

Dissecting Partitioning

BRUCE MOMJIAN



This presentation shows the purpose, effect, and optimizations of Postgres's partitioning implementation.

<https://momjian.us/presentations>



Creative Commons Attribution License

Last updated: February 2024

Outline

1. Purpose and history
2. Partitioning in action
3. Partition pruning
4. Partition-local optimizations
5. Time-based partitioning
6. Row migration
7. psql support
8. Limitations
9. Complex architectures

1. Purpose and History



https://www.flickr.com/photos/photo_oto/

Purpose of Partitioning

- Query efficiency
 - faster sequential scans for partition-key-qualified queries
 - shallower indexes for faster index scans
 - improve cache efficiency for frequently accessed partitions
 - partition-local optimizations
- Data customization
 - partition-specific indexes, constraints, and triggers
- Maintenance
 - improve efficiency of modified row cleanup
 - movement of partitions to tablespaces with different storage characteristics
 - easily attach/detach partitions for maintenance
 - simpler bulk operations, including removal of old time-based partitions

<https://www.postgresql.org/docs/current/ddl-partitioning.html>

<https://hevodata.com/learn/postgresql-partitions/>

<https://www.youtube.com/watch?v=edQZauVU-ws>

Partitioning Prior to Postgres 10

Prior to Postgres 10, partitioning was accomplished by combining:

- Inheritance
- CHECK constraints and *constraint_exclusion*
- Triggers for INSERT routing of new rows
- Triggers for UPDATES that change partitioned key columns
- Use of external tools like pg_partman (https://github.com/pgpartman/pg_partman)

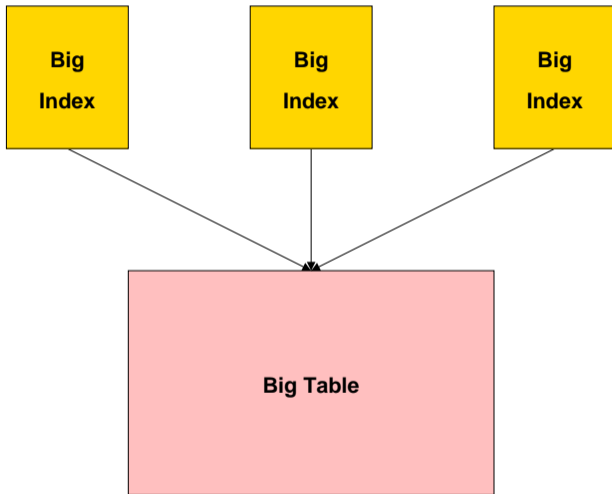
Enhancing Partitioning Capabilities

Postgres 10 and later added:

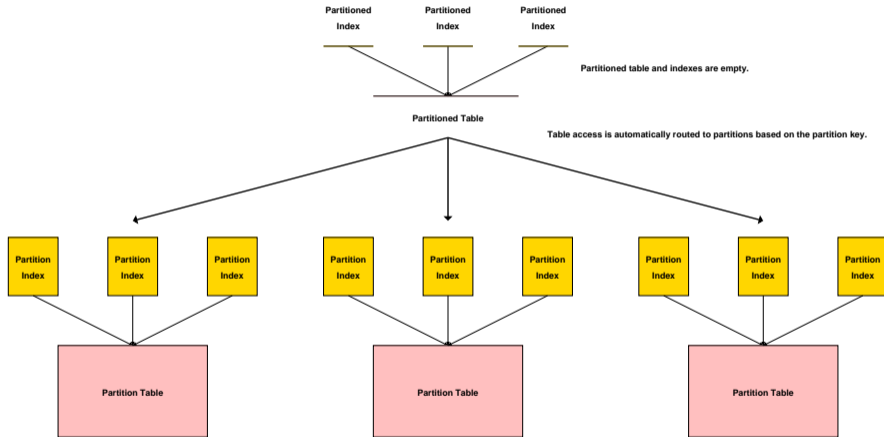
- Range, hash, and list partition syntax
- DEFAULT partitions*
- PRIMARY KEYS on partitioned* tables and FOREIGN KEYS to and from partitioned tables
- More efficient pruning
- Executor-stage pruning
- Partition-local optimizations
- Used with foreign data wrappers

* “Partitioned tables” are parent tables that are referenced in most queries and store no data, while “partitions” are tables where the data is stored.

Without Partitioning



With Partitioning



2. Partitioning in Action



<https://www.flickr.com/photos/105105658@N03/>

Range Partitioned Table Creation

```
CREATE TABLE range_partitioned (name TEXT)  
PARTITION BY RANGE (name);
```

```
CREATE TABLE range_partition_less_j  
PARTITION OF range_partitioned  
FOR VALUES FROM (MINVALUE) TO ('j');
```

```
CREATE TABLE range_partition_j_to_s  
PARTITION OF range_partitioned  
FOR VALUES FROM ('j') TO ('s');
```

```
CREATE TABLE range_partition_s_greater  
PARTITION OF range_partitioned  
FOR VALUES FROM ('s') TO (MAXVALUE);
```

Summary and causal information is in **blue**; significant information is in **red**. All the queries used in this presentation are available at <https://momjian.us/main/writings/pgsql/partitioning.sql>.

DEFAULT Partition Table Creation

*-- Range partitioned tables require DEFAULT partitions for NULL storage.
-- CHECK prevents non-NULLs and avoids partition scan checking for
-- values that might be in the newly-created partition.*

```
CREATE TABLE range_partition_nulls  
PARTITION OF range_partitioned  
(CHECK (name IS NULL))  
DEFAULT;
```

The Result

```
\d+ range_partitioned
```

```
                Partitioned table "public.range_partitioned"
```

Column	Type	Collation	Nullable	Default	Storage	Compression	...
name	text				extended		...

```
Partition key: RANGE (name)
```

```
Partitions: range_partition_j_to_s FOR VALUES FROM ('j') TO ('s'),  
            range_partition_less_j FOR VALUES FROM (MINVALUE) TO ('j'),  
            range_partition_s_greater FOR VALUES FROM ('s') TO (MAXVALUE),  
            range_partition_nulls DEFAULT
```

Hash Partitioned Table Creation

```
CREATE TABLE hash_partitioned (name TEXT)  
PARTITION BY HASH (name);
```

```
CREATE TABLE hash_partition_mod_0  
PARTITION OF hash_partitioned  
FOR VALUES WITH (MODULUS 3, REMAINDER 0);
```

```
CREATE TABLE hash_partition_mod_1  
PARTITION OF hash_partitioned  
FOR VALUES WITH (MODULUS 3, REMAINDER 1);
```

```
CREATE TABLE hash_partition_mod_2  
PARTITION OF hash_partitioned  
FOR VALUES WITH (MODULUS 3, REMAINDER 2);
```

The Result

```
\d+ hash_partitioned
```

```
Partitioned table "public.hash_partitioned"
```

Column	Type	Collation	Nullable	Default	Storage	Compression	...
name	text				extended		...

```
Partition key: HASH (name)
```

```
Partitions: hash_partition_mod_0 FOR VALUES WITH (modulus 3, remainder 0),  
            hash_partition_mod_1 FOR VALUES WITH (modulus 3, remainder 1),  
            hash_partition_mod_2 FOR VALUES WITH (modulus 3, remainder 2)
```

List Partitioned Table Creation

```
CREATE TYPE employment_status_type AS ENUM ('employed', 'unemployed', 'retired');
```

```
CREATE TABLE list_partitioned (  
    name TEXT,  
    employment_status employment_status_type  
)  
PARTITION BY LIST (employment_status);
```

```
CREATE TABLE list_partition_employed  
PARTITION OF list_partitioned  
FOR VALUES IN ('employed');
```

```
CREATE TABLE list_partition_unemployed  
PARTITION OF list_partitioned  
FOR VALUES IN ('unemployed');
```

```
-- allow NULL partition key values  
CREATE TABLE list_partition_retired_and_null  
PARTITION OF list_partitioned  
FOR VALUES IN ('retired', NULL);
```

