Partitioning Improvements in PostgreSQL 11

Álvaro Herrera
alvherre@2ndQuadrant.com

2ndQuadrant Ltd.
http://www.2ndQuadrant.com/

PGConf Brasil 2018
http://pgconf.com.br/

2ndQuadrant PostgreSQL

Before Declarative Partitioning

- Early “partitioning” introduced in PostgreSQL 8.1 (2005)
- Heavily based on relation inheritance (from OOP)
- Novelty was “constraint exclusion”
  - a sort of “theorem prover” using queries and constraints
- Huge advance at the time

Example DDL

```
CREATE TABLE measurement (
    city_id int not null, logdate date not null,
    peaktemp int, unitsales int);

CREATE TABLE measurement_y2006m02 (
  CHECK ( logdate >= DATE '2006-02-01' AND
          logdate < DATE '2006-03-01' )
) INHERITS (measurement);

CREATE TABLE measurement_y2006m03 (
  CHECK ( logdate >= DATE '2006-03-01' AND
          logdate < DATE '2006-04-01' )
) INHERITS (measurement);
```

Example DDL (2)

```
CREATE OR REPLACE FUNCTION measurement_insert_trigger()
RETURNS TRIGGER AS $$
BEGIN
    IF ( NEW.logdate >= DATE '2006-02-01' AND
         NEW.logdate < DATE '2006-03-01' ) THEN
        INSERT INTO measurement_y2006m02 VALUES (NEW.*);
    ELSIF ( NEW.logdate >= DATE '2006-03-01' AND
            NEW.logdate < DATE '2006-04-01' ) THEN
        INSERT INTO measurement_y2006m03 VALUES (NEW.*);
    ELSIF ( ... )
        ...
    ELSE
        INSERT INTO measurement_default VALUES (NEW.*);
    END IF;
    RETURN NULL;
END;
$$;
```

2ndQuadrant PostgreSQL

Declarative Partitioning

- Introduced in PostgreSQL 10
- Easier to manage
- Better tuple routing performance

Declarative Partitioning DDL (Postgres 10)

```
CREATE TABLE orders (
   order_id BIGINT, order_date TIMESTAMP WITH TIME ZONE, ...
) PARTITION BY RANGE (order_date);

CREATE TABLE orders_2018_08 -- create empty partition
  PARTITION OF clientes FOR VALUES
  FROM ('2018-08-01') TO ('2018-08-31');

-- pre-filled table attached after the fact
ALTER TABLE orders
  ATTACH PARTITION orders_2018_01
  FOR VALUES FROM ('2018-01-01') TO ('2018-01-31');

-- No code needed for tuple routing!!
```

2ndQuadrant PostgreSQL

Decl. Partitioning: limitations

- Only `LIST` and `RANGE`
- No default partition
- Still using constraint exclusion
- Most DDL must be applied per partition
  - indexes, triggers
  - constraints (incl. foreign keys)
- some features don't work
  - ON CONFLICT DO UPDATE
  - UPDATE across partitions

Prelude to PostgreSQL 11

- Diversion: Change in version numbering
- Everybody now must know that versioning changed
- Must attend conferences every year!!

Partitioning in PostgreSQL 11

- New partitioning features
- Better support for DDL commands
- Performance optimizations

New Partitioning Features

- `DEFAULT` partition
- Row migration on `UPDATE`
- Hash partitioning
- `INSERT ON CONFLICT DO UPDATE`

New feature: DEFAULT partition

```
CREATE TABLE orders_def
  PARTITION OF orders
  FOR VALUES DEFAULT;
```

- Receives tuples for which there is no other partition
- Range partitioning: The default partition receives NULLs

New feature: DEFAULT partition

```
CREATE TABLE orders_def
  PARTITION OF orders
  FOR VALUES DEFAULT;
```

- Receives tuples for which there is no other partition
- Range partitioning: The default partition receives NULLs
- Please test!

