

# PostgreSQL: La Ruta del Particionamiento

2ndQuadrant – Professional PostgreSQL

Valdivia, Chile, Agosto 2020

## Particionamiento declarativo



## Particionamiento declarativo

- ▶ Introducido en PostgreSQL 10 (2017)
- ▶ Mejoras significativas con cada nueva versión
- ▶ Más práctico de manejar para usuarios que el sistema antiguo
- ▶ Mejor rendimiento que el sistema antiguo en muchas áreas
- ▶ ... pero aún nos faltan cosas importantes

## Particionamiento declarativo

- ▶ Introducido en PostgreSQL 10 (2017)
- ▶ Mejoras significativas con cada nueva versión
- ▶ Más práctico de manejar para usuarios que el sistema antiguo
- ▶ Mejor rendimiento que el sistema antiguo en muchas áreas
- ▶ ... pero aún nos faltan cosas importantes

# DDL básico de particionamiento

```
CREATE TABLE pedidos (  
    pedido_id BIGINT,  
    pedido_fecha TIMESTAMP WITH TIME ZONE, ...  
) PARTITION BY RANGE (pedido_fecha);
```

# Estrategias de particionamiento

RANGE Rango de valores para cada columna de particionamiento

LIST Lista de valores para cada columna de particionamiento

HASH *Hash* de los valores de las columnas de particionamiento

▶ Añadido en PostgreSQL 11

# Estrategias de particionamiento

RANGE Rango de valores para cada columna de particionamiento

LIST Lista de valores para cada columna de particionamiento

HASH *Hash* de los valores de las columnas de particionamiento

▶ Añadido en PostgreSQL 11

## Estrategias de particionamiento: RANGE

```
-- crea partición vacía  
CREATE TABLE pedidos_2018_01  
    PARTITION OF pedidos FOR VALUES FROM ('2018-01-01') TO ('2018-02-01');
```

```
-- agrega como partición una tabla existente que puede tener datos  
ALTER TABLE pedidos ATTACH PARTITION pedidos_2018_02  
FOR VALUES FROM ('2018-02-01') TO ('2018-03-01');
```

- ▶ Intervalo cerrado a la izquierda, abierto a la derecha
- ▶ ¡No se necesita trigger/regla para rutear tuplas!
  - ▶ gran ventaja comparado con sistema antiguo



## Estrategias de particionamiento: RANGE

```
-- crea partición vacía
CREATE TABLE pedidos_2018_01
  PARTITION OF pedidos FOR VALUES FROM ('2018-01-01') TO ('2018-02-01');

-- agrega como partición una tabla existente que puede tener datos
ALTER TABLE pedidos ATTACH PARTITION pedidos_2018_02
FOR VALUES FROM ('2018-02-01') TO ('2018-03-01');
```

- ▶ Intervalo cerrado a la izquierda, abierto a la derecha
- ▶ ¡No se necesita trigger/regla para rutear tuplas!
  - ▶ gran ventaja comparado con sistema antiguo

## Estrategias de particionamiento: RANGE

```
-- crea partición vacía
CREATE TABLE pedidos_2018_01
  PARTITION OF pedidos FOR VALUES FROM ('2018-01-01') TO ('2018-02-01');

-- agrega como partición una tabla existente que puede tener datos
ALTER TABLE pedidos ATTACH PARTITION pedidos_2018_02
FOR VALUES FROM ('2018-02-01') TO ('2018-03-01');
```

- ▶ Intervalo cerrado a la izquierda, abierto a la derecha
- ▶ ¡No se necesita trigger/regla para rutear tuplas!
  - ▶ gran ventaja comparado con sistema antiguo

## Estrategias de particionamiento: RANGE

```
-- crea partición vacía  
CREATE TABLE pedidos_2018_01  
    PARTITION OF pedidos FOR VALUES FROM ('2018-01-01') TO ('2018-02-01');
```

```
-- agrega como partición una tabla existente que puede tener datos  
ALTER TABLE pedidos ATTACH PARTITION pedidos_2018_02  
FOR VALUES FROM ('2018-02-01') TO ('2018-03-01');
```

- ▶ Intervalo cerrado a la izquierda, abierto a la derecha
- ▶ ¡No se necesita trigger/regla para rutear tuplas!
  - ▶ gran ventaja comparado con sistema antiguo