The Result

```
\d+ list_partitioned
```

```
        Partitioned table "public.list_partitioned"
   Column          |          Type          | Collation | Nullable | Default |...
-----+-----+-----+-----+-----+...
 name              | text                   |           |          |         |...
 employment_status | employment_status_type |           |          |         |...
Partition key: LIST (employment_status)
Partitions: list_partition_employed FOR VALUES IN ('employed'),
             list_partition_retired_and_null FOR VALUES IN ('retired', NULL),
             list_partition_unemployed FOR VALUES IN ('unemployed')
```


Populating the Range Partitioned Table

```
-- This method of generating random data is explained at
-- https://momjian.us/main/blogs/pgblog/2012.html#July\_24\_2012
INSERT INTO range_partitioned
SELECT
(
  SELECT initcap(string_agg(x, ''))
  FROM (
    SELECT chr(ascii('a') + floor(random() * 26)::integer)
    FROM generate_series(1, 2 + (random() * 8)::integer + b * 0)
  ) AS y(x)
)
FROM generate_series(1, 100000) AS a(b);
```

Populating the Hash Partitioned Table

```
INSERT INTO hash_partitioned
SELECT
(
  SELECT initcap(string_agg(x, ''))
  FROM (
    SELECT chr(ascii('a') + floor(random() * 26)::integer)
    FROM generate_series(1, 2 + (random() * 8)::integer + b * 0)
  ) AS y(x)
)
FROM generate_series(1, 100000) AS a(b);
```

Populating the List Partitioned Table

```
INSERT INTO list_partitioned
SELECT
(
  SELECT initcap(string_agg(x, ''))
  FROM (
    SELECT chr(ascii('a') + floor(random() * 26)::integer)
    FROM generate_series(1, 2 + (random() * 8)::integer + b * 0)
  ) AS y(x)
),
(
  SELECT CASE floor(random() * 3 + b * 0)
    WHEN 0 THEN 'employed'::employment_status_type
    WHEN 1 THEN 'unemployed'::employment_status_type
    WHEN 2 THEN 'retired'::employment_status_type
  END
)
FROM generate_series(1, 100000) AS a(b);
```

Inserting NULL Values

```
INSERT INTO range_partitioned VALUES (NULL);  
INSERT INTO hash_partitioned VALUES (NULL);  
INSERT INTO list_partitioned VALUES ('test', NULL);
```

Creating Indexes

```
CREATE INDEX i_range_partitioned ON range_partitioned (name);  
CREATE INDEX i_hash_partitioned ON hash_partitioned (name);  
CREATE INDEX i_list_partitioned ON list_partitioned (name);  
  
ANALYZE;
```

Where are NULLs Stored?

-- NULLs are stored in the DEFAULT range partition.

```
SELECT *, tableoid::regclass
FROM range_partitioned
WHERE name IS NULL;
  name |      tableoid
-----+-----
(null) | range_partition_nulls
```

-- NULLs are always stored in the REMAINDER 0 hash partition;
-- see src/backend/partitioning/partbounds.c:compute_partition_hash_value()

```
SELECT *, tableoid::regclass
FROM hash_partitioned
WHERE name IS NULL;
  name |      tableoid
-----+-----
(null) | hash_partition_mod_0
```

-- NULLs are stored in the list partition for NULL values.

```
SELECT *, tableoid::regclass
FROM list_partitioned
WHERE employment_status IS NULL;
  name | employment_status |      tableoid
-----+-----+-----
test | (null)             | list_partition_retired_and_null
```

First Five Range Partitioned Rows

```
SELECT *, tableoid::regclass
FROM range_partitioned
ORDER BY 2, 1
LIMIT 5;
```

name	tableoid
Aa	range_partition_less_j
Aa	range_partition_less_j
Aa	range_partition_less_j
Aa	range_partition_less_j
Aa	range_partition_less_j

Random Range Partitioned Rows

```
WITH sample AS
(
  SELECT *, tableoid::regclass
  FROM range_partitioned
  ORDER BY random()
  LIMIT 5
)
SELECT * FROM sample
ORDER BY 2, 1;
```

name	tableoid
Gmgarubn	range_partition_less_j
Ousrtai	range_partition_j_to_s
Pbtmufce	range_partition_j_to_s
Qgt	range_partition_j_to_s
Ymsqpxxm	range_partition_s_greater

Random Hash Partitioned Rows

```
WITH sample AS  
(  
    SELECT *, tableoid::regclass  
    FROM hash_partitioned  
    ORDER BY random()  
    LIMIT 5  
)  
SELECT * FROM sample  
ORDER BY 2, 1;
```

name	tableoid
Yxh	hash_partition_mod_0
Asp	hash_partition_mod_1
Bgbvewd	hash_partition_mod_1
Jemquglx	hash_partition_mod_2
Xtlvuqc	hash_partition_mod_2

Random List Partitioned Rows

WITH sample AS

```
(  
  SELECT *, tableoid::regclass  
  FROM list_partitioned  
  ORDER BY random()  
  LIMIT 5  
)
```

SELECT * FROM sample

ORDER BY 3, 2, 1;

name	employment_status	tableoid
Iwxcn	employed	list_partition_employed
Btlfaascx	unemployed	list_partition_unemployed
Sz	unemployed	list_partition_unemployed
Xpdi	unemployed	list_partition_unemployed
Uwunkpdhmv	retired	list_partition_retired_and_null

Selecting Range Partition Boundaries

```
-- Use lower case for range boundaries if your collation sorts lower case first for case-insensitive equal string,  
-- e.g., range 'j' to 's' would include 'j', but 'J' to 'S' would not. 'ja' would be in 'J' to 'S' since 'ja' is  
-- not case-insensitive equal to 'J'.
```

```
SHOW lc_collate;
```

```
lc_collate
```

```
-----  
en_US.UTF-8
```

```
-- Case ordering ignored because of case-insensitive inequality.
```

```
-- https://www.unicode.org/reports/tr10/#Scope
```

```
SELECT 'a' < 'J' AND 'J' < 'z';
```

```
?column?
```

```
-----  
t
```

```
SELECT 'ja' < 'Jb' AND 'Jc' < 'jd';
```

```
?column?
```

```
-----  
t
```

```
-- Case ordering only honored for case-insensitive equality.
```

```
SELECT 'ja' < 'Ja';
```

```
?column?
```

```
-----  
t
```

```
SELECT 'island' < 'Island' AND 'islaNd' < 'iSlanD';
```

```
?column?
```

```
-----  
t
```

3. Partition Pruning



<https://www.flickr.com/photos/anguskirk/>

Pruning Stages

Pruning eliminates access to unnecessary partitions. There are three possible stages of pruning, earlier ones being more efficient than later ones:

1. Optimizer

- shown by EXPLAIN

2. Executor initialization

- shown by EXPLAIN (ANALYZE) as “Subplans Removed”
- appears under “Append” and “Merge Append”
- see [src/backend/executor/execPartition.c::ExecInitPartitionPruning\(\)](#)

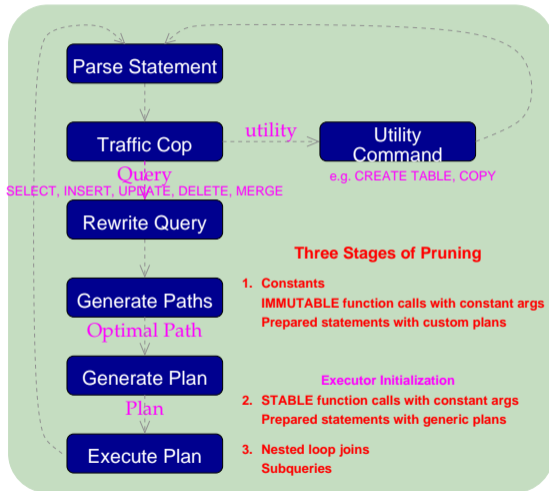
3. Executor running

- shown by EXPLAIN (ANALYZE) as “never executed”
- happens during “Append” and “Merge Append”
- see [src/backend/executor/execPartition.c::ExecFindMatchingSubPlans\(\)](#)