New feature: Row migration on UPDATE

```
UPDATE orders SET order_date = '2018-08-02'
  WHERE order_date = '2018-07-31';
```

- Ability to move rows from one partition to another
- Hopefully not typical usage
- May have funny corner cases under concurrency

New feature: Row migration on UPDATE

```
UPDATE orders SET order_date = '2018-08-02'
  WHERE order_date = '2018-07-31';
```

- Ability to move rows from one partition to another
- Hopefully not typical usage
- May have funny corner cases under concurrency
- Please test!

New feature: hash partitioning

```
CREATE TABLE clientes (
  cliente_id INTEGER, ...
) PARTITION BY HASH (cliente_id);

CREATE TABLE clientes_0 PARTITION OF clientes
  FOR VALUES WITH (MODULUS 3, REMAINDER 0);
CREATE TABLE clientes_1 PARTITION OF clientes
  FOR VALUES WITH (MODULUS 3, REMAINDER 1);
CREATE TABLE clientes_2 PARTITION OF clientes
  FOR VALUES WITH (MODULUS 6, REMAINDER 2);
CREATE TABLE clientes_2 PARTITION OF clientes
  FOR VALUES WITH (MODULUS 6, REMAINDER 5);
CREATE TABLE clientes_2 PARTITION OF clientes
```

New subfeature: Re-hashing (2)

```
CREATE TABLE clientes_00 (LIKE clientes);
CREATE TABLE clientes_01 (LIKE clientes);

WITH moved AS (
  DELETE FROM clientes_0
    WHERE satisfies_hash_partition('clientes'::regclass, 6, 0,
                                   cliente_id)
    RETURNING *)
INSERT INTO clientes_00 SELECT * FROM moved;

WITH moved AS (
  DELETE FROM clientes_0
    WHERE satisfies_hash_partition('clientes'::regclass, 6, 3,
                                   cliente_id)
    RETURNING *)
INSERT INTO clientes_01 SELECT * FROM moved;
```

New subfeature: Re-hashing (2)

```
ALTER TABLE clientes DETACH PARTITION clientes_0;
ALTER TABLE clientes ATTACH PARTITION clientes_00
  FOR VALUES WITH (MODULUS 6, REMAINDER 0);
ALTER TABLE clientes ATTACH PARTITION clientes_01
  FOR VALUES WITH (MODULUS 6, REMAINDER 3);
```

New feature: ON CONFLICT DO UPDATE

```
CREATE TABLE order_items (
  order_id INTEGER NOT NULL,
  item_id INTEGER NOT NULL,
  quantity INTEGER NOT NULL CHECK (quantity > 0),
  UNIQUE (order_id, item_id)
) PARTITION BY HASH (order_id);
-- create partitions

INSERT INTO order_items VALUES (888, 12345, 5)
  ON CONFLICT (order_id, item_id) DO UPDATE
  SET quantity = order_items.quantity + EXCLUDED.quantity;
```

Better DDL support

- CREATE INDEX
- UNIQUE & PRIMARY KEY constraints
- FOREIGN KEY constraints
- Row-level triggers

Better DDL: CREATE INDEX

- `CREATE INDEX` applies to parent table
- Cascades to each partition
  - If identical index already exists, it is attached
  - If not, a new index is created
- Clones the index when new partitions are added
  - or attaches an existing index

Better DDL: CREATE INDEX

- `CREATE INDEX` applies to parent table
- Cascades to each partition
  - If identical index already exists, it is attached
  - If not, a new index is created
- Clones the index when new partitions are added
  - or attaches an existing index
- Index can be created ON ONLY parent table
  - No cascading occurs
  - Partition indexes can be attached later
    - `ALTER INDEX ATTACH PARTITION`
  - Once all partition indexes are attached, parent index becomes valid

