> [CURRENT_TIMEAMP, 'infinity']`

Constructing Ranges

```
SELECT '[1,10]'::int4range;

 int4range
-----------
 [1,11)
(1 row)
```

Constructing Ranges

- Constructor functions too
  - Defaults to '[)'

```
test=# SELECT numrange(9.0, 9.5);

  numrange
------------
 [9.0,9.5)
(1 row)
```

Bonus

- Can have arrays of ranges

```
test=# SELECT ARRAY[int4range(1,3),
  int4range(2,4), int4range(3,8)];
             array
---------------------------
 {"[1,3)","[2,4)","[3,8)"}
```

Simple Overlaps

```
SELECT *
FROM cars
WHERE cars.price_range && int4range(13000, 15000, '[]')
ORDER BY lower(cars.price_range);

 id |         name          |  price_range
----+-----------------------+---------------
  5 | Ford Mustang          | [11000,15001)
  6 | Lincoln Continental | [12000,14001)
(2 rows)
```

Range Indexes

- Creating a GiST index on ranges speeds up queries with these operators:

```
=
&&
<@
@>
<<
>>
-|-
&<
&>
```

Range Indexes

```
CREATE INDEX cars_price_range_idx ON cars USING gist (price_range);

-- EXPLAIN $PREVIOUS_QUERY

 QUERY PLAN
----------------------------------------------------------------------
   ---------------------
 Sort  (cost=129.66..129.87 rows=84 width=49)
   Sort Key: (lower(price_range))
   ->  Bitmap Heap Scan on cars2  (cost=4.95..126.97 rows=84 width=49)
         Recheck Cond: (price_range && '[13000,15000)'::int4range)
         ->  Bitmap Index Scan on cars2_price_range_idx
   (cost=0.00..4.93 rows=84 width=0)
               Index Cond: (price_range && '[13000,15000)'::int4range)
(6 rows)
```

Performance

```
test=# EXPLAIN ANALYZE SELECT * FROM ranges WHERE
int4range(500,1000) && bounds;

QUERY PLAN
------------
Bitmap Heap Scan on ranges
(actual time=0.283..0.370 rows=653 loops=1)
   Recheck Cond: ('[500,1000)'::int4range && bounds)
   ->  Bitmap Index Scan on ranges_bounds_gist_idx
       (actual time=0.275..0.275 rows=653 loops=1)
         Index Cond: ('[500,1000)'::int4range &&
bounds)
 Total runtime: 0.435 ms
```

What If the Range is Much Larger?

```
test=# EXPLAIN ANALYZE SELECT * FROM ranges WHERE
int4range(10000,1000000) && bounds;

QUERY PLAN
-------------
 Bitmap Heap Scan on ranges
 (actual time=184.028..270.323 rows=993068 loops=1)
   Recheck Cond: ('[10000,1000000)'::int4range &&
bounds)
   ->  Bitmap Index Scan on ranges_bounds_gist_idx
       (actual time=183.060..183.060 rows=993068 loops=1)
         Index Cond: ('[10000,1000000)'::int4range &&
bounds)
 Total runtime: 313.743 ms
```

Another Index Detour: SP-GiST

- "space-partitioned generalized search tree"
- designed for handling unbalanced data structures
  - quadtrees
  - k-d trees
  - radix trees
- searches are fast if match partitioning rules

```
CREATE INDEX ranges_bounds_spgist_idx ON ranges
  spgist(bounds);
```

SP-GiST and Ranges (9.3+)

- SP-GiST indexes support ranges

```
EXPLAIN ANALYZE SELECT * FROM ranges WHERE 500 <@ bounds;

   QUERY PLAN
------------------------
 Bitmap Heap Scan on ranges  (cost=20.41..1748.32 rows=516
   width=17) (actual time=0.558..1.463 rows=1502 loops=1)
    Recheck Cond: (500 <@ bounds)
    ->  Bitmap Index Scan on ranges_bounds_spgist_idx
   (cost=0.00..20.28 rows=516 width=0) (actual
   time=0.413..0.413 rows=1502 loops=1)
          Index Cond: (500 <@ bounds)
 Total runtime: 1.585 ms
```