<https://momjian.us/main/presentations/performance.html#optimizer>

<https://momjian.us/main/presentations/performance.html#beyond>

Pruning Stages Diagram



Pruning Using NULL Constants: Stage 1

```
\set EXPLAIN 'EXPLAIN (COSTS OFF)'
```

```
:EXPLAIN SELECT *  
FROM range_partitioned  
WHERE name IS NULL;
```

```
QUERY PLAN
```

```
-----  
Seq Scan on range_partition_nulls range_partitioned  
  Filter: (name IS NULL)
```

```
-- NULLs are always stored in the REMAINDER 0 hash partition.
```

```
:EXPLAIN SELECT *  
FROM hash_partitioned  
WHERE name IS NULL;
```

```
QUERY PLAN
```

```
-----  
Index Only Scan using hash_partition_mod_0_name_idx on hash_partition_mod_0 hash_partitioned  
  Index Cond: (name IS NULL)
```

```
:EXPLAIN SELECT *  
FROM list_partitioned  
WHERE employment_status IS NULL;
```

```
QUERY PLAN
```

```
-----  
Seq Scan on list_partition_retired_and_null list_partitioned  
  Filter: (employment_status IS NULL)
```

Pruning Using Non-NULL Constants: Stage 1

```
:EXPLAIN SELECT *  
FROM range_partitioned  
WHERE name = 'Ma';
```

QUERY PLAN

```
Index Only Scan using range_partition_j_to_s_name_idx on range_partition_j_to_s range_partitioned  
Index Cond: (name = 'Ma'::text)
```

```
:EXPLAIN SELECT *  
FROM hash_partitioned  
WHERE name = 'Ma';
```

QUERY PLAN

```
Index Only Scan using hash_partition_mod_2_name_idx on hash_partition_mod_2 hash_partitioned  
Index Cond: (name = 'Ma'::text)
```

```
:EXPLAIN SELECT *  
FROM list_partitioned  
WHERE employment_status = 'retired';
```

QUERY PLAN

```
Seq Scan on list_partition_retired_and_null list_partitioned  
Filter: (employment_status = 'retired'::employment_status_type)
```


Use of Pruning and Per-Partition Index: Stage 1

```
:EXPLAIN SELECT *  
FROM list_partitioned  
WHERE employment_status = 'retired' AND  
      name = 'Ma';
```

QUERY PLAN

```
-----  
Index Scan using list_partition_retired_and_null_name_idx on list_partition_retired_and_null list_partitioned  
  Index Cond: (name = 'Ma'::text)  
  Filter: (employment_status = 'retired'::employment_status_type)
```

```
\d+ list_partitioned
```

```
          Partitioned table "public.list_partitioned"  
   Column          |          Type          | Collation | Nullable | Default | Storage  | Compression | Stats target | Description  
-----+-----+-----+-----+-----+-----+-----+-----+-----  
name               | text                  |           |          |         | extended |             |             |  
employment_status | employment_status_type |           |          |         | plain    |             |             |  
Partition key: LIST (employment_status)  
Indexes:  
  "i_list_partitioned" btree (name)  
Partitions: list_partition_employed FOR VALUES IN ('employed'),  
            list_partition_retired_and_null FOR VALUES IN ('retired', NULL),  
            list_partition_unemployed FOR VALUES IN ('unemployed')
```

Pruning of Custom-Plan Prepared Statements: Stage 1

```
PREPARE part_test AS  
SELECT *  
FROM range_partitioned  
WHERE name = $1;
```

```
:EXPLAIN EXECUTE part_test('Ba');
```

QUERY PLAN

```
-----  
Index Only Scan using range_partition_less_j_name_idx on range_partition_less_j range_partitioned  
Index Cond: (name = 'Ba'::text)
```

```
:EXPLAIN EXECUTE part_test('Ma');
```

QUERY PLAN

```
-----  
Index Only Scan using range_partition_j_to_s_name_idx on range_partition_j_to_s range_partitioned  
Index Cond: (name = 'Ma'::text)
```

```
:EXPLAIN EXECUTE part_test('Ta');
```

QUERY PLAN

```
-----  
Index Only Scan using range_partition_s_greater_name_idx on range_partition_s_greater range_partitioned  
Index Cond: (name = 'Ta'::text)
```

By default, custom plans are used for the first five executions, and then generic plans optionally used.

Pruning of Generic-Plan Prepared Statements: Stage 2

```
\set EXPLAIN_ANALYZE 'EXPLAIN (ANALYZE, SUMMARY OFF, TIMING OFF, COSTS OFF)'
```

```
-- force a generic plan
```

```
SET plan_cache_mode TO force_generic_plan;
```

```
-- Pruning happens during executor initialization
```

```
:EXPLAIN_ANALYZE EXECUTE part_test('Ba');
```

QUERY PLAN

```
-----  
Append (actual rows=9 loops=1)
```

```
Subplans Removed: 2
```

```
-> Index Only Scan using range_partition_less_j_name_idx on range_partition_less_j range_partitioned_1 (actual rows=9 loops=1)  
    Index Cond: (name = $1)  
    Heap Fetches: 0
```

```
:EXPLAIN_ANALYZE EXECUTE part_test('Ma');
```

QUERY PLAN

```
-----  
Append (actual rows=6 loops=1)
```

```
Subplans Removed: 2
```

```
-> Index Only Scan using range_partition_j_to_s_name_idx on range_partition_j_to_s range_partitioned_1 (actual rows=6 loops=1)  
    Index Cond: (name = $1)  
    Heap Fetches: 0
```

```
:EXPLAIN_ANALYZE EXECUTE part_test('Ta');
```

QUERY PLAN

```
-----  
Append (actual rows=12 loops=1)
```

```
Subplans Removed: 2
```

```
-> Index Only Scan using range_partition_s_greater_name_idx on range_partition_s_greater range_partitioned_1 (actual rows=12 loops=1)  
    Index Cond: (name = $1)  
    Heap Fetches: 0
```

Pruning of IMMUTABLE Function Calls

-- IMMUTABLE function calls are evaluated in the optimizer.

\do+ ||

Schema	Name	Left arg type	Right arg type	List of operators		Function	Description
					Result type		
pg_catalog		anycompatible	anycompatiblearray	anycompatiblearray	anycompatiblearray	array_prepend	prepend element onto front of array
pg_catalog		anycompatiblearray	anycompatible	anycompatiblearray	anycompatiblearray	array_append	append element onto end of array
pg_catalog		anycompatiblearray	anycompatiblearray	anycompatiblearray	anycompatiblearray	array_cat	concatenate
pg_catalog		anynonarray	text	text	text	anytextcat	concatenate
pg_catalog		bit varying	bit varying	bit varying	bit varying	bitcat	concatenate
pg_catalog		bytea	bytea	bytea	bytea	byteacat	concatenate
pg_catalog		jsonb	jsonb	jsonb	jsonb	jsonb_concat	concatenate
pg_catalog		text	anynonarray	text	text	textanycat	concatenate
pg_catalog		text	text	text	text	textcat	concatenate
pg_catalog		tsquery	tsquery	tsquery	tsquery	tsquery_or	OR-concatenate
pg_catalog		tsvector	tsvector	tsvector	tsvector	tsvector_concat	concatenate

Pruning of IMMUTABLE Function Calls

```
\x on
```

```
\df+ textcat
```

```
-[ RECORD 1 ]-----+-----  
Schema          | pg_catalog  
Name            | textcat  
Result data type | text  
Argument data types | text, text  
Type            | func  
Volatility       | immutable  
Parallel        | safe  
Owner           | postgres  
Security        | invoker  
Access privileges |  
Language        | internal  
Source code     | textcat  
Description     | implementation of || operator
```

```
\x off
```

Pruning of IMMUTABLE Function Calls: Stage 1

```
:EXPLAIN_ANALYZE SELECT *, tableoid::regclass  
FROM range_partitioned  
WHERE name = 'M' || 'a';
```

QUERY PLAN

```
Index Scan using range_partition_j_to_s_name_idx on range_partition_j_to_s range_partitioned (actual rows=6 loops=1)  
  Index Cond: (name = 'Ma'::text)
```

Pruning of STABLE Function Calls

-- STABLE function calls can also cause this.

