Ranges, Partitioning, and Limitations

PgConf Asia 2017: Tokyo
December 2017
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.

A look at what is [not] possible with PostgreSQL Native Partitioning in Version 10, and a look to the future.

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.

[1] - I haven't looked too hard [2].
[2] - A year later, I still haven't found anything [3].
[3] - Nope, still haven't.

Range Basics: Bounds

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

Notation	Means	Notation	Means
`(x`	values > *x*	`[x`	values >= *x*
`y)`	values < *y*	`y]`	values <= y
`(,`	No lower bound	`,)`	No upper bound
`(,)`	everything	`empty`	No values

Constructing Ranges

Casting from text:

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

Omitting a bound means unbounded, regardless of inc/excl

Creation through constructor function

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

Nulling a bound is the same as omitting it.

Note: no polymorphic constructor

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

NOPE

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 on 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.

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

Attribute functions:

Ranges can be decomposed into their component attributes.

```
# create temp table temps(state text, rng numrange);
# insert into temps values ('ice',        '(,0.0)'),
                           ('water',      numrange(0.0,100.0,'[)')),
                           ('steam',      numrange(100.0,null)),
                           ('heat death', 'empty');
# select * from temps;
   state    |     rng
------------+--------------
 ice        | (,0.0)
 water      | [0.0,100.0)
 steam      | [100.0,)
 heat death | empty
```

Attribute functions In Action:

This is really useful when sharing examples, but might be confusing if you think that's a currency symbol.

```
# \pset null '¤'
Null display is "¤".
# select state, lower(rng) as low, lower_inc(rng) as low_inc,
lower_inf(rng) as low_inf, upper(rng), upper_inc(rng), upper_inf(rng),
isempty(rng) as empty from temps;

   state    |  low  | low_inc | low_inf | upper | upper_inc | upper_inf | empty
------------+-------+---------+---------+-------+-----------+-----------+-------
 ice        |     ¤ | f       | t       |   0.0 | f         | f         | f
 water      |   0.0 | t       | f       | 100.0 | f         | f         | f
 steam      | 100.0 | t       | f       |     ¤ | f         | t         | f
 heat death |     ¤ | f       | f       |     ¤ | f         | f         | t
(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.

Expression	**Result**
`select '(1,10]'::int4range;`	`[2,11)`

`select '[yesterday,today]'::daterange =`
`'[yesterday,tomorrow)'::daterange;`
	`t`

`select '[1,3]'::numrange = '[1,4)'::numrange;`	`f`

`select '[1,3]'::numrange =`
`'[1,3.0000000000000000001)'::numrange;`
	`f`

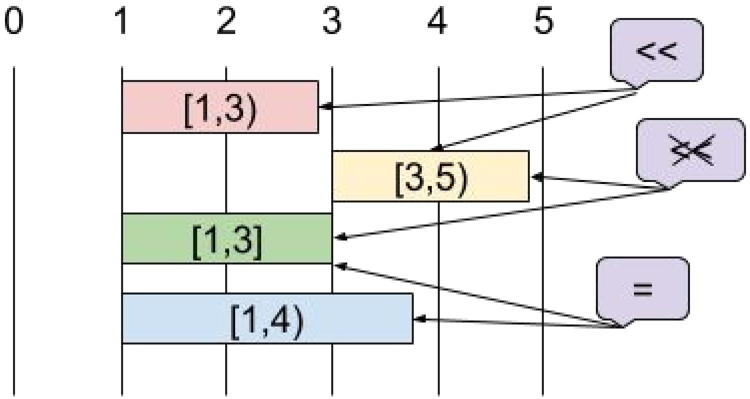
Operators: <, <=, >, >=

- 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 <<

- "Strictly to the left of" (i.e. no overlap)
- a << b if normalized upper bound of *a* is < normalized lower bound of *b*