Better DDL: CREATE INDEX

- `CREATE INDEX` applies to parent table
- Cascades to each partition
  - If identical index already exists, it is attached
  - If not, a new index is created
- Clones the index when new partitions are added
  - or attaches an existing index
- Index can be created ON ONLY parent table
  - No cascading occurs
  - Partition indexes can be attached later
    - `ALTER INDEX ATTACH PARTITION`
  - Once all partition indexes are attached, parent index becomes valid
  - This is what `pg_dump` does

Better DDL: UNIQUE constraints

- UNIQUE constraints are just indexes that are UNIQUE
- ... well, add a `pg_constraint` row
  - So we clone that too

Better DDL: UNIQUE constraints

- UNIQUE constraints are just indexes that are UNIQUE
- ... well, add a `pg_constraint` row
  - So we clone that too
- Limitation: all columns in partition key must appear in constraint
- Local unicity ensures global unicity
- To do better requires global indexes or other tricks

Better DDL: FOREIGN KEY constraints

- FKs in partitioned tables referencing non-partitioned tables
- Doing the other way around requires more effort :-(
- New partitions clone the constraints/trigger
- User doesn't need to do anything

Better DDL: Row-level triggers

- `AFTER` triggers `FOR EACH ROW` on partitioned table
- Cloned to each partition on creation

Performance: Faster pruning

- Constraint exclusion is slow and limited
- Partition pruning is completely new, more advanced tech
- It produces a “pruning program“ from query WHERE clause and partition bounds
- Initially, pruning applies at plan time
  - just like constraint exclusion

Pruning example

```
EXPLAIN (ANALYZE, COSTS off)
   SELECT * FROM clientes
   WHERE cliente_id = 1234;
                              QUERY PLAN
-----------------------------------------------------------------------
 Append (actual time=0.054..2.787 rows=1 loops=1)
   -> Seq Scan on clientes_2 (actual time=0.052..2.785 rows=1 loops=1)
         Filter: (cliente_id = 1234)
         Rows Removed by Filter: 12570
 Planning Time: 0.292 ms
 Execution Time: 2.822 ms
(6 filas)
```

No pruning example

```
SET enable_partition_pruning TO off;
EXPLAIN (ANALYZE, COSTS off)
   SELECT * FROM clientes
   WHERE cliente_id = 1234;
                                QUERY PLAN
-------------------------------------------------------------------------
 Append (actual time=6.658..10.549 rows=1 loops=1)
   -> Seq Scan on clientes_1 (actual time=4.724..4.724 rows=0 loops=1)
         Filter: (cliente_id = 1234)
         Rows Removed by Filter: 24978
   -> Seq Scan on clientes_00 (actual time=1.914..1.914 rows=0 loops=1)
         Filter: (cliente_id = 1234)
         Rows Removed by Filter: 12644
   -> Seq Scan on clientes_2 (actual time=0.017..1.021 rows=1 loops=1)
         Filter: (cliente_id = 1234)
         Rows Removed by Filter: 12570
   -> Seq Scan on clientes_3 (actual time=0.746..0.746 rows=0 loops=1)
         Filter: (cliente_id = 1234)
         Rows Removed by Filter: 12448
   -> Seq Scan on clientes_01 (actual time=0.648..0.648 rows=0 loops=1)
```

Performance: Runtime pruning

- Partition pruning can be applied at execution time too
- Many queries can be optimized better at “run” time
- Two chances for runtime pruning
  - When bound parameters are given values (bind time)
  - Values obtained from other execution nodes

Runtime pruning example

```
explain (analyze, costs off, summary off, timing off)
   execute ab_q1 (2, 2, 3);
                      QUERY PLAN
-----------------------------------------------------
 Append (actual rows=0 loops=1)
   Subplans Removed: 6
   -> Seq Scan on ab_a2_b1 (actual rows=0 loops=1)
         Filter: ((a >= $1) AND (a <= $2) AND (b <= $3))
   -> Seq Scan on ab_a2_b2 (actual rows=0 loops=1)
         Filter: ((a >= $1) AND (a <= $2) AND (b <= $3))
   -> Seq Scan on ab_a2_b3 (actual rows=0 loops=1)
         Filter: ((a >= $1) AND (a <= $2) AND (b <= $3))
(8 rows)
```