\x on

\df+ concat

List of functions

-[RECORD 1]-----+-----

Schema	pg_catalog
Name	concat
Result data type	text
Argument data types	VARIADIC "any"
Type	func
Volatility	stable
Parallel	safe
Owner	postgres
Security	invoker
Access privileges	
Language	internal
Source code	text_concat
Description	concatenate values

\x off

Pruning of STABLE Function Calls: Stage 2

```
-- pruning happens during executor initialization
:EXPLAIN_ANALYZE SELECT *, tableoid::regclass
FROM range_partitioned
WHERE name = concat('M', 'a');
```

QUERY PLAN

Append (actual rows=6 loops=1)

Subplans Removed: 2

-> Index Scan using range_partition_j_to_s_name_idx on range_partition_j_to_s range_partitioned_1 (actual rows=6 loops=1)
Index Cond: (name = concat('M', 'a'))

Pruning of Subqueries: Stage 3

```
CREATE TABLE nested_outer (name) AS VALUES ('Pa'), ('Qa'), ('Ra');
```

```
ANALYZE nested_outer;
```

```
-- pruning happens during executor running
```

```
:EXPLAIN_ANALYZE SELECT *
```

```
FROM range_partitioned
```

```
WHERE name IN (SELECT * FROM nested_outer);
```

QUERY PLAN

```
Nested Loop (actual rows=24 loops=1)
```

```
-> HashAggregate (actual rows=3 loops=1)
```

```
    Group Key: nested_outer.name
```

```
    Batches: 1 Memory Usage: 24kB
```

```
    -> Seq Scan on nested_outer (actual rows=3 loops=1)
```

```
-> Append (actual rows=8 loops=3)
```

```
    -> Index Only Scan using range_partition_less_j_name_idx on range_partition_less_j range_partitioned_1 (never executed)
```

```
        Index Cond: (name = nested_outer.name)
```

```
        Heap Fetches: 0
```

```
    -> Index Only Scan using range_partition_j_to_s_name_idx on range_partition_j_to_s range_partitioned_2 (actual rows=8 loops=3)
```

```
        Index Cond: (name = nested_outer.name)
```

```
        Heap Fetches: 0
```

```
    -> Index Only Scan using range_partition_s_greater_name_idx on range_partition_s_greater range_partitioned_3 (never executed)
```

```
        Index Cond: (name = nested_outer.name)
```

```
        Heap Fetches: 0
```

```
    -> Seq Scan on range_partition_nulls range_partitioned_4 (never executed)
```

```
        Filter: (nested_outer.name = name)
```

Pruning of Joins: Stage 3

```
-- pruning happens during executor running
:EXPLAIN_ANALYZE SELECT *
FROM nested_outer JOIN range_partitioned USING (name);
```

QUERY PLAN

```
Nested Loop (actual rows=24 loops=1)
-> Seq Scan on nested_outer (actual rows=3 loops=1)
-> Append (actual rows=8 loops=3)
  -> Index Only Scan using range_partition_less_j_name_idx on range_partition_less_j range_partitioned_1 (never executed)
      Index Cond: (name = nested_outer.name)
      Heap Fetches: 0
  -> Index Only Scan using range_partition_j_to_s_name_idx on range_partition_j_to_s range_partitioned_2 (actual rows=8 loops=3)
      Index Cond: (name = nested_outer.name)
      Heap Fetches: 0
  -> Index Only Scan using range_partition_s_greater_name_idx on range_partition_s_greater range_partitioned_3 (never executed)
      Index Cond: (name = nested_outer.name)
      Heap Fetches: 0
  -> Seq Scan on range_partition_nulls range_partitioned_4 (never executed)
      Filter: (nested_outer.name = name)
```

4. Partition-Local Optimizations



<https://www.flickr.com/photos/donaldjudge/>

Partition-Local Optimizations

Partition-local optimizations perform operations on individual partitions and combine their results, rather than operating only on partitioned tables as a whole. Postgres currently supports such optimizations for aggregates and joins.

Combining partition-local results can be expensive, and therefore these optimizations are disabled by default. Performance testing is recommended before enabling these for production queries.

Aggregates Without `partitionwise_aggregate`: Range

```
-- partitionwise_aggregate is disabled by default.  
:EXPLAIN_ANALYZE SELECT name, COUNT(*)  
FROM range_partitioned  
GROUP BY name;
```

QUERY PLAN

```
HashAggregate (actual rows=90608 loops=1)  
  Group Key: range_partitioned.name  
  Batches: 5  Memory Usage: 8241kB  Disk Usage: 1552kB  
  -> Append (actual rows=100001 loops=1)  
    -> Seq Scan on range_partition_less_j range_partitioned_1 (actual rows=34626 loops=1)  
    -> Seq Scan on range_partition_j_to_s range_partitioned_2 (actual rows=34535 loops=1)  
    -> Seq Scan on range_partition_s_greater range_partitioned_3 (actual rows=30839 loops=1)  
    -> Seq Scan on range_partition_nulls range_partitioned_4 (actual rows=1 loops=1)
```

Aggregates Without partitionwise_aggregate: Hash

```
:EXPLAIN_ANALYZE SELECT name, COUNT(*)  
FROM hash_partitioned  
GROUP BY name;
```

QUERY PLAN

HashAggregate (actual rows=90645 loops=1)

Group Key: hash_partitioned.name

Batches: 5 Memory Usage: 8241kB Disk Usage: 1544kB

-> Append (actual rows=100001 loops=1)

-> Seq Scan on hash_partition_mod_0 hash_partitioned_1 (actual rows=33530 loops=1)

-> Seq Scan on hash_partition_mod_1 hash_partitioned_2 (actual rows=32979 loops=1)

-> Seq Scan on hash_partition_mod_2 hash_partitioned_3 (actual rows=33492 loops=1)

Aggregates Without partitionwise_aggregate: List

```
:EXPLAIN_ANALYZE SELECT employment_status, COUNT(*)  
FROM list_partitioned  
GROUP BY employment_status;
```

QUERY PLAN

HashAggregate (actual rows=4 loops=1)

Group Key: list_partitioned.employment_status

Batches: 1 Memory Usage: 24kB

-> Append (actual rows=100001 loops=1)

-> Seq Scan on list_partition_employed list_partitioned_1 (actual rows=33027 loops=1)

-> Seq Scan on list_partition_unemployed list_partitioned_2 (actual rows=33467 loops=1)

-> Seq Scan on list_partition_retired_and_null list_partitioned_3 (actual rows=33507 loops=1)

Aggregates With partitionwise_aggregate: Range

```
SET enable_partitionwise_aggregate = true;
```

```
-- needed because the cost of combining per-partition HashAggregates results
```

```
-- with many distinct values is high
```

```
SET cpu_tuple_cost = 0;
```

```
:EXPLAIN_ANALYZE SELECT name, COUNT(*)
```

```
FROM range_partitioned
```

```
GROUP BY name;
```

QUERY PLAN

```
-----  
Append (actual rows=90608 loops=1)  
-> HashAggregate (actual rows=31382 loops=1)  
    Group Key: range_partitioned.name  
    Batches: 1 Memory Usage: 4113kB  
    -> Seq Scan on range_partition_less_j range_partitioned (actual rows=34626 loops=1)  
-> HashAggregate (actual rows=31256 loops=1)  
    Group Key: range_partitioned_1.name  
    Batches: 1 Memory Usage: 4113kB  
    -> Seq Scan on range_partition_j_to_s range_partitioned_1 (actual rows=34535 loops=1)  
-> HashAggregate (actual rows=27969 loops=1)  
    Group Key: range_partitioned_2.name  
    Batches: 1 Memory Usage: 3857kB  
    -> Seq Scan on range_partition_s_greater range_partitioned_2 (actual rows=30839 loops=1)  
-> HashAggregate (actual rows=1 loops=1)  
    Group Key: range_partitioned_3.name  
    Batches: 1 Memory Usage: 24kB  
    -> Seq Scan on range_partition_nulls range_partitioned_3 (actual rows=1 loops=1)
```