## Estrategias de particionamiento: RANGE (2)

```
CREATE TABLE lecturas_sensores (sensor_id BIGINT, momento TIMESTAMPTZ,  
    lectura DOUBLE PRECISION)  
    PARTITION BY RANGE (momento, sensor_id);
```

```
CREATE TABLE lecturas_sensores_2016 PARTITION OF lecturas_sensores  
    FOR VALUES FROM ('2016-01-01', MINVALUE) TO ('2017-01-01', MINVALUE);
```

```
CREATE TABLE lecturas_sensores_2017_1 PARTITION OF lecturas_sensores  
    FOR VALUES FROM ('2017-01-01', MINVALUE) TO ('2017-01-02', 10000);
```

```
CREATE TABLE lecturas_sensores_2017_2 PARTITION OF lecturas_sensores  
    FOR VALUES FROM ('2017-07-02', MINVALUE) TO ('2018-01-01', MINVALUE);
```

## Estrategias de particionamiento: RANGE (2)

```
CREATE TABLE lecturas_sensores (sensor_id BIGINT, momento TIMESTAMPTZ,  
    lectura DOUBLE PRECISION)  
    PARTITION BY RANGE (momento, sensor_id);
```

```
CREATE TABLE lecturas_sensores_2016 PARTITION OF lecturas_sensores  
    FOR VALUES FROM ('2016-01-01', MINVALUE) TO ('2017-01-01', MINVALUE);
```

```
CREATE TABLE lecturas_sensores_2017_1 PARTITION OF lecturas_sensores  
    FOR VALUES FROM ('2017-01-01', MINVALUE) TO ('2017-01-02', 10000);
```

```
CREATE TABLE lecturas_sensores_2017_2 PARTITION OF lecturas_sensores  
    FOR VALUES FROM ('2017-07-02', MINVALUE) TO ('2018-01-01', MINVALUE);
```

## Estrategias de particionamiento: RANGE (2)

```
CREATE TABLE lecturas_sensores (sensor_id BIGINT, momento TIMESTAMPTZ,  
    lectura DOUBLE PRECISION)  
    PARTITION BY RANGE (momento, sensor_id);
```

```
CREATE TABLE lecturas_sensores_2016 PARTITION OF lecturas_sensores  
    FOR VALUES FROM ('2016-01-01', MINVALUE) TO ('2017-01-01', MINVALUE);
```

```
CREATE TABLE lecturas_sensores_2017_1 PARTITION OF lecturas_sensores  
    FOR VALUES FROM ('2017-01-01', MINVALUE) TO ('2017-01-02', 10000);
```

```
CREATE TABLE lecturas_sensores_2017_2 PARTITION OF lecturas_sensores  
    FOR VALUES FROM ('2017-07-02', MINVALUE) TO ('2018-01-01', MINVALUE);
```

## Estrategias de particionamiento: RANGE (2)

```
CREATE TABLE lecturas_sensores (sensor_id BIGINT, momento TIMESTAMPTZ,  
    lectura DOUBLE PRECISION)  
    PARTITION BY RANGE (momento, sensor_id);
```

```
CREATE TABLE lecturas_sensores_2016 PARTITION OF lecturas_sensores  
    FOR VALUES FROM ('2016-01-01', MINVALUE) TO ('2017-01-01', MINVALUE);
```

```
CREATE TABLE lecturas_sensores_2017_1 PARTITION OF lecturas_sensores  
    FOR VALUES FROM ('2017-01-01', MINVALUE) TO ('2017-01-02', 10000);
```

```
CREATE TABLE lecturas_sensores_2017_2 PARTITION OF lecturas_sensores  
    FOR VALUES FROM ('2017-07-02', MINVALUE) TO ('2018-01-01', MINVALUE);
```

## Estrategias de particionamiento: LIST

```
CREATE TYPE paises AS ENUM ('Argentina', 'Bolivia', 'Brasil', 'Chile', 'Colombia',  
    'Costa Rica', 'Cuba', 'Ecuador', 'El Salvador', 'Guatemala', 'Haití', 'Honduras',  
    'México', 'Nicaragua', 'Panamá', 'Paraguay', 'Perú', 'República Dominicana',  
    'Uruguay', 'Venezuela');
```

```
CREATE TABLE clientes (cliente_id INTEGER, pais PAISES, ...)  
    PARTITION BY LIST (pais);
```

```
CREATE TABLE clientes_co PARTITION OF clientes  
    FOR VALUES IN ('Colombia');
```

```
CREATE TABLE clientes_ar_cl PARTITION OF clientes  
    FOR VALUES IN ('Argentina', 'Chile');
```