Scheduling

```
CREATE TABLE travel_log (
  id serial PRIMARY KEY,
  name varchar(255),
  travel_range daterange,
  EXCLUDE USING gist (travel_range WITH &&)
);

INSERT INTO travel_log (name, trip_range) VALUES ('Chicago',
   daterange('2012-03-12', '2012-03-17'));

INSERT INTO travel_log (name, trip_range) VALUES ('Austin',
   daterange('2012-03-16', '2012-03-18'));

ERROR:  conflicting key value violates exclusion constraint
   "travel_log_trip_range_excl"
DETAIL:  Key (trip_range)=([2012-03-16,2012-03-18)) conflicts
   with existing key (trip_range)=([2012-03-12,2012-03-17)).
```

Extending Ranges

```
CREATE TYPE inetrange AS RANGE (
   SUBTYPE = inet
);

SELECT '192.168.1.8'::inet <@ inetrange('192.168.1.1',
   '192.168.1.10');

?column?
----------
 t

SELECT '192.168.1.20'::inet <@ inetrange('192.168.1.1',
   '192.168.1.10');

 ?column?
----------
 f
```

…back to the original problem

```
SELECT name
FROM ranges
WHERE range && int4range(18,26,'[]');
```

Composite Types

```
CREATE TYPE address AS (
   street TEXT,
   city TEXT,
   state TEXT,
   zip CHAR(10)
);
```

Composite Types

```
CREATE TABLE customer (
   full_name TEXT,
   mail_address address
);
```

Composite Types

```
INSERT INTO customer VALUES
('Joe Lee',
 ROW('100 Broad Street', 'Red
 Bank', 'NJ', '07701'));

INSERT INTO customer VALUES
('Joe Lee',
 ('100 Broad Street', 'Red Bank',
 'NJ', '07701'));
```

Composite Types with SELECT

```
SELECT (mail_address).city
FROM customer
WHERE (mail_address).state = 'NJ';

SELECT (customer.mail_address).city
FROM customer
WHERE (customer.mail_address).state = 'NJ';
```

Composite Types and JSON

```
SELECT row_to_json(customer)
FROM customer;

{
  "full_name":"Joe Lee",
  "mail_address": {
      "street":"100 Broad Street",
      "city":"Red Bank",
      "state":"NJ",
      "zip":"07701"
  }
}
```

Composite Type Operators

- Create a new function using CREATE FUNCTION that accepts one or two arguments using this type
- Use CREATE OPERATOR to choose what operator should be used to invoke this function

```
CREATE OPERATOR = (
    PROCEDURE = addr_eq,
   LEFTARG=address,
   RIGHTARG=address
);
```

Custom Data Types

- Still not enough choices? Create your own.
  - PostGIS geometry
  - Hstore
  - BioPostgres

Custom Data Types

- Needs an input function

```
Datum
tinyint_in(PG_FUNCTION_ARGS)
{
  char *num = PG_GETARG_CSTRING(0);
  PG_RETURN_TINYINT(pg_atoi(num, sizeof(tinyint), '\0'));
}
```

Custom Data Types

- Needs an output function

```
Datum
tinyint_out(PG_FUNCTION_ARGS)
{
  tinyint arg1 = PG_GETARG_TINYINT(0);
  /* sign, 3 digits, '\0' */
  char *result = (char *) palloc(5);

  pg_itoa(arg1, result);
  PG_RETURN_CSTRING(result);
}
```

Custom Data Types

- And a type defintion

```
CREATE TYPE tinyint (
    INPUT = tinyint_in,
    OUTPUT = tinyint_out,
);
```

And that's it...