Aggregates With partitionwise_aggregate: Hash

```
:EXPLAIN_ANALYZE SELECT name, COUNT(*)  
FROM hash_partitioned  
GROUP BY name;
```

QUERY PLAN

```
-----  
Append (actual rows=90645 loops=1)  
-> HashAggregate (actual rows=30465 loops=1)  
    Group Key: hash_partitioned.name  
    Batches: 1 Memory Usage: 4113kB  
    -> Seq Scan on hash_partition_mod_0 hash_partitioned (actual rows=33530 loops=1)  
-> HashAggregate (actual rows=29868 loops=1)  
    Group Key: hash_partitioned_1.name  
    Batches: 1 Memory Usage: 4113kB  
    -> Seq Scan on hash_partition_mod_1 hash_partitioned_1 (actual rows=32979 loops=1)  
-> HashAggregate (actual rows=30312 loops=1)  
    Group Key: hash_partitioned_2.name  
    Batches: 1 Memory Usage: 4113kB  
    -> Seq Scan on hash_partition_mod_2 hash_partitioned_2 (actual rows=33492 loops=1)
```

Aggregates With partitionwise_aggregate: List

```
-- not needed for the next query because few distinct values  
RESET cpu_tuple_cost;
```

```
:EXPLAIN ANALYZE SELECT employment_status, COUNT(*)  
FROM list_partitioned  
GROUP BY employment_status;
```

QUERY PLAN

```
-----  
Append (actual rows=4 loops=1)  
-> HashAggregate (actual rows=1 loops=1)  
   Group Key: list_partitioned.employment_status  
   Batches: 1  Memory Usage: 24kB  
   -> Seq Scan on list_partition_employed list_partitioned (actual rows=33027 loops=1)  
-> HashAggregate (actual rows=1 loops=1)  
   Group Key: list_partitioned_1.employment_status  
   Batches: 1  Memory Usage: 24kB  
   -> Seq Scan on list_partition_unemployed list_partitioned_1 (actual rows=33467 loops=1)  
-> HashAggregate (actual rows=2 loops=1)  
   Group Key: list_partitioned_2.employment_status  
   Batches: 1  Memory Usage: 24kB  
   -> Seq Scan on list_partition_retired_and_null list_partitioned_2 (actual rows=33507 loops=1)
```

```
RESET enable_partitionwise_aggregate;
```

Cross Partition Join: Setup

```
CREATE TABLE range_partitioned2 (name TEXT)  
PARTITION BY RANGE (name);
```

```
CREATE TABLE range_partition_less_j2  
PARTITION OF range_partitioned2  
FOR VALUES FROM (MINVALUE) TO ('j');
```

```
CREATE TABLE range_partition_j_to_s2  
PARTITION OF range_partitioned2  
FOR VALUES FROM ('j') TO ('s');
```

```
CREATE TABLE range_partition_s_greater2  
PARTITION OF range_partitioned2  
FOR VALUES FROM ('s') TO (MAXVALUE);
```

```
CREATE TABLE range_partition_nulls2  
PARTITION OF range_partitioned2  
(CHECK (name IS NULL))  
DEFAULT;
```

Cross Partition Join: Populate

```
INSERT INTO range_partitioned2
SELECT
(
  SELECT initcap(string_agg(x, ''))
  FROM (
    SELECT chr(ascii('a') + floor(random() * 26)::integer)
    FROM generate_series(1, 2 + (random() * 8)::integer + b * 0)
  ) AS y(x)
)
FROM generate_series(1, 100000) AS a(b);

ANALYZE;
```

Cross Partition Join Without `partitionwise_join`

```
-- partitionwise_aggregate is disabled by default.  
:EXPLAIN_ANALYZE SELECT *  
FROM range_partitioned JOIN range_partitioned2 USING (name);  
                                QUERY PLAN
```

```
-----  
Hash Join (actual rows=67468 loops=1)  
  Hash Cond: (range_partitioned.name = range_partitioned2.name)  
    -> Append (actual rows=100001 loops=1)  
      -> Seq Scan on range_partition_less_j range_partitioned_1 (actual rows=34626 loops=1)  
      -> Seq Scan on range_partition_j_to_s range_partitioned_2 (actual rows=34535 loops=1)  
      -> Seq Scan on range_partition_s_greater range_partitioned_3 (actual rows=30839 loops=1)  
      -> Seq Scan on range_partition_nulls range_partitioned_4 (actual rows=1 loops=1)  
    -> Hash (actual rows=100000 loops=1)  
        Buckets: 131072 Batches: 1 Memory Usage: 4833kB  
        -> Append (actual rows=100000 loops=1)  
          -> Seq Scan on range_partition_less_j2 range_partitioned2_1 (actual rows=34646 loops=1)  
          -> Seq Scan on range_partition_j_to_s2 range_partitioned2_2 (actual rows=34606 loops=1)  
          -> Seq Scan on range_partition_s_greater2 range_partitioned2_3 (actual rows=30748 loops=1)  
          -> Seq Scan on range_partition_nulls2 range_partitioned2_4 (actual rows=0 loops=1)
```

Cross Partition Join With partitionwise_join

```
SET enable_partitionwise_join = true;
```

```
:EXPLAIN_ANALYZE SELECT *  
FROM range_partitioned JOIN range_partitioned2 USING (name);
```

QUERY PLAN

Append (actual rows=67468 loops=1)

-> Hash Join (actual rows=23917 loops=1)

Hash Cond: (range_partitioned_1.name = range_partitioned2_1.name)

-> Seq Scan on range_partition_less_j range_partitioned_1 (actual rows=34626 loops=1)

-> Hash (actual rows=34646 loops=1)

Buckets: 65536 Batches: 1 Memory Usage: 1832kB

-> Seq Scan on range_partition_less_j2 range_partitioned2_1 (actual rows=34646 loops=1)

-> Hash Join (actual rows=23096 loops=1)

Hash Cond: (range_partitioned_2.name = range_partitioned2_2.name)

-> Seq Scan on range_partition_j_to_s range_partitioned_2 (actual rows=34535 loops=1)

-> Hash (actual rows=34606 loops=1)

Buckets: 65536 Batches: 1 Memory Usage: 1830kB

-> Seq Scan on range_partition_j_to_s2 range_partitioned2_2 (actual rows=34606 loops=1)

-> Hash Join (actual rows=20455 loops=1)

Hash Cond: (range_partitioned_3.name = range_partitioned2_3.name)

-> Seq Scan on range_partition_s_greater range_partitioned_3 (actual rows=30839 loops=1)

-> Hash (actual rows=30748 loops=1)

Buckets: 32768 Batches: 1 Memory Usage: 1428kB

-> Seq Scan on range_partition_s_greater2 range_partitioned2_3 (actual rows=30748 loops=1)

-> Nested Loop (actual rows=0 loops=1)

-> Seq Scan on range_partition_nulls2 range_partitioned2_4 (actual rows=0 loops=1)

-> Index Only Scan using range_partition_nulls_name_idx on range_partition_nulls range_partitioned_4 (never executed)

Index Cond: (name = range_partitioned2_4.name)

Heap Fetches: 0

5. Time-Based Partitioning



<https://www.flickr.com/photos/amylovesyah/>

Time-Based Partitioning

Time-based partitioning stores data in partitions based on some time component. New partitions need to be created to match time-based requirements, and old partitions can be archived, deleted, or moved to tablespaces with cheaper storage. Time-based partitioning have added complexity because of time zone and daylight saving time aspects.

