Range Types:
Your Life Will Never Be The Same

Jonathan S. Katz
CTO, VenueBook
September 18, 2012
What’s in a Range?

• Conference schedule
• Pick a number from 1-10
  – Integer or real?
• Budget for buying a new laptop
Ranges are Everywhere...

- Scheduling
- Probability
- Intersections of ordered data
How Do We Deal With Ranges?

CREATE TABLE employee_schedule (  
    id serial,
    employee_id integer REFERENCES employees(id),
    start_time timestamptz,
    end_time timestamptz
);
Who is on duty at...

```
SELECT *
FROM employee_schedule
WHERE
  employee_id = 24 AND
  CURRENT_TIMESTAMP BETWEEN start_time AND end_time;

-- start_time <= CURRENT_TIMESTAMP <= end_time
```
Can I schedule an employee shift?

• Easy!

SELECT EXISTS(id)
FROM employee_schedule
WHERE
  employee_id = 24 AND
  ( '2012-09-18 10:00' <= start_time AND
    '2012-09-18 11:00' >= start_time AND
    '2012-09-18 10:00' <= end_time AND
    '2012-09-18 11:00' <= end_time
  ) OR ( ...wait this is really hard)
Why Overlaps Are Difficult
In PostgreSQL 9.1, Can I...

- Use a built-in function to determine if my ranges overlap?
- Easily create a composite type and add logic to recognize the ranges?
- Change to a different database software that makes the problem easier?
...can someone smarter than me make my life easier?
...Yes!!!

Support range data types.

Selectivity estimation functions are missing for some range type operators, which is a TODO.

Jeff Davis
Built-In Ranges

- INT4RANGE (integer)
- INT8RANGE (bigint)
- NUMRANGE (numeric)
- TSRANGE (timestamp without time zone)
- TSTZ RANGE (timestamp with time zone)
- DATERANGE (date)
Range Bounds

• Ranges can be inclusive, exclusive or both

• Math review:
  - \([2, 4]\) => \(2 \leq x \leq 4\)
  - \([2, 4)\) => \(2 \leq x < 4\)
  - \((2, 4]\) => \(2 < x \leq 4\)
  - \((2, 4)\) => \(2 < x < 4\)

• Can also be empty
Ranges...Unbound

• Ranges can be infinite
  – \([2, \infty) = 2 \leq x < \infty\)
  – \((-\infty, 2] = -\infty < x \leq 2\)

• CAVEAT EMPTOR
  – “infinity” has special meaning with timestamp ranges
    – \([\text{today}, \infty) = [\text{today}, \infty)\)
    – \([\text{today}, \text{‘infinity’}) <> [\text{today}, \text{‘infinity’}]\)
Constructing Ranges

• Simple!

test=# SELECT '[:1,10]'::int4range;

    int4range
   --------
    [1,11)
   (1 row)
test=# SELECT '2012-03-28, 2012-04-02'::daterange;

daterange
-------------------------
[2012-03-28,2012-04-03)
(1 row)
Constructing Ranges

• Constructor functions too
  – Defaults to ‘[]’

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

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

test=# SELECT tsrange('2012-04-01 00:00:00', '2012-04-01 12:00:00', '[]');

    tsrange
-------------------------------
["2012-04-01 00:00:00","2012-04-01 12:00:00"]
(1 row)
Using Ranges

• Normal comparison operations

```
SELECT int4range(100,200) = int4range(100,200);
-- true
SELECT int4range(100,200) <> int4range(200,300);
-- true
SELECT int4range(100,200) < int4range(200,300);
-- true
SELECT int4range(100,200) <= int4range(200,300);
-- true
SELECT int4range(100,200) >= int4range(200,300);
-- false
SELECT int4range(100,200) > int4range(200,300);
-- false
```
Why Your Life Will Change

• Let’s see the magic with an example

• Shopping for a used car
  – Cars listed with a price range
  – Have a min/max budget
## Inspect Our Data

- Sort by range lower bound

```sql
test=# SELECT * FROM cars ORDER BY lower(cars.price_range);
```

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>price_range</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Buick Skylark</td>
<td>[2000,4001)</td>
</tr>
<tr>
<td>3</td>
<td>Pontiac GTO</td>
<td>[5000,7501)</td>
</tr>
<tr>
<td>4</td>
<td>Chevrolet Camero</td>
<td>[10000,12001)</td>
</tr>
<tr>
<td>5</td>
<td>Ford Mustang</td>
<td>[11000,15001)</td>
</tr>
<tr>
<td>6</td>
<td>Lincoln Continental</td>
<td>[12000,14001)</td>
</tr>
<tr>
<td>7</td>
<td>BMW M3</td>
<td>[35000,42001)</td>
</tr>
<tr>
<td>8</td>
<td>Audi RS4</td>
<td>[41000,45001)</td>
</tr>
<tr>
<td>9</td>
<td>Porsche 911</td>
<td>[47000,58001)</td>
</tr>
<tr>
<td>10</td>
<td>Lamborghini LP700</td>
<td>[385000,400001)</td>
</tr>
</tbody>
</table>

(9 rows)
Car Shopping: Conceptually Simple
Car Shopping: Nightmarishly Complicated

