Ranges, Partitioning, and Limitations

PGConf US 2016
Corey Huinker
What is this talk about?

An overview of what Range Types are and what they can do.

A series of gripes about what they can't do.

Cool uses for Range Types in my work at Moat (http://moat.com).
Why are Range Types Important?

- They allow your data to more accurately convey meaning.
- They allow your code to more accurately convey intention.
- Indexability, Exclusion constraints
- No other RDBMS has them [1], giving PostgreSQL an expressive advantage.

Range Basics: Bounds

Ranges behave like and are denoted by standard mathematical Interval Notation.

<table>
<thead>
<tr>
<th>Notation</th>
<th>Means</th>
<th>Notation</th>
<th>Means</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(x$</td>
<td>values $&gt;$ $x$</td>
<td>$[x$</td>
<td>values $\geq$ $x$</td>
</tr>
<tr>
<td>$y)$</td>
<td>values $&lt;$ $y$</td>
<td>$y]$</td>
<td>values $\leq$ $y$</td>
</tr>
<tr>
<td>$(,)$</td>
<td>No lower bound</td>
<td>$,)$</td>
<td>No upper bound</td>
</tr>
<tr>
<td>$(,)$</td>
<td>everything</td>
<td>empty $</td>
<td>$ No values</td>
</tr>
</tbody>
</table>
Constructing Ranges

Casting from text:

```sql
select '[low,high])::rangetype
select '[low,)::rangetype
```

Creation through constructor function

```sql
select rangetype(low,high,'[]')
select rangetype(null,high,'[]')
```

Note: no polymorphic constructor

```sql
select to_range(null::rangetype,low,high,'[]');
```

- Omitting a bound means unbounded, regardless of inc/excl
- Nulling a bound is the same as omitting it.
Range Basics: Existing Types

- `int4range`: Range of integer
- `int8range`: Range of bigint
- `numrange`: Range of numeric
- `tsrange`: Range of timestamp without time zone
- `tstzrange`: Range of timestamp with time zone
- `daterange`: Range of date
- `boolrange`: Range of boolean
- `textrange`: Range of text
Why no `textrange` type?

- Collation Sequences.
  - Would need one `textrange` per collation sequence.
- No telling how many collations are installed.
  - Or what order they were installed in.
- Need one oid per range type, just like any other type.
- Would have to pre-allocate them with static type definitions.
- Not going to burn that many oids on a bunch of maybes.
  - So just define one type per collation sequence that you'll need
    - You probably only need "C" and maybe one other.

```sql
create type textrange_c as range (subtype = text, collation = "C");
```
Attribute functions:

Ranges can be decomposed into their component attributes.

```sql
# create temp table temps(state text, rng numrange);
# insert into temps values ('ice',        '(',32.0)'),
#                        ('water',      numrange(32.0,212.0,'[]'))),
#                        ('steam',      numrange(212.0,null)),
#                        ('heat death', 'empty');

# select * from temps;

<table>
<thead>
<tr>
<th>state</th>
<th>rng</th>
</tr>
</thead>
<tbody>
<tr>
<td>ice</td>
<td>(,32.0)</td>
</tr>
<tr>
<td>water</td>
<td>[32.0,212.0)</td>
</tr>
<tr>
<td>steam</td>
<td>[212.0,)</td>
</tr>
<tr>
<td>heat death</td>
<td>empty</td>
</tr>
</tbody>
</table>
```
Null display is "¤". This is really useful when sharing examples, but might be confusing if you think that's a currency symbol.

<table>
<thead>
<tr>
<th>state</th>
<th>low</th>
<th>low_inc</th>
<th>low_inf</th>
<th>upper</th>
<th>upper_inc</th>
<th>upper_inf</th>
<th>empty</th>
</tr>
</thead>
<tbody>
<tr>
<td>ice</td>
<td>❌</td>
<td>❌</td>
<td>❌</td>
<td>❌</td>
<td>❌</td>
<td>❌</td>
<td>❌</td>
</tr>
<tr>
<td>water</td>
<td>32.0</td>
<td>❌</td>
<td>❌</td>
<td>❌</td>
<td>❌</td>
<td>❌</td>
<td>❌</td>
</tr>
<tr>
<td>steam</td>
<td>212.0</td>
<td>❌</td>
<td>❌</td>
<td>❌</td>
<td>❌</td>
<td>❌</td>
<td>❌</td>
</tr>
<tr>
<td>heat death</td>
<td>❌</td>
<td>❌</td>
<td>❌</td>
<td>❌</td>
<td>❌</td>
<td>❌</td>
<td>❌</td>
</tr>
</tbody>
</table>