Create Per-Month Partitions

```
CREATE TABLE month_partitioned (day DATE, temperature NUMERIC(5,2))  
PARTITION BY RANGE (day);
```

```
CREATE TABLE month_partition_2023_01  
PARTITION OF month_partitioned  
FOR VALUES FROM ('2023-01-01') TO ('2023-02-01');
```

```
CREATE TABLE month_partition_2023_02  
PARTITION OF month_partitioned  
FOR VALUES FROM ('2023-02-01') TO ('2023-03-01');
```

```
CREATE TABLE month_partition_2023_03  
PARTITION OF month_partitioned  
FOR VALUES FROM ('2023-03-01') TO ('2023-04-01');
```

```
CREATE TABLE month_partition_other  
PARTITION OF month_partitioned  
DEFAULT;
```

Populate Per-Month Partitions

```
INSERT INTO month_partitioned
SELECT
(
  SELECT '2023-01-01'::date +
    floor(random() * ('2023-04-01'::date - '2023-01-01'::date) + b * 0)::integer
),
(
  SELECT floor(random() * 10000) / 100 + b * 0
)
FROM generate_series(1, 100000) AS a(b);

CREATE INDEX i_month_partitioned ON month_partitioned (day);

ANALYZE;
```

Random Partition Rows

```
WITH sample AS  
(  
  SELECT *, tableoid::regclass  
  FROM month_partitioned  
  ORDER BY random()  
  LIMIT 5  
)
```

```
)  
SELECT * FROM sample  
ORDER BY 3, 1;
```

day	temperature	tableoid
2023-01-16	64.45	month_partition_2023_01
2023-02-07	46.03	month_partition_2023_02
2023-02-13	87.14	month_partition_2023_02
2023-03-07	97.04	month_partition_2023_03
2023-03-08	59.02	month_partition_2023_03

Partition Pruning

```
:EXPLAIN_ANALYZE SELECT *  
FROM month_partitioned  
WHERE day = '2023-02-01';
```

QUERY PLAN

```
Bitmap Heap Scan on month_partition_2023_02 month_partitioned (actual rows=1169 loops=1)  
  Recheck Cond: (day = '2023-02-01'::date)  
  Heap Blocks: exact=170  
-> Bitmap Index Scan on month_partition_2023_02_day_idx (actual rows=1169 loops=1)  
    Index Cond: (day = '2023-02-01'::date)
```

Default Partition Usage

```
INSERT INTO month_partitioned VALUES ('2023-05-01', 87.31);
```

```
:EXPLAIN_ANALYZE SELECT *  
FROM month_partitioned  
WHERE day = '2023-05-01';
```

QUERY PLAN

```
Seq Scan on month_partition_other month_partitioned (actual rows=1 loops=1)  
  Filter: (day = '2023-05-01'::date)
```

Null Uses the DEFAULT Partition

```
INSERT INTO month_partitioned VALUES (NULL, 46.24);
```

```
:EXPLAIN_ANALYZE SELECT *  
FROM month_partitioned  
WHERE day IS NULL;
```

QUERY PLAN

```
Seq Scan on month_partition_other month_partitioned (actual rows=1 loops=1)  
  Filter: (day IS NULL)  
  Rows Removed by Filter: 1
```

Attaching and Detaching Partitions

```
ALTER TABLE month_partitioned DETACH PARTITION month_partition_other;
```

```
INSERT INTO month_partitioned VALUES (NULL, 46.24);
```

```
ERROR: no partition of relation "month_partitioned" found for row
```

```
DETAIL: Partition key of the failing row contains (day) = (null).
```

```
ALTER TABLE month_partitioned ATTACH PARTITION month_partition_other DEFAULT;
```

Pruning Using a STABLE Function

```
-- Simulate CURRENT_DATE by using the STABLE function concat().  
\set CURRENT_DATE concat(''2023-02-01'' || ''''')::date
```

```
:EXPLAIN_ANALYZE SELECT *  
FROM month_partitioned  
WHERE day = :CURRENT_DATE;
```

QUERY PLAN

Append (actual rows=1169 loops=1)

Subplans Removed: 3

-> Bitmap Heap Scan on **month_partition_2023_02** month_partitioned_1 (actual rows=1169 loops=1)

Recheck Cond: (day = (concat('2023-02-01'::text))::date)

Heap Blocks: exact=170

-> Bitmap Index Scan on month_partition_2023_02_day_idx (actual rows=1169 loops=1)

Index Cond: (day = (concat('2023-02-01'::text))::date)

Partition Expiration and Creation

```
CREATE TABLE month_partition_2023_04  
PARTITION OF month_partitioned  
FOR VALUES FROM ('2023-04-01') TO ('2023-05-01');  
  
DROP TABLE month_partition_2023_01;
```

The table could also be archived before deletion or moved to a different tablespace. `pg_partman` (https://github.com/pgpartman/pg_partman) can help with auto-partition creation.

Create Timestamp with Time Zone Partitions

```
SET timezone = 'America/New_York';
```

```
CREATE TABLE month_ts_tz_partitioned (event_time TIMESTAMP WITH TIME ZONE, temperature NUMERIC(5,2))  
PARTITION BY RANGE (event_time);
```

```
-- date evaluated at creation time
```

```
CREATE TABLE month_ts_tz_partition_2023_01  
PARTITION OF month_ts_tz_partitioned  
FOR VALUES FROM ('2023-01-01') TO ('2023-02-01');
```

```
CREATE TABLE month_ts_tz_partition_2023_02  
PARTITION OF month_ts_tz_partitioned  
FOR VALUES FROM ('2023-02-01') TO ('2023-03-01');
```

```
CREATE TABLE month_ts_tz_partition_2023_03  
PARTITION OF month_ts_tz_partitioned  
FOR VALUES FROM ('2023-03-01') TO ('2023-04-01');
```

```
CREATE TABLE month_ts_tz_partition_other  
PARTITION OF month_ts_tz_partitioned  
DEFAULT;
```

DATE Data Type Has No Time Zone

```
SELECT EXTRACT(EPOCH FROM '2023-01-01'::date);  
      extract
```

```
-----  
1672531200
```

```
SELECT EXTRACT(EPOCH FROM '2023-01-01 00:00:00-00'::timestampz);  
      extract
```

```
-----  
1672531200.000000
```

```
SELECT EXTRACT(EPOCH FROM '2023-01-01'::timestampz);  
      extract
```

```
-----  
1672549200.000000
```

```
SELECT EXTRACT(EPOCH FROM '2023-01-01 00:00:00-05'::timestampz);  
      extract
```

```
-----  
1672549200.000000
```

Populate Partitions

```
INSERT INTO month_ts_tz_partitioned
SELECT
(
  SELECT '2023-01-01 00:00:00'::timestampz +
    (floor(random() *
      (extract(EPOCH FROM '2023-04-01'::timestampz) -
        extract(EPOCH FROM '2023-01-01'::timestampz)) +
      b * 0)::integer || 'seconds')::interval
),
(
  SELECT floor(random() * 10000) / 100 + b * 0
)
FROM generate_series(1, 100000) AS a(b);

-- add row to the DEFAULT partition
INSERT INTO month_ts_tz_partitioned VALUES ('2023-04-05 00:00:00', 50);

CREATE INDEX i_month_ts_tz_partitioned ON month_ts_tz_partitioned (event_time);

ANALYZE;
```

We must cast to TIMESTAMPTZ, not DATE, to align with the partition boundaries.