## Estrategias de particionamiento: LIST

```
CREATE TYPE paises AS ENUM ('Argentina', 'Bolivia', 'Brasil', 'Chile', 'Colombia',  
    'Costa Rica', 'Cuba', 'Ecuador', 'El Salvador', 'Guatemala', 'Haití', 'Honduras',  
    'México', 'Nicaragua', 'Panamá', 'Paraguay', 'Perú', 'República Dominicana',  
    'Uruguay', 'Venezuela');
```

```
CREATE TABLE clientes (cliente_id INTEGER, pais PAISES, ...)  
    PARTITION BY LIST (pais);
```

```
CREATE TABLE clientes_co PARTITION OF clientes  
    FOR VALUES IN ('Colombia');
```

```
CREATE TABLE clientes_ar_cl PARTITION OF clientes  
    FOR VALUES IN ('Argentina', 'Chile');
```

# Niveles de lock

Sobre la tabla particionada:

- ▶ `CREATE TABLE .. PARTITION OF`
  - ▶ `AccessExclusiveLock`
- ▶ `ALTER TABLE .. ATTACH PARTITION`
  - ▶ `AccessExclusiveLock` (PostgreSQL 10, 11)
  - ▶ `ShareUpdateExclusiveLock` (PostgreSQL 12)
- ▶ `ALTER TABLE .. DETACH PARTITION`
  - ▶ `AccessExclusiveLock`
  - ▶ `DETACH PARTITION CONCURRENTLY`
    - ▶ `ShareUpdateExclusive` (PostgreSQL 14?)

# Niveles de lock

Sobre la tabla particionada:

- ▶ `CREATE TABLE .. PARTITION OF`
  - ▶ `AccessExclusiveLock`
- ▶ `ALTER TABLE .. ATTACH PARTITION`
  - ▶ `AccessExclusiveLock` (PostgreSQL 10, 11)
  - ▶ `ShareUpdateExclusiveLock` (PostgreSQL 12)
- ▶ `ALTER TABLE .. DETACH PARTITION`
  - ▶ `AccessExclusiveLock`
  - ▶ `DETACH PARTITION CONCURRENTLY`
    - ▶ `ShareUpdateExclusive` (PostgreSQL 14?)

## Niveles de lock

Sobre la tabla particionada:

- ▶ `CREATE TABLE .. PARTITION OF`
  - ▶ `AccessExclusiveLock`
- ▶ `ALTER TABLE .. ATTACH PARTITION`
  - ▶ `AccessExclusiveLock` (PostgreSQL 10, 11)
  - ▶ `ShareUpdateExclusiveLock` (PostgreSQL 12)
- ▶ `ALTER TABLE .. DETACH PARTITION`
  - ▶ `AccessExclusiveLock`
  - ▶ `DETACH PARTITION CONCURRENTLY`
    - ▶ `ShareUpdateExclusive` (PostgreSQL 14?)

## Estrategias de particionamiento: HASH

```
CREATE TABLE productos (producto_id INTEGER, nombre TEXT, descripcion TEXT)  
    PARTITION BY HASH (producto_id);
```

```
CREATE TABLE productos_0 PARTITION OF productos FOR VALUES WITH (MODULUS 4, REMAINDER 0);  
CREATE TABLE productos_1 PARTITION OF productos FOR VALUES WITH (MODULUS 4, REMAINDER 1);  
CREATE TABLE productos_2 PARTITION OF productos FOR VALUES WITH (MODULUS 4, REMAINDER 2);
```

```
CREATE TABLE productos_7 PARTITION OF productos FOR VALUES WITH (MODULUS 8, REMAINDER 7);  
CREATE TABLE productos_3 PARTITION OF productos FOR VALUES WITH (MODULUS 8, REMAINDER 3);
```

## Estrategias de particionamiento: HASH

```
CREATE TABLE productos (producto_id INTEGER, nombre TEXT, descripcion TEXT)  
    PARTITION BY HASH (producto_id);
```

```
CREATE TABLE productos_0 PARTITION OF productos FOR VALUES WITH (MODULUS 4, REMAINDER 0);  
CREATE TABLE productos_1 PARTITION OF productos FOR VALUES WITH (MODULUS 4, REMAINDER 1);  
CREATE TABLE productos_2 PARTITION OF productos FOR VALUES WITH (MODULUS 4, REMAINDER 2);
```

```
CREATE TABLE productos_7 PARTITION OF productos FOR VALUES WITH (MODULUS 8, REMAINDER 7);  
CREATE TABLE productos_3 PARTITION OF productos FOR VALUES WITH (MODULUS 8, REMAINDER 3);
```