```
# select '[1,3)'::int4range << '[3,5)'::int4range as a1,
         '[1,3]'::int4range << '[3,5)'::int4range as a2;
 a1 | a2
----+----
 t  | f
```



Operator >>

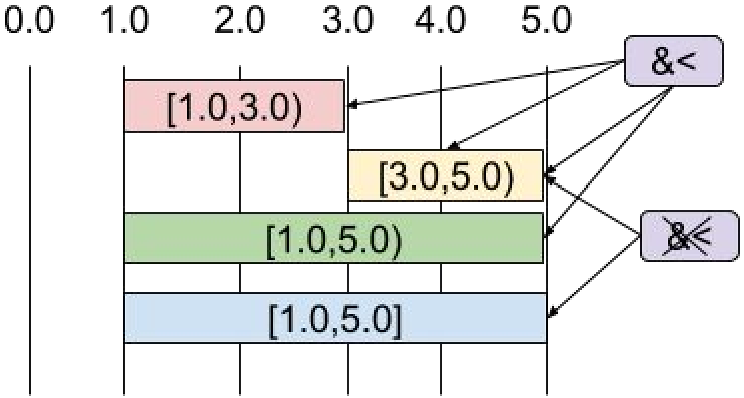
- "Strictly to the right of"
- a >> b if normalized lower bound of *a* is > normalized upper bound of *b*

```
# select '[today,tomorrow)'::daterange >>
             '[yesterday,today)'::daterange as a1,
         '[today,tomorrow)'::daterange >>
             '[yesterday,today]'::daterange as a2;
 a1 | a2
----+----
 t  | f
```

Operator `&<`

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

```
# select daterange('[today,tomorrow)') &<
                 daterange('[yesterday,today)') as x,
         int4range('[10,20)') &< int4range('[10,20]') as y;
 x | y
---+---
 f | t
(1 row)
```



Operator &>

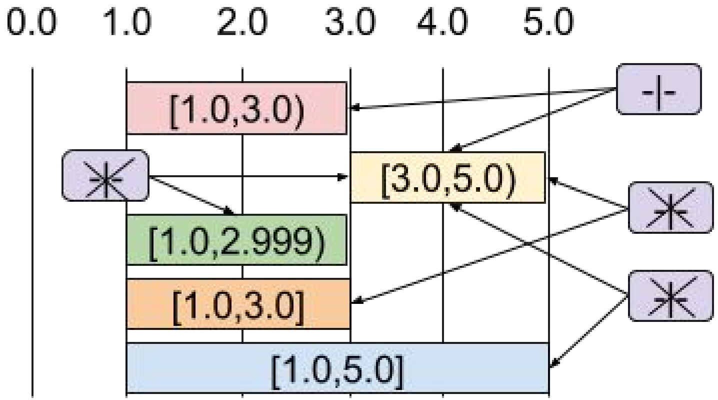
- "Does not extend to the left of"
- No element of *a* is < least element of *b*

```
# 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

```
# 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

```
# 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;

 u | v | w | x | y
---+---+---+---+---
 t | t | t | t | ¤
```

Operator `&&`

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

```
# select '[20,30)'::int4range && '[1,100]'::int4range as v,
         '(,)'::int4range     && 'empty'::int4range as x;
 v | x
---+---
 t | f

#  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

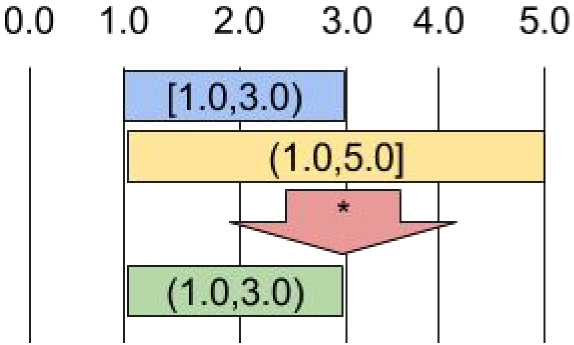
```
# select int4range(1,4) + int4range(2,10) as x;
   x
--------
 [1,10)

# 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 versions in range_type_functions on PGXN

Operator *

- Intersection: all elements in common, if any

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



Operator -

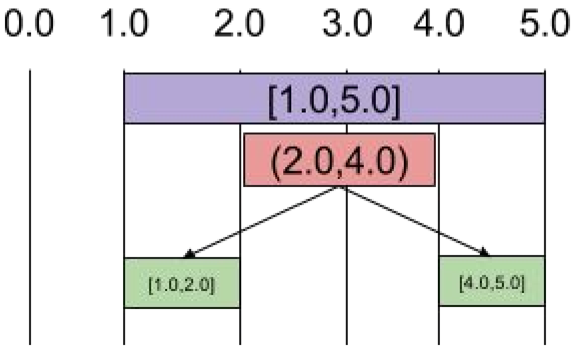
- Difference: all elements in *a* but not in *b*
- Will raise an error if the difference would return 2 disjoint sets

```
# 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.

```
hypotethical# 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 (|=)

```
hypotethical# 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;

 w | x | y | z
---+---+---+---
 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

```
hypotethical# 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;
```

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

Can be simulated by creating a singleton range:

```
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

```
hypotethical# 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;

 w |  x | y | z
---+----+---+---
 0 | -1 | 0 | 1
```

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.