Partition Details

```
\d+ month_ts_tz_partitioned
```

```
Partitioned table "public.month_ts_tz_partitioned"
```

Column	Type	Collation	Nullable	Default	Storage	Compression	Stats target	Description
event_time	timestamp with time zone				plain			
temperature	numeric(5,2)				main			

```
Partition key: RANGE (event_time)
```

```
Indexes:
```

```
"i_month_ts_tz_partitioned" btree (event_time)
```

```
Partitions: month_ts_tz_partition_2023_01 FOR VALUES FROM ('2023-01-01 00:00:00-05') TO ('2023-02-01 00:00:00-05'),  
            month_ts_tz_partition_2023_02 FOR VALUES FROM ('2023-02-01 00:00:00-05') TO ('2023-03-01 00:00:00-05'),  
            month_ts_tz_partition_2023_03 FOR VALUES FROM ('2023-03-01 00:00:00-05') TO ('2023-04-01 00:00:00-04'),  
            month_ts_tz_partition_other DEFAULT
```

First Partition Row

```
SELECT CURRENT_TIMESTAMP;  
        current_timestamp
```

```
-----  
2023-03-20 09:35:05.375191-04
```

```
SELECT *, tableoid::regclass  
FROM   month_ts_tz_partitioned  
ORDER BY 1  
LIMIT 1;
```

```
        event_time          | temperature |          tableoid  
-----+-----+-----  
2023-01-01 00:01:26-05 |          8.14 | month_ts_tz_partition_2023_01
```

First Partition Row in a Different Time Zone

```
SET timezone = 'Asia/Tokyo';
```

```
SELECT CURRENT_TIMESTAMP;  
       current_timestamp
```

```
-----  
2023-03-20 22:35:05.376393+09
```

```
SELECT *, tableoid::regclass  
FROM   month_ts_tz_partitioned  
ORDER BY 1  
LIMIT 1;
```

```
-----+-----+-----  
event_time | temperature | tableoid  
-----+-----+-----  
2023-01-01 14:01:26+09 | 8.14 | month_ts_tz_partition_2023_01
```

Partition Bounds Adjusted

```
\d+ month_ts_tz_partitioned
```

```
Partitioned table "public.month_ts_tz_partitioned"
  Column |          Type          | Collation | Nullable | Default | Storage | Compression | Stats target | Description
-----+-----+-----+-----+-----+-----+-----+-----+-----
event_time | timestamp with time zone |           |          |         | plain   |              |              |
temperature | numeric(5,2)           |           |          |         | main    |              |              |
Partition key: RANGE (event_time)
Indexes:
    "i_month_ts_tz_partitioned" btree (event_time)
Partitions: month_ts_tz_partition_2023_01 FOR VALUES FROM ('2023-01-01 14:00:00+09') TO ('2023-02-01 14:00:00+09'),
            month_ts_tz_partition_2023_02 FOR VALUES FROM ('2023-02-01 14:00:00+09') TO ('2023-03-01 14:00:00+09'),
            month_ts_tz_partition_2023_03 FOR VALUES FROM ('2023-03-01 14:00:00+09') TO ('2023-04-01 13:00:00+09'),
            month_ts_tz_partition_other DEFAULT
```


The Same in UTC

```
SET timezone = 'UTC';
```

```
SELECT *, tableoid::regclass  
FROM month_ts_tz_partitioned  
ORDER BY 1  
LIMIT 1;
```

event_time	temperature	tableoid
2023-01-01 05:01:26+00	8.14	month_ts_tz_partition_2023_01

The Same in UTC

```
\d+ month_ts_tz_partitioned
```

```
Partitioned table "public.month_ts_tz_partitioned"
```

Column	Type	Collation	Nullable	Default	Storage	Compression	Stats target	Description
event_time	timestamp with time zone				plain			
temperature	numeric(5,2)				main			

```
Partition key: RANGE (event_time)
```

```
Indexes:
```

```
"i_month_ts_tz_partitioned" btree (event_time)
```

```
Partitions: month_ts_tz_partition_2023_01 FOR VALUES FROM ('2023-01-01 05:00:00+00') TO ('2023-02-01 05:00:00+00'),  
            month_ts_tz_partition_2023_02 FOR VALUES FROM ('2023-02-01 05:00:00+00') TO ('2023-03-01 05:00:00+00'),  
            month_ts_tz_partition_2023_03 FOR VALUES FROM ('2023-03-01 05:00:00+00') TO ('2023-04-01 04:00:00+00'),  
            month_ts_tz_partition_other DEFAULT
```

No Pruning of a Function Call on a Column

```
SET timezone = 'America/New_York';
```

```
:EXPLAIN_ANALYZE SELECT *  
FROM month_ts_tz_partitioned  
WHERE date(event_time) = '2023-02-05';
```

QUERY PLAN

Append (actual rows=1093 loops=1)

- > Seq Scan on [month_ts_tz_partition_2023_01](#) month_ts_tz_partitioned_1 (actual rows=0 loops=1)
Filter: (date(event_time) = '2023-02-05'::date)
Rows Removed by Filter: 34240
- > Seq Scan on [month_ts_tz_partition_2023_02](#) month_ts_tz_partitioned_2 (actual rows=1093 loops=1)
Filter: (date(event_time) = '2023-02-05'::date)
Rows Removed by Filter: 30075
- > Seq Scan on [month_ts_tz_partition_2023_03](#) month_ts_tz_partitioned_3 (actual rows=0 loops=1)
Filter: (date(event_time) = '2023-02-05'::date)
Rows Removed by Filter: 34547
- > Seq Scan on [month_ts_tz_partition_other](#) month_ts_tz_partitioned_4 (actual rows=0 loops=1)
Filter: (date(event_time) = '2023-02-05'::date)
Rows Removed by Filter: 45

Date Range Can Be Pruned

```
:EXPLAIN_ANALYZE SELECT *  
FROM month_ts_tz_partitioned  
WHERE event_time >= '2023-02-05' AND  
       event_time < '2023-02-06';
```

QUERY PLAN

```
-----  
Bitmap Heap Scan on month_ts_tz_partition_2023_02 month_ts_tz_partitioned (actual rows=1093 loops=1)  
  Recheck Cond: ((event_time >= '2023-02-05 00:00:00-05'::timestamp with time zone) AND  
                 (event_time < '2023-02-06 00:00:00-05'::timestamp with time zone))  
  Heap Blocks: exact=169  
-> Bitmap Index Scan on month_ts_tz_partition_2023_02_event_time_idx (actual rows=1093 loops=1)  
   Index Cond: ((event_time >= '2023-02-05 00:00:00-05'::timestamp with time zone) AND  
               (event_time < '2023-02-06 00:00:00-05'::timestamp with time zone))
```

Date Calculation Can Be Pruned

```
-- simulate CURRENT_TIMESTAMP by using the STABLE function concat().
\set CURRENT_TIMESTAMP concat('2023-02-05 23:43:51' || ''')::timestampz
```

```
-- pruning happening during executor initialization
:EXPLAIN ANALYZE SELECT *
FROM month_ts_tz_partitioned
WHERE event_time > :CURRENT_TIMESTAMP - '24 hours'::interval AND
       event_time <= :CURRENT_TIMESTAMP;
```

QUERY PLAN

Append (actual rows=1091 loops=1)

Subplans Removed: 3

-> Bitmap Heap Scan on month_ts_tz_partition_2023_02 month_ts_tz_partitioned_1 (actual rows=1091 loops=1)
Recheck Cond: ((event_time > ((concat('2023-02-05 23:43:51')::text)::timestamp with time zone - '24:00:00'::interval)) AND
(event_time <= (concat('2023-02-05 23:43:51')::text)::timestamp with time zone))
Heap Blocks: exact=169

-> Bitmap Index Scan on month_ts_tz_partition_2023_02_event_time_idx (actual rows=1091 loops=1)
Index Cond: ((event_time > ((concat('2023-02-05 23:43:51')::text)::timestamp with time zone - '24:00:00'::interval)) AND
(event_time <= (concat('2023-02-05 23:43:51')::text)::timestamp with time zone))

Timestamp Range Can Be Pruned

```
:EXPLAIN_ANALYZE SELECT *  
FROM month_ts_tz_partitioned  
WHERE event_time >= '2023-03-01 00:00:00' AND  
       event_time < '2023-03-02 00:00:00';
```

QUERY PLAN

```
-----  
Bitmap Heap Scan on month_ts_tz_partition_2023_03 month_ts_tz_partitioned (actual rows=1101 loops=1)  
  Recheck Cond: ((event_time >= '2023-03-01 00:00:00-05'::timestamp with time zone) AND  
                 (event_time < '2023-03-02 00:00:00-05'::timestamp with time zone))  
  Heap Blocks: exact=187  
-> Bitmap Index Scan on month_ts_tz_partition_2023_03_event_time_idx (actual rows=1101 loops=1)  
   Index Cond: ((event_time >= '2023-03-01 00:00:00-05'::timestamp with time zone) AND  
               (event_time < '2023-03-02 00:00:00-05'::timestamp with time zone))
```