- Budget of $13,000 - $15,000, find cars price in that range

```sql
SELECT *
FROM cars
WHERE
  (cars.min_price ≤ 13000 AND cars.min_price ≤ 15000 AND cars.max_price ≥ 13000 AND cars.max_price ≤ 15000)
OR
  (cars.min_price ≤ 13000 AND cars.min_price ≤ 15000 AND cars.max_price ≥ 13000 AND cars.max_price ≥ 15000)
OR
  (cars.min_price ≥ 13000 AND cars.min_price ≤ 15000 AND cars.max_price ≥ 13000 AND cars.max_price ≤ 15000)
OR
  (cars.min_price ≥ 13000 AND cars.min_price ≤ 15000 AND cars.max_price ≥ 13000 AND cars.max_price ≥ 15000)
)
ORDER BY cars.min_price;
```
Car Shopping: Magically Painless

• Budget of $13,000 - $15,000, find cars price in that range

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

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>price_range</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>Ford Mustang</td>
<td>[11000,15001)</td>
</tr>
<tr>
<td>6</td>
<td>Lincoln Continental</td>
<td>[12000,14001)</td>
</tr>
</tbody>
</table>

(2 rows)
In more details

• &&
  
  – the “overlap” operator
  
  – take two ranges: [x,y] and [a,b]

\[(a \leq x \ \text{AND} \ a \leq y \ \text{AND} \ b \geq x \ \text{AND} \ b \leq y) \ \text{OR} \]
\[(a \leq x \ \text{AND} \ a \leq y \ \text{AND} \ b \geq x \ \text{AND} \ b \geq y) \ \text{OR} \]
\[(a \geq x \ \text{AND} \ a \leq y \ \text{AND} \ b \geq x \ \text{AND} \ b \leq y) \ \text{OR} \]
\[(a \geq x \ \text{AND} \ a \leq y \ \text{AND} \ b \geq x \ \text{AND} \ b \geq y) \]

(Math for the win: inverse only two lines)
The Saver

• Find cars whose price does not exceed $13,000

```
SELECT * 
FROM cars 
WHERE cars.price_range << int4range(13000, 15000) 
ORDER BY lower(cars.price_range);
```

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>price_range</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Buick Skylark</td>
<td>[2000,4001]</td>
</tr>
<tr>
<td>3</td>
<td>Pontiac GTO</td>
<td>[5000,7501]</td>
</tr>
<tr>
<td>4</td>
<td>Chevrolet Camero</td>
<td>[10000,12001]</td>
</tr>
</tbody>
</table>

The Cautious

- Budget of $13,000 - $15,000, but want to see cheaper options

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

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>price_range</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Buick Skylark</td>
<td>[2000,4001)</td>
</tr>
<tr>
<td>3</td>
<td>Pontiac GTO</td>
<td>[5000,7501)</td>
</tr>
<tr>
<td>4</td>
<td>Chevrolet Camaro</td>
<td>[10000,12001)</td>
</tr>
<tr>
<td>5</td>
<td>Ford Mustang</td>
<td>[11000,15001)</td>
</tr>
<tr>
<td>6</td>
<td>Lincoln Continental</td>
<td>[12000,14001)</td>
</tr>
</tbody>
</table>

(5 rows)
The Dreamer

- Budget of $13,000 - $15,000, but want to see what lies beyond…

```sql
SELECT *
FROM cars
WHERE cars.price_range >> int4range(13000, 15000)
ORDER BY lower(cars.price_range);
```

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>price_range</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>BMW M3</td>
<td>[35000,42001)</td>
</tr>
<tr>
<td>8</td>
<td>Audi RS4</td>
<td>[41000,45001)</td>
</tr>
<tr>
<td>9</td>
<td>Porsche 911</td>
<td>[47000,58001)</td>
</tr>
<tr>
<td>10</td>
<td>Lamborghini LP700</td>
<td>[385000,400001)</td>
</tr>
</tbody>
</table>

(4 rows)
Determine Negotiating Window

- For cars in my budget, what prices am I looking at?

```
SELECT *
  , cars.price_range * int4range(13000, 15000) AS price_window
FROM cars
WHERE cars.price_range && int4range(13000, 15000)
ORDER BY lower(cars.price_range);
```

```
<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>price_range</th>
<th>price_window</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>Ford Mustang</td>
<td>[11000,15001)</td>
<td>[13000,15000)</td>
</tr>
<tr>
<td>6</td>
<td>Lincoln Continental</td>
<td>[12000,14001)</td>
<td>[13000,14001)</td>
</tr>
</tbody>
</table>

(2 rows)
```
Are Range Queries Fast?

• Well...

QUERY PLAN

---------------------------------------------
Sort  (cost=11.76..11.77 rows=1 width=552)
  Sort Key: (lower(price_range))
  -> Seq Scan on cars  (cost=0.00..11.75 rows=1 width=552)
    Filter: (price_range && '[13000,15001))::int4range)

• But wait, I didn’t add any indexing!
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)

• Note: I used a more populous table to make the index scan to occur
Scheduling

• ...now is super easy*
• Unique constraints to save the day!
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 ('Boston', daterange('2012-03-07', '2012-03-09'));
INSERT INTO travel_log (name, trip_range) VALUES ('Chicago', daterange('2012-03-12', '2012-03-17'));

Scheduling
Scheduling

test=# 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]).

• Easy.
And That’s Not All!

- Ranges can be extended – I kid you not

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
In the Wild?
For More Information

- http://wiki.postgresql.org/wiki/RangeTypes
Conclusion

• If you are not completely smitten by range types, then I have failed at explaining them

• Upgrade to PostgreSQL 9.2 – now.
  – (or this coming Monday [9/24/2012])
Thanks To...

• Jeff Davis for implementing range types
• Alexander Korotkov for GiST improvements for handling range type data
Contact

• Jonathan S. Katz
• jonathan@venuebook.com
• @jkatz05