## Estrategias de particionamiento: HASH

```
CREATE TABLE productos (producto_id INTEGER, nombre TEXT, descripcion TEXT)  
    PARTITION BY HASH (producto_id);
```

```
CREATE TABLE productos_0 PARTITION OF productos FOR VALUES WITH (MODULUS 4, REMAINDER 0);  
CREATE TABLE productos_1 PARTITION OF productos FOR VALUES WITH (MODULUS 4, REMAINDER 1);  
CREATE TABLE productos_2 PARTITION OF productos FOR VALUES WITH (MODULUS 4, REMAINDER 2);
```

```
CREATE TABLE productos_7 PARTITION OF productos FOR VALUES WITH (MODULUS 8, REMAINDER 7);  
CREATE TABLE productos_3 PARTITION OF productos FOR VALUES WITH (MODULUS 8, REMAINDER 3);
```

## Características adicionales

- ▶ `ON CONFLICT DO UPDATE` (PostgreSQL 11)
- ▶ `UPDATE` entre particiones (“migración” de tuplas) (PostgreSQL 11)
- ▶ Triggers `FOR EACH STATEMENT`
- ▶ Triggers `FOR EACH ROW, AFTER` (PostgreSQL 11)
- ▶ Triggers `FOR EACH ROW, BEFORE` (PostgreSQL 12)



## Migración de tuplas

```
=# select tableoid::regclass, * from clientes where cliente_id = 1234;
```

tableoid	cliente_id	pais
clientes_pe_01	1234	Perú

```
=# update clientes set pais = 'Argentina' where cliente_id = 1234;
```

```
=# select tableoid::regclass, * from clientes where cliente_id = 1234;
```

tableoid	cliente_id	pais
clientes_ar	1234	Argentina

## Migración de tuplas

```
=# select tableoid::regclass, * from clientes where cliente_id = 1234;
```

tableoid	cliente_id	pais
clientes_pe_01	1234	Perú

```
=# update clientes set pais = 'Argentina' where cliente_id = 1234;
```

```
=# select tableoid::regclass, * from clientes where cliente_id = 1234;
```

tableoid	cliente_id	pais
clientes_ar	1234	Argentina

# Índices en tablas particionadas

- ▶ `CREATE INDEX ON pedidos (pedido_id)`
  - ▶ (Desde PostgreSQL 11)
- ▶ Particiones existentes adquieren el índice automáticamente
  - ▶ pero requiere *ShareLock*
  - ▶ es decir, bloquea INSERTs etc
- ▶ Particiones futuras obtendrán el índice
- ▶ Después de `ATTACH`, el índice debe existir
  - ▶ Si no existe, se crea uno nuevo
    - ▶ lo cual requiere bloqueo
  - ▶ mejor: crearlo de antemano
    - ▶ pasa a ser parte de la jerarquía

# Índices en tablas particionadas

- ▶ `CREATE INDEX ON pedidos (pedido_id)`
  - ▶ (Desde PostgreSQL 11)
- ▶ Particiones existentes adquieren el índice automáticamente
  - ▶ pero requiere *ShareLock*
  - ▶ es decir, bloquea INSERTs etc
- ▶ Particiones futuras obtendrán el índice
- ▶ Después de ATTACH, el índice debe existir
  - ▶ Si no existe, se crea uno nuevo
    - ▶ lo cual requiere bloqueo
  - ▶ mejor: crearlo de antemano
    - ▶ pasa a ser parte de la jerarquía

## Índices en tablas particionadas (2)

Crear índices en masa:

1. Crear índices individuales en modo concurrente en las particiones “hoja”

```
SELECT pg_catalog.format('CREATE INDEX CONCURRENTLY ON %s (pedido_id);',  
                          p.relid::pg_catalog.regclass)  
FROM pg_catalog.pg_partition_tree('pedidos') p  
WHERE isleaf \gexec
```