Custom Data Types

- Should add operators

```
Datum
tinyint_eq(PG_FUNCTION_ARGS)
{
  PG_RETURN_BOOL(PG_GETARG_TINYINT(0) == PG_GETARG_TINYINT(1));
}
```

Extensions

- "pg_contrib"
  - additional supplied modules
  - some provide additional data types outside of core
- Postgres 9.1+
  - CREATE EXTENSION "extension-name";
- Postgres <=9.0
  - psql -f path/to/contrib/install.sql yourdb
- must be database owner or superuser for both methods

cube

- data type for n-dimensional cubes
- stored as 64-bit floats
- CREATE EXTENSION cube;

```
SELECT '1'::cube;
SELECT '(1,2,3)'::cube;
SELECT '(1,2,3),(4,5,6)'::cube;
```

cube

- supports <, <=, =, >=, >, <>
- &&
  - cube overlap
- <@
  - a <@ b cube a is contained by cube b
- @>
  - a @> b cube a contains cube b

cube

- Indexing
  - B-tree
  - GiST

hstore

- key-value store in PostgreSQL
- stores keys and values as strings
- installation
  - "CREATE EXTENSION hstore"

```
SELECT 'jk=>1, jm=>2'::hstore;
--------------------
 "jk"=>"1", "jm"=>"2"
```

hstore

```
SELECT hstore(ARRAY['jk', 'jm'], ARRAY['1',
  '2']);
----------------------
 "jk"=>"1", "jm"=>"2"

SELECT hstore(ARRAY['jk', '1', 'jm', '2']);
----------------------
 "jk"=>"1", "jm"=>"2"

SELECT hstore(ROW('jk', 'jm'));
------------------------
 "f1"=>"jk", "f2"=>"jm"
```

hstore

```
SELECT ('jk=>1, jm=>2'::hstore) -> 'jk';
----------
 1

SELECT ('jk=>1, jm=>2'::hstore) -> ARRAY['jk','jm'];
----------
 {1,2}

SELECT delete('jk=>1, jm=>2'::hstore, 'jm');
-----------
 "jk"=>"1"
```

hstore

```
SELECT ('jk=>1, jm=>2'::hstore) @> 'jk=>1'::hstore;
----------
 t

SELECT ('jk=>1, jm=>2'::hstore) ? 'sf';
----------
f

SELECT ('jk=>1, jm=>2'::hstore) ?& ARRAY['jk', 'sf'];
----------
f

SELECT ('jk=>1, jm=>2'::hstore) ?| ARRAY['jk', 'sf'];
----------
t
```

hstore

```
SELECT hstore_to_array('jk=>1, jm=>2'::hstore);
-----------------
 {jk,1,jm,2}

SELECT hstore_to_matrix('jk=>1, jm=>2'::hstore);
------------------
 {{jk,1},{jm,2}}

SELECT hstore_to_json('jk=>1, jm=>2'::hstore);
------------------------
 {"jk": "1", "jm": "2"}

SELECT hstore_to_json_loose('jk=>1, jm=>2'::hstore);
----------------------
 {"jk": 1, "jm": 2}
```

hstore

- akeys, avals
  - array
- skeys, svals
  - set
- each
  - set of all keys + valls
- slice
  - similar to "hstore -> ARRAY[]"
- delete

```
SELECT delete('jk=>1, jm=>2'::hstore, 'jm');
-----------
 "jk"=>"1"
```

hstore

- supports GiST and GIN indexes
    - @>, ?, ?&, ?|
- supports B-tree and hash indexes
    - "=" comparisons
    - enables UNIQUE hstore columns
    - DISTINCT, GROUP BY, ORDER BY

Stretch Break #3

- Exercise:
  – come up with ideas for new data type extensions

Just Kidding…Conclusion

- There are a ***lot*** of data types in PostgreSQL
- …and if there are not enough, you can create more

References

- PostgreSQL 9.2
  - http://www.postgresql.org/docs/current/static/index.html
- PostgreSQL 9.3beta1
  - http://www.postgresql.org/docs/devel/static/index.html
- Other talks
  - https://wiki.postgresql.org/images/4/46/Knn.pdf
  - https://wiki.postgresql.org/images/f/f0/Range-types.pdf

Contact

- Jonathan S. Katz
  - jonathan.katz@excoventures.com
  - @jkatz05
- Jim Mlodgenski
  - jim@stormdb.com
  - @jim_mlodgenski
- Feedback please!
  - https://papers.pgcon.org/feedback/PGCon2013/event/633.en.html