Another runtime pruning example

```
explain (analyze, costs off, summary off, timing off)
select * from tbl1 join tprt on tbl1.col1 < tprt.col1;
                              QUERY PLAN
-----------------------------------------------------------------------
 Nested Loop (actual rows=1 loops=1)
   -> Seq Scan on tbl1 (actual rows=1 loops=1)
   -> Append (actual rows=1 loops=1)
         -> Index Scan using tprt1_idx on tprt_1 (never executed)
               Index Cond: (tbl1.col1 < col1)
         -> Index Scan using tprt2_idx on tprt_2 (never executed)
               Index Cond: (tbl1.col1 < col1)
         -> Index Scan using tprt5_idx on tprt_5 (never executed)
               Index Cond: (tbl1.col1 < col1)
         -> Index Scan using tprt6_idx on tprt_6 (actual rows=1 loops=1)
               Index Cond: (tbl1.col1 < col1)
(15 rows)
```

Performance: Partitionwise joins

- Applies to joins between partitioned tables
- Normal case: join produces cartesian product of partitions
- Partitionwise join: join occurs “per partition”
  - If partition bounds are identical
  - only joins those partitions with matching bounds

Partitionwise join example

```
CREATE TABLE orders (order_id int, client_id int)
  PARTITION BY RANGE (order_id);
CREATE TABLE orders_1000 PARTITION OF orders
  for values FROM (1) TO (1000);
CREATE TABLE orders_2000 PARTITION OF orders
  FOR VALUES FROM (1000) TO (2000);

CREATE TABLE order_items (order_id int, item_id int)
  PARTITION BY RANGE (order_id);
CREATE TABLE order_items_1000 PARTITION OF order_items
  for VALUES FROM (1) TO (1000);
CREATE TABLE order_items_2000 PARTITION OF order_items
  FOR VALUES FROM (1000) TO (2000);
```

Partitionwise join example

```
SET enable_partitionwise_join TO off;
EXPLAIN (COSTS OFF) SELECT * FROM orders JOIN order_items
USING (order_id) WHERE customer_id = 64;
                                QUERY PLAN
-------------------------------------------------------------------------
 Hash Join
   Hash Cond: (order_items_1000.order_id = orders_1000.order_id)
   -> Append
         -> Seq Scan on order_items_1000
         -> Seq Scan on order_items_2000
   -> Hash
         -> Append
               -> Bitmap Heap Scan on orders_1000
                     Recheck Cond: (customer_id = 64)
                     -> Bitmap Index Scan on orders_1000_customer_id_idx
                           Index Cond: (customer_id = 64)
               -> Seq Scan on orders_2000
                     Filter: (customer_id = 64)
(13 filas)
```

Partitionwise join example

```
EXPLAIN (COSTS OFF) SELECT * FROM orders JOIN order_items
USING (order_id) WHERE customer_id = 64;
                                  QUERY PLAN
-----------------------------------------------------------------------------
 Append
   -> Hash Join
         Hash Cond: (order_items_1000.order_id = orders_1000.order_id)
         -> Seq Scan on order_items_1000
         -> Hash
               -> Bitmap Heap Scan on orders_1000
                     Recheck Cond: (customer_id = 64)
                     -> Bitmap Index Scan on orders_1000_customer_id_idx
                           Index Cond: (customer_id = 64)
   -> Nested Loop
         -> Seq Scan on orders_2000
               Filter: (customer_id = 64)
         -> Index Scan using order_items_2000_order_id_idx on order_item
               Index Cond: (order_id = orders_2000.order_id)
```

Thanks!

Questions?

2ndQuadrant PostgreSQL