```
# 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 `CHECK` / `WHERE` clauses or dealing with foreign systems that don't support that range type or ranges in general.

```
# 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;

         l         |         u          |                    b
-------------------+------------------+-----------------------------------------
 x >= '1'::integer | x < '5'::integer | zz >= '1'::integer and zz < '5'::integer
```

Found in range_type_functions on PGXN

Partitioning by Ranges Use Case

Use case is a series of very large data warehouse tables:

- Hundreds of millions of rows.
- Grouped by a taxonomy of 5 text strings of increasing length. Values are essentially enumerations.
- Need a way to partition the table, but only text types available.
- Distribution is highly uneven along strict alphabetical lines.
- Distributions change with the ebb and flow of customers' activity.

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.

```
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

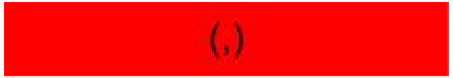
range_partitioning module

- `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');
```

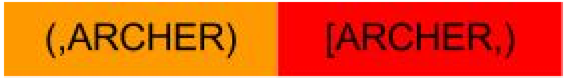


(,)

[1] The niche is defined as "Shows I can name".

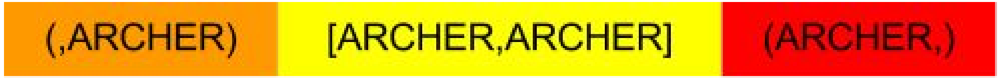
range_partitioning example

```
/* 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)');
```



(,ARCHER) [ARCHER,)

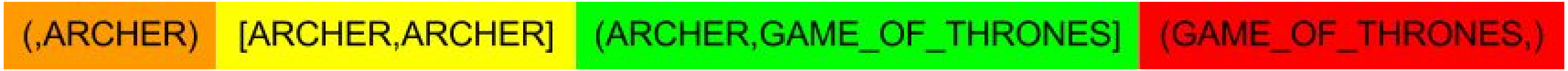
```
select range_partitioning.create_partition('public.spoiler_alerts',
                                           '[ARCHER,ARCHER]');
```



(,ARCHER) [ARCHER,ARCHER] (ARCHER,)

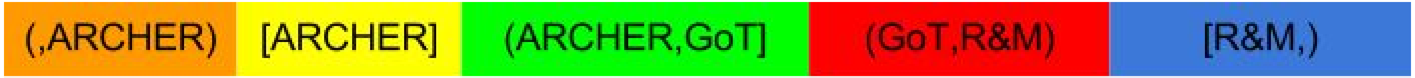
range_partitioning example (part 2)

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



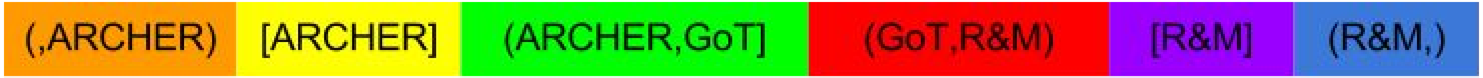
(,ARCHER) [ARCHER,ARCHER] (ARCHER,GAME_OF_THRONES] (GAME_OF_THRONES,)

```
/* 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,)');
```



(,ARCHER) [ARCHER] (ARCHER,GoT] (GoT,R&M) [R&M,)

```
select range_partitioning.create_partition('public.spoiler_alerts',
                                   '[RICK_AND_MORTY,RICK_AND_MORTY]');
```



(,ARCHER) [ARCHER] (ARCHER,GoT] (GoT,R&M) [R&M] (R&M,)

range_partitioning: partition list

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

 partition_number |               range
------------------+-----------------------------------
                0 | (GAME_OF_THRONES,RICK_AND_MORTY)
                1 | (,ARCHER)
                2 | [ARCHER,ARCHER]
                3 | (ARCHER,GAME_OF_THRONES]
                4 | (RICK_AND_MORTY,)
                5 | [RICK_AND_MORTY,RICK_AND_MORTY]
```

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.