(4 rows)
Operators: =, <>

Discrete ranges normalize to the [) bound via the defined *canonical* function, and are then tested for equivalence. Continuous ranges do not have a *canonical* function, and are tested as-is.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>select '((1,10]'::int4range;</td>
<td>[2,11)</td>
</tr>
<tr>
<td>select '[yesterday,today]'::daterange = '[yesterday,tomorrow)'::daterange;</td>
<td>t</td>
</tr>
<tr>
<td>select '[1,3]'::numrange = '[1,4)'::numrange;</td>
<td>f</td>
</tr>
<tr>
<td>select '[1,3]'::numrange = '[1,3.00000000000000000001)'::numrange;</td>
<td>f</td>
</tr>
</tbody>
</table>
Operators: <, <=, >, >=

- Test lower bound scalar first, then use upper bound as a tiebreaker
  - Which isn't really intuitive, but then again neither are the alternatives:
    - Median?
    - Number of (discrete) values contained?
- Therefore, not generally useful for userland queries.
- Used internally for indexing.
Operator $\ll$

- "Strictly to the left of"
- $a \ll b$ if normalized upper bound of $a$ is $<$ normalized lower bound of $b$

```sql
# select '[1,3)::int4range << '[3,5)::int4range as a1,
    '[1,3)::int4range << '[3,5)::int4range as a2;

a1 | a2
----+----
  t  |  f
```
Operator >>

- "Strictly to the right of"
- $a >> b$ if normalized lower bound of $a$ is $>$ normalized upper bound of $b$

# select '([today,tomorrow])::daterange >>

  '([yesterday,today])::daterange as a1,

  '([today,tomorrow])::daterange >>

    '([yesterday,today])::daterange as a2;

<table>
<thead>
<tr>
<th>a1</th>
<th>a2</th>
</tr>
</thead>
<tbody>
<tr>
<td>t</td>
<td>f</td>
</tr>
</tbody>
</table>

------
Operator &<

- "Does not extend to the right of"
- No element of $a$ is $>$ greatest element of $b$

```python
# select daterange('[today,tomorrow)') &<
    daterange('[yesterday,today)') as x,
    int4range('[10,20)') &< int4range('[10,20]') as y;

  x | y
----+---
  f | t
(1 row)
```
Operator &>

- "Does not extend to the left of"
- No element of $a$ is $<$ least element of $b$

```sql
# select '[3,10))::int4range &> '[1,4))::int4range as x,
    '[0,10))::int4range &> '[1,4))::int4range as y;

  x | y
---+---
  t | f
```
Operator - | -

- "adjacent"
- There is no overlap nor space between a and b.
- It doesn't matter which range is lower

```sql
# select '[4,10]':int4range -|[1,4]':int4range as x,  
  '[1,3]':int4range -|[5,10]':int4range as y,  
  '[1,10]':int4range -|[5,15]':int4range as z;

  x | y | z
---+---+---
t | f | f
```
Operators `<@` and `@>`

- "contains", same as the geometric operators
- The value or range on the pointy side fits entirely within the range on the `@` side
- It doesn't matter which range is lower

```sql
# select 1 <@ '[1,4]::int4range as u,
  '[20,30]::int4range <@ '[1,100]::int4range as v,
  'infinity'::date <@ '(,)::daterange as w,
  '(,)::int4range @> 'empty'::int4range as x,
  '(,)::int4range @> null as y;
```

<table>
<thead>
<tr>
<th>u</th>
<th>v</th>
<th>w</th>
<th>x</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>t</td>
<td>t</td>
<td>t</td>
<td>t</td>
<td>✗</td>
</tr>
</tbody>
</table>
Operator &&

- "overlap", same as the geometric operator
- At least one value can fit in both ranges

```sql
# select '[20,30))::int4range && '[1,100))::int4range as v,
   '(,):int4range && 'empty)::int4range as x;

 v | x
---+---
t | f
```

```sql
# select 'empty)::int4range <@ '(,):int4range as v,
   'empty)::int4range && '(,):int4range as x;

 v | x
---+---
t | f
```
Operator + (and the range_merge() function)

- Union: All elements in both, if there are no gaps

```sql
# select int4range(1,4) + int4range(2,10) as x;
   x
----------
[1,10)

# select int4range(1,2) + int4range(99,100) as y;
ERROR: result of range union would not be contiguous

# select range_merge(int4range(1,2),int4range(99,100)) as z;
   z
----------
[1,100)
```