2. Crear índices en “padre”, *adquiere* índices en particiones

```
CREATE INDEX ON pedidos (pedido_id);
```

## Funciones de introspección: pg\_partition\_root

### ► PostgreSQL 12

```
=# select pg_partition_root('clientes_ar_cl'::regclass);
```

pg_partition_root
clientes

## Funciones de introspección: pg\_partition\_ancestors

### ► PostgreSQL 12

```
=# select pg_partition_ancestors('clientes_ar_cl'::regclass);
```

pg_partition_ancestors
clientes_ar_cl
clientes

## Funciones de introspección: pg\_partition\_tree

### ► PostgreSQL 12

```
=# SELECT * FROM pg_partition_tree('clientes_ar_cl'::regclass);
```

relid	parentrelid	isleaf	level
clientes		f	0
clientes_ar_cl	clientes	t	1
clientes_co	clientes	t	1



## Particionamiento multi-nivel

- ▶ Particiones pueden particionarse
- ▶ ... con estrategias diferentes

```
CREATE TABLE clientes (cliente_id INTEGER, pais PAISES, ...)
    PARTITION BY LIST (pais);
```

```
CREATE TABLE clientes_pe PARTITION OF clientes FOR VALUES IN ('Perú')
    PARTITION BY RANGE (cliente_id);
```

```
CREATE TABLE clientes_pe_01 PARTITION OF clientes_pe
    FOR VALUES FROM (1) TO (10000);
```

```
CREATE TABLE clientes_pe_02 PARTITION OF clientes_pe
    FOR VALUES FROM (10000) TO (20000);
```

## Particionamiento multi-nivel

- ▶ Particiones pueden particionarse
- ▶ ... con estrategias diferentes

```
CREATE TABLE clientes (cliente_id INTEGER, pais PAISES, ...)
    PARTITION BY LIST (pais);
```

```
CREATE TABLE clientes_pe PARTITION OF clientes FOR VALUES IN ('Perú')
    PARTITION BY RANGE (cliente_id);
```

```
CREATE TABLE clientes_pe_01 PARTITION OF clientes_pe
    FOR VALUES FROM (1) TO (10000);
```

```
CREATE TABLE clientes_pe_02 PARTITION OF clientes_pe
    FOR VALUES FROM (10000) TO (20000);
```

## Particionamiento multi-nivel (2)

```
=# select * from pg_partition_tree('clientes');
```

relid	parentrelid	isleaf	level
clientes		f	0
clientes_pe	clientes	f	1
clientes_ar	clientes	t	1
clientes_pe_01	clientes_pe	t	2
clientes_pe_02	clientes_pe	t	2

(5 filas)

psql

```
=# \dP
```

Listado de relaciones particionadas

Esquema	Nombre	Dueño	Tipo	Tabla
public	clientes	alvherre	tabla particionada	pedidos
public	pedidos	alvherre	tabla particionada	
public	pedidosidx	alvherre	índice particionado	

(3 filas)

## psql (2)

```
=# \d+ clientes
```

Tabla particionada «public.clientes»

Columna	Tipo	Ordenamiento	Nulable	Por omisión	Almacenamiento
cliente_id	integer				plain
pais	paises				plain

Llave de partición: LIST (pais)

Particiones: clientes\_ar\_cl FOR VALUES IN ('Argentina', 'Chile'),  
clientes\_pe FOR VALUES IN ('Perú'), PARTITIONED

## Partición DEFAULT

- ▶ PostgreSQL 11
- ▶ Partición que recibe valores no válidos en otras particiones
- ▶ Particionado RANGE: recibe valores NULL
- ▶ Importantes dolores de cabeza
  - ▶ Nivel de lock es siempre AccessExclusiveLock
  - ▶ Migrar registros requiere bloqueo

## Partición DEFAULT

- ▶ PostgreSQL 11
- ▶ Partición que recibe valores no válidos en otras particiones
- ▶ Particionado RANGE: recibe valores NULL
- ▶ Importantes dolores de cabeza
  - ▶ Nivel de lock es siempre AccessExclusiveLock
  - ▶ Migrar registros requiere bloqueo

## Características adicionales (2)

- ▶ Índices, llaves primarias (PostgreSQL 11)
- ▶ Llaves foráneas “desde” (PostgreSQL 11)
- ▶ Llaves foráneas “hacia” (PostgreSQL 12)



# Índices únicos

- ▶ Los índices únicos requieren incluir la llave de particionamiento
- ▶ Corolario 1: las llaves primarias deben incluir llave de particionamiento
- ▶ Corolario 2: muchas tablas no son buenas candidatos a particionar
- ▶ Peor aún en particionamiento multi-nivel

# Índices únicos

- ▶ Los índices únicos requieren incluir la llave de particionamiento
- ▶ Corolario 1: las llaves primarias deben incluir llave de particionamiento
- ▶ Corolario 2: muchas tablas no son buenas candidatos a particionar
- ▶ Peor aún en particionamiento multi-nivel