```
/* 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.attrelid = '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,)")
```

New in 10.0: Native Range Partitions

```
# create table metrics (ad_date date, client text, ... )
  partition by range (ad_date, client) COLLATE "en_US";
# create table metrics_20170202 partition of metrics for values
('2017-02-02','GROUND') to ('2017-02-02','HOG');
```

- Ranges can be of composite types
- Partitions can themselves be partitioned.
  - Subpartitions don't have to be partitioned by the same thing
- Text can have collations sequences specified, so no more defining text subtypes.
- Ranges can only be of interval [a,b), [a,), (,z) or (,)

New in 10.0: Native Range Partitions

- No default values clauses for RANGE or LIST partitioning
- Inserts whose values have no valid partition will fail

```
$ CREATE TABLE h2o_temps (c numeric) PARTITION BY RANGE(c);
CREATE TABLE
$ SAVEPOINT p1;
SAVEPOINT
$ INSERT INTO h2o_temps VALUES (37.5);
ERROR:  no partition of relation "h2o_temps" found for row
DETAIL:  Partition key of the failing row contains (c) = (37.5).
$ ROLLBACK to p1;
ROLLBACK
$ CREATE TABLE water PARTITION OF h2o_temps FOR VALUES FROM (0.0) TO (100.0);
CREATE TABLE
$ INSERT INTO h2o_temps VALUES (1.0), (99.99999);
INSERT 0 2
```

New in 10.0: Native Range Partitions

- Impossible to have overlaps in coverage

```
$ SAVEPOINT p2;
SAVEPOINT
$ CREATE TABLE swimming_pool PARTITION OF h2o_temps FOR VALUES FROM (25.0) TO (29.0);
ERROR:  partition "swimming_pool" would overlap partition "water"
$ ROLLBACK TO p2;
ROLLBACK
```

New in 10.0: Native Range Partitions

Automatic tuple routing! No more triggers!
Even works for COPY

```
$ COPY h2o_temps FROM PROGRAM 'seq 99 102';
COPY 4
$ SELECT * FROM steam;
  c
-----
 100
 101
 102
(3 rows)

$ SELECT * FROM water;
    c
----------
        1
 99.99999
       99
(3 rows)
```

New in 10.0: Native Range Partitions

UPDATEs which would move a row to a new partition will fail.

```
$ SAVEPOINT p4;
SAVEPOINT
$ UPDATE h2o_temps
  SET c = c + 100.0
  WHERE c <@ numrange(99.0, 100.0, '()');
ERROR:  new row for relation "water" violates partition constraint
DETAIL:  Failing row contains (199.99999).
$ ROLLBACK TO p4;
ROLLBACK
```

New in 10.0: Native Range Partitions

Hacky workarounds possible for now

```
$ WITH hotter as ( DELETE FROM h2o_temps WHERE c <@ numrange(99.0, 100.0, '()') RETURNING * )
  INSERT INTO h2o_temps SELECT c + 100.00 FROM hotter;
INSERT 0 1
$ SELECT * FROM steam;
     c
-----------
       100
       101
       102
 199.99999
(4 rows)

$ SELECT * FROM water;
  c
-----
 1.0
  99
(2 rows)
```

Native Range Partition Maintenance

- Cannot split/alter range definition on a partition directly.
- But can detach the partition and re-attach with different range assignment (assuming data is valid for new range). This can be done in a transaction

Oracle has more partition operations but the aren't transactional.

```
$  INSERT INTO h2o_temps VALUES (10000.0);
INSERT 0 1
$ ALTER TABLE h2o_temps DETACH PARTITION steam;
ALTER TABLE
$ CREATE TABLE plasma PARTITION OF h2o_temps FOR VALUES FROM (3000.0) TO (MAXVALUE);
CREATE TABLE
$ WITH not_really_steam as ( DELETE FROM steam WHERE c >= 3000.0 RETURNING c )
  INSERT INTO plasma SELECT c FROM not_really_steam;
INSERT 0 1
$ ALTER TABLE h2o_temps ATTACH PARTITION steam FOR VALUES FROM (100.0) TO (3000.0);
ALTER TABLE
```

Future Direction: range_partitioning

- Become obsolete.
  - Only feature not covered by v10 is singleton [val1,val1] partitions
- Show features that should be in native partitioning in v11+
  - Splitting partitions and migrating rows

    ```
    ALTER TABLE foo SPLIT PARTITION foo_b FOR RANGE (...);
    CREATE TABLE foo PARTITION OF some_partitiong FOR RANGE(...) SPLIT EXISTING;
    ```

  - Generate values to implement [val1, val1+ lim(x) -> 0) partitions
- 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

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

Special thanks to MOAT (http://moat.com) for having hard problems that needed interesting solutions, and being willing to open source the bits that were generically useful.