New in 9.5!
Available for earlier version in range_type_functions on PGXN
Operator *

- Intersection: all elements in common, if any

```python
# select int4range(1,4) * int4range(4,100) as x,
    int4range(1,4,'[]') * int4range(4,100) as y;
      x  |  y
-----+-----
empty | [4,5)
```
Operator –

- Difference: all elements in a but not in b
- Will raise an error if the difference would return 2 disjoint sets

```sql
# select int4range(1,100) - int4range(1,10) as x;
  x
----------
  [10,100)

# select int4range(1,100) - int4range(2,10) as x;
ERROR:  result of range difference would not be contiguous
```
Missing Function: range_split()

- Same as the – operator, but returning the left side remainder and right side remainder
- returns an array of the resulting ranges
- a SRF would be nice too.

```sql
hypothetical
# select range_split('[1,100]::int4range,
                      '[2,4]::int4range) as x;

 x
---------------------
{[1,2), [2,5), [5,100]}
```
Missing Operators =|, |=

Operators to test whether two ranges share a lower (|=) bound or upper bound (|=)

hypothetical# select '[1,4]::int4range =| '[1,10]::int4range as w,' [1,4]::int4range =| '(1,10]::int4range as x,' [1,4]::int4range |= '(-,4]::int4range as y,' [1,4]::int4range |= '(-,4)']::int4range as z;

<table>
<thead>
<tr>
<th>w</th>
<th>x</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
</table>
t | f | t | f
Missing Operators: elem <<, >>

- Same as the current <</>> operators, but allow the one arg to be a scalar.
- May be a problem for existing bitshift operators

```sql
hypothetical# select 1::integer << '1,10]::int4range as w,
                        1::integer << '(1,4]::int4range as x,
                        4::integer >> '1,4]::int4range as y,
                        4::integer >> '4) as z;
```

| w | x | y | z |
|---+---+---+---|
| f | t | f | t |

Can be simulated by creating a singleton range:

```sql
int4range(1,1,'[]') << int4range(2,11,'[]')
```
Missing Operator: `elem <=> range`

- Returns 0 if element `a <@ range b`.
- -1 if `a << b`, 1 if `a >> b`.
- basically `strcmp()` but for ranges.

```sql
hypothetical# select 1::integer <=> ' [1,10] '::int4range as w,
                  1::integer <=> '(1,4] '::int4range as x,
                  4::integer <=> ' [1,4] '::int4range as y,
                  4::integer <=> '(,4) '::int4range as z;
```

<table>
<thead>
<tr>
<th>w</th>
<th>x</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Implemented as `element_range_comp()` in `range_type_functions` on PGXN.
Missing Functions: is_singleton()

- Return true if the range can contain only one element.

```sql
# select is_singleton('[4,5]'::int4range);
  is_singleton
------------
   t

# select is_singleton('[4,5]'::int4range);
  is_singleton
------------
   f
```

Found in
range_type_functions on
PGXN
Missing Functions: get bounds

- Represent either or both bounds conditions as SQL
- Helpful when constructing \texttt{CHECK} / \texttt{WHERE} clauses or dealing with foreign systems that don't support that range type or ranges in general.

```sql
# with t(c) as (values('[1,4]'::int4range))
select get_lower_bound_condition_expr(c) as l,
       get_upper_bound_condition_expr(c) as u,
       get_bounds_condition_expr(c,'zz') as b from t;
```

<table>
<thead>
<tr>
<th>l</th>
<th>u</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{x &gt;= '1'::integer}</td>
<td>\texttt{x &lt; '5'::integer}</td>
<td>\texttt{zz &gt;= '1'::integer and zz &lt; '5'::integer}</td>
</tr>
</tbody>
</table>

Found in `range_type_functions` on PGXN
Partitioning by Ranges Use Case

Use case is a series of "typeahead search" tables:

● Hundreds of millions of rows.
● Grouped by a taxonomy of 5 text strings of increasing length.
● The searchable text is usually 5-20 words per record.
● Need a way to partition the table, but only text types available.
● Distribution is highly uneven along strict alphabetical lines.
Text Range Partitioning Advantages

- Partitions have smaller GIN indexes on the searchable columns, so smaller BitmapAnd steps.
- Ability to isolate very large clients.
- Search dataset evolves over time the lumps in the data move, but slowly.
- Partition maintenance only when data is starting to skew, much different from timeseries.

```sql
create type textrange_murica as range (subtype = text,
  collation = "en_US");
```
range_partitioning module