# Índices únicos

- ▶ Los índices únicos requieren incluir la llave de particionamiento
- ▶ Corolario 1: las llaves primarias deben incluir llave de particionamiento
- ▶ Corolario 2: muchas tablas no son buenas candidatos a particionar
- ▶ Peor aún en particionamiento multi-nivel

## Índices únicos

- ▶ Los índices únicos requieren incluir la llave de particionamiento
- ▶ Corolario 1: las llaves primarias deben incluir llave de particionamiento
- ▶ Corolario 2: muchas tablas no son buenas candidatos a particionar
- ▶ Peor aún en particionamiento multi-nivel

## Poda de particiones

- ▶ “Podar” significa excluir particiones durante consultas
- ▶ Procesar menos particiones es mejorar el rendimiento
- ▶ Ejemplo de poda en tiempo de optimización

```
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)
```

## Poda de particiones

```
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
```

## *Poda* en tiempo de ejecución

- ▶ Muchas consultas se pueden podar además en tiempo de ejecución
- ▶ Dos momentos para podar en tiempo de ejecución
  - ▶ Cuando variables reciben valores “bind”
  - ▶ Cuando parámetros reciben valores desde otros nodos de ejecución

## Ejemplo de poda en tiempo de ejecución (1)

```
EXPLAIN (analyze, costs off, summary off, timing off)
  EXECUTE buscar_clientes(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)
```



## Ejemplo de poda en tiempo de ejecución (2)

```
EXPLAIN (analyze, ...) 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)
```

is better)

2000

1500



is better)

15000

Prim



Run-time

(better)

12500

10000



Plan-time

better)

10000

7500



## Mantenimiento (vacuum, analyze)

- ▶ Mayoría de operaciones de mantenimiento funcionan por partición
- ▶ Autovacuum se hace cargo en forma normal
- ▶ Única excepción: ANALYZE a la jerarquía completa
- ▶ En proceso para PostgreSQL 14

# Replicación Lógica

```
CREATE PUBLICATION pub_clientes FOR TABLE clientes;  
  
ALTER PUBLICATION pub_productos ADD TABLE productos;
```

## Particionamiento: futuro cercano

- ▶ REINDEX CONCURRENTLY
- ▶ CREATE INDEX CONCURRENTLY
- ▶ ALTER TABLE .. DETACH CONCURRENTLY
- ▶ Autovacuum: autoanalyze
- ▶ ModifyTable (UPDATE, DELETE)



## Particionamiento: futuro cercano

- ▶ REINDEX CONCURRENTLY
- ▶ CREATE INDEX CONCURRENTLY
- ▶ ALTER TABLE .. DETACH CONCURRENTLY
- ▶ Autovacuum: autoanalyze
- ▶ ModifyTable (UPDATE, DELETE)

## Particionamiento: futuro cercano

- ▶ REINDEX CONCURRENTLY
- ▶ CREATE INDEX CONCURRENTLY
- ▶ ALTER TABLE .. DETACH CONCURRENTLY
- ▶ Autovacuum: autoanalyze
- ▶ ModifyTable (UPDATE, DELETE)

## Particionamiento: futuro lejano

- ▶ Índices globales
- ▶ Append/MergeAppend asíncrono (FDW)
- ▶ Mejoras para particiones tablas foráneas

¿Preguntas?

Agradezco su atención

- ▶ `pgsql-es-ayuda@lists.postgresql.org`
- ▶ `info@2ndQuadrant.com`

## Ejemplo para JOINS a nivel de partición

```
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);
```

## Ejemplo sin JOINS a nivel de partición

### ► PostgreSQL 11

```
SET enable_partitionwise_join TO off;  
EXPLAIN (COSTS OFF)  
  SELECT * FROM orders JOIN order_items USING (order_id)  
  WHERE customer_id = 64;
```

## Sin JOIN a nivel de partición

### 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)

# Activando JOIN a nivel de partición

## 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\_items\_2000

Index Cond: (order\_id = orders\_2000.order\_id)



## JOINS a nivel de partición: avanzado

- ▶ PostgreSQL 13
- ▶ Las estrategias de partición ya no requieren ser idénticas

## Rehashing: inicial

```
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_5 PARTITION OF clientes FOR VALUES WITH (MODULUS 6, REMAINDER 5);
```

## Rehashing: migración

```
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;
```