- On PGXN
- Functions closely match those in pg_partman.
  - `create_parent(table, column_name)`
    - starts with implied range of (,)
  - `create_partition(table, new_range)`
    - new partition range must be perfect subset of an existing range, and match lower or upper bound.
  - `drop_partition(lost_part, kept_part)`
    - merge all data from lost_part into kept_part
range_partitioning module

- SELECT / INSERT / UPDATE queries are transparent.
- Does trigger function for transparent INSERT.
- Probably better having bulk loads separated by partitioned value, and probing for the destination partition with `get_destination_partition()`, if possible.
- The `create_parent()` function cannot seamlessly derive the base type if more than one range type has that base type.
- Ranges are specified as un-casted text strings.
range_partitioning example

Use case: Message board for fans of TV shows. The site's users skew heavily towards certain niche shows.[1]

/* Turn existing table into a parent table. One partition with range (,) */
select range_partitioning.create_parent('public.spoiler_alerts',
   'tv_show_name');

/* Create a partition just for the show ARCHER, but all new partitions must share an edge with an existing partition, so you may need to explicitly create more than one */
select range_partitioning.create_partition('public.spoiler_alerts',
   '(,ARCHER)');
select range_partitioning.create_partition('public.spoiler_alerts',
   '[ARCHER,ARCHER]');

[1] The niche is defined as "Shows I can name".
range_partitioning example (part 2)

/* Create a partition that covers DAREDEVIL, FAMILY_GUY, and some others */
select range_partitioning.create_partition('public.spoiler_alerts',
   '(ARCHER,GAME_OF_THRONES]');

/* Create a partition just for the show RICK_AND_MORTY, again sharing an edge */
select range_partitioning.create_partition('public.spoiler_alerts',
   '(RICK_AND_MORTY,)');
select range_partitioning.create_partition('public.spoiler_alerts',
   '[RICK_AND_MORTY,RICK_AND_MORTY]');
range_partitioning: partition list

```sql
# select partition_number, range
from range_partitioning.partition
where master_class = 'public.spoiler_alerts'::regclass;

<table>
<thead>
<tr>
<th>partition_number</th>
<th>range</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(GAME_OF_THRONES,RICK_AND_MORTY)</td>
</tr>
<tr>
<td>1</td>
<td>(,ARCHER)</td>
</tr>
<tr>
<td>2</td>
<td>[ARCHER,ARCHER]</td>
</tr>
<tr>
<td>3</td>
<td>(ARCHER,GAME_OF_THRONES)</td>
</tr>
<tr>
<td>4</td>
<td>(RICK_AND_MORTY,)</td>
</tr>
<tr>
<td>5</td>
<td>[RICK_AND_MORTY,RICK_AND_MORTY]</td>
</tr>
</tbody>
</table>
```
range_partitioning type discovery

The `create_parent(table, column)` function doesn't need to have the range type specified if only one range type would work for that column.

```sql
/* if this returns more than one row, then we have to specify a range type */
select  rt.rngtypid
from    pg_attribute a
join    pg_range rt
on      rt.rngsubtype = a.atttypid
and     rt.rngcollation = a.attcollation
where   a.attrrelid = 'my_schema.my_parent_table'::regclass
and     a.attname = 'my_partitioning_column';
```
Complex Range Partitioning

• Possible to partition on ranges of complex types
  ○ That complex type must exist in the table itself, it can't be more than one column
  ■ So re-expose the components in a view.

# create type quite_complex as (a text collate "C", b text collate "C", c text collate "C", d text collate "C");

CREATE TYPE
# create type qc_range as range (subtype = quite_complex);
CREATE TYPE
# select '[("Abel,Baker,Charlie,Delta's"),"(Walter,X-Ray,Yellow,)")':
qc_range;

qc_range

-----------------------------------------------
["(Abel,Baker,Charlie,Delta's"),"(Walter,X-Ray,Yellow,)")]
Future Direction: range_partitioning

- Add functions to predict proper partition ranges for equal-ish row counts
  - width_buckets() works ok, but will sometimes skip some buckets entirely. You ask for 16 partitions, get 13.
- Add functions to analyze existing partitions for skew
- Become obsolete.
  - Native Partitions coming to PostgreSQL in 9.7, probably.
  - Existing work supports ranges but not range syntax.
Links

Range Partitioning extension:
   PGXN: http://pgxn.org/dist/range_partitioning/
   GitHub: https://github.com/moat/range_partitioning

Range Type Functions:
   PGXN: http://pgxn.org/dist/range_type_functions/
   GitHub: https://github.com/moat/range_type_functions