Where Are Per-Day Rows?

```
SELECT COUNT(*)  
FROM month_ts_tz_partitioned  
WHERE event_time >= '2023-03-01 00:00:00' AND  
       event_time < '2023-03-02 00:00:00';
```

count

1101

Where Are Per-Day Rows?

```
SELECT *, tableoid::regclass
FROM month_ts_tz_partitioned
WHERE event_time >= '2023-03-01 00:00:00' AND
       event_time < '2023-03-02 00:00:00'
ORDER BY 1
LIMIT 1;
```

event_time	temperature	tableoid
2023-03-01 00:00:20-05	49.85	month_ts_tz_partition_2023_03

```
SELECT *, tableoid::regclass
FROM month_ts_tz_partitioned
WHERE event_time >= '2023-03-01 00:00:00' AND
       event_time < '2023-03-02 00:00:00'
ORDER BY 1 DESC
LIMIT 1;
```

event_time	temperature	tableoid
2023-03-01 23:59:35-05	27.94	month_ts_tz_partition_2023_03

Query in a Different Time Zone

```
SET timezone = 'Asia/Tokyo';
```

```
:EXPLAIN ANALYZE SELECT *  
FROM month_ts_tz_partitioned  
WHERE event_time >= '2023-03-01 00:00:00' AND  
       event_time < '2023-03-02 00:00:00';
```

QUERY PLAN

```
-----  
Append (actual rows=1109 loops=1)  
-> Bitmap Heap Scan on month_ts_tz_partition_2023_02 month_ts_tz_partitioned_1 (actual rows=622 loops=1)  
    Recheck Cond: ((event_time >= '2023-03-01 00:00:00+09'::timestamp with time zone) AND  
                  (event_time < '2023-03-02 00:00:00+09'::timestamp with time zone))  
    Heap Blocks: exact=166  
-> Bitmap Index Scan on month_ts_tz_partition_2023_02_event_time_idx (actual rows=622 loops=1)  
    Index Cond: ((event_time >= '2023-03-01 00:00:00+09'::timestamp with time zone) AND  
                (event_time < '2023-03-02 00:00:00+09'::timestamp with time zone))  
-> Bitmap Heap Scan on month_ts_tz_partition_2023_03 month_ts_tz_partitioned_2 (actual rows=487 loops=1)  
    Recheck Cond: ((event_time >= '2023-03-01 00:00:00+09'::timestamp with time zone) AND  
                  (event_time < '2023-03-02 00:00:00+09'::timestamp with time zone))  
    Heap Blocks: exact=169  
-> Bitmap Index Scan on month_ts_tz_partition_2023_03_event_time_idx (actual rows=487 loops=1)  
    Index Cond: ((event_time >= '2023-03-01 00:00:00+09'::timestamp with time zone) AND  
                (event_time < '2023-03-02 00:00:00+09'::timestamp with time zone))
```

Different Count

```
SELECT COUNT(*)  
FROM month_ts_tz_partitioned  
WHERE event_time >= '2023-03-01 00:00:00' AND  
       event_time < '2023-03-02 00:00:00';
```

count

1109

Partition Range

```
SELECT *, tableoid::regclass
FROM month_ts_tz_partitioned
WHERE event_time >= '2023-03-01 00:00:00' AND
       event_time < '2023-03-02 00:00:00'
ORDER BY 1
LIMIT 1;
```

event_time	temperature	tableoid
2023-03-01 00:00:32+09	16.32	month_ts_tz_partition_2023_02

```
SELECT *, tableoid::regclass
FROM month_ts_tz_partitioned
WHERE event_time >= '2023-03-01 00:00:00' AND
       event_time < '2023-03-02 00:00:00'
ORDER BY 1 DESC
LIMIT 1;
```

event_time	temperature	tableoid
2023-03-01 23:56:22+09	19.78	month_ts_tz_partition_2023_03

Range Boundaries Are Set at Creation

-- caused by mismatch with America/New_York time zone boundaries

```
CREATE TABLE month_ts_tz_partition_2023_04
PARTITION OF month_ts_tz_partitioned
FOR VALUES FROM ('2023-04-01') TO ('2023-05-01');
ERROR: partition "month_ts_tz_partition_2023_04" would
       overlap partition "month_ts_tz_partition_2023_03"
LINE 3: FOR VALUES FROM ('2023-04-01') TO ('2023-05-01');
```

^

Matching Rows in the DEFAULT Partition

```
SET timezone = 'America/New_York';
```

```
-- caused by daylight saving time change
```

```
CREATE TABLE month_ts_tz_partition_2023_04
```

```
PARTITION OF month_ts_tz_partitioned
```

```
FOR VALUES FROM ('2023-04-01') TO ('2023-05-01');
```

```
ERROR: updated partition constraint for default partition
```

```
    "month_ts_tz_partition_other" would be violated by some row
```

Move DEFAULT Rows to a New Partition

```
BEGIN WORK;

-- lock table and/or detach DEFAULT partition?
CREATE TEMP TABLE tmp_default AS
SELECT *
FROM month_ts_tz_partition_other
WHERE event_time >= '2023-04-01 00:00:00' AND
       event_time < '2023-05-01 00:00:00';

DELETE FROM month_ts_tz_partition_other
WHERE event_time >= '2023-04-01 00:00:00' AND
       event_time < '2023-05-01 00:00:00';

CREATE TABLE month_ts_tz_partition_2023_04
PARTITION OF month_ts_tz_partitioned
FOR VALUES FROM ('2023-04-01') TO ('2023-05-01');

INSERT INTO month_ts_tz_partitioned
SELECT * FROM tmp_default;

SELECT * FROM month_ts_tz_partition_other;
event_time | temperature
-----+-----

COMMIT;
```

New Partition Contents

```
SELECT * FROM month_ts_tz_partition_2023_04;  
event_time | temperature  
-----+-----  
2023-04-05 00:00:00-04 | 50.00
```

```
SELECT * FROM month_ts_tz_partition_other;  
event_time | temperature  
-----+-----
```

6. Row Migration



<https://www.flickr.com/photos/ashokbo/>

Rows in 'j' to 's' Partition

```
SELECT *, tableoid::regclass
FROM range_partitioned
WHERE name = 'Ma'
ORDER BY 2, 1;
```

name	tableoid
Ma	range_partition_j_to_s
Ma	range_partition_j_to_s
Ma	range_partition_j_to_s
Ma	range_partition_j_to_s
Ma	range_partition_j_to_s
Ma	range_partition_j_to_s

Migration to Greater than 's' Partition

```
UPDATE range_partitioned  
SET name = 'zz_' || name  
WHERE name = 'Ma';
```

```
SELECT *, tableoid::regclass  
FROM range_partitioned  
WHERE name = 'zz_Ma'  
ORDER BY 2, 1;
```

name	tableoid
zz_Ma	range_partition_s_greater
zz_Ma	range_partition_s_greater
zz_Ma	range_partition_s_greater
zz_Ma	range_partition_s_greater
zz_Ma	range_partition_s_greater
zz_Ma	range_partition_s_greater

8. psql Support



<https://www.flickr.com/photos/feillu/>

psql support

```
COMMENT ON TABLE range_partitioned IS 'Section 2';  
COMMENT ON TABLE hash_partitioned IS 'Section 2';  
COMMENT ON TABLE list_partitioned IS 'Section 2';  
COMMENT ON TABLE range_partitioned2 IS 'Section 4';  
COMMENT ON TABLE month_partitioned IS 'Section 5';  
COMMENT ON TABLE month_ts_tz_partitioned IS 'Section 5';
```

`\dPt+`

List of partitioned tables

Schema	Name	Owner	Total size	Description
public	hash_partitioned	postgres	3888 kB	Section 2
public	list_partitioned	postgres	4280 kB	Section 2
public	month_partitioned	postgres	2896 kB	Section 5
public	month_ts_tz_partitioned	postgres	13 MB	Section 5
public	range_partitioned	postgres	3904 kB	Section 2
public	range_partitioned2	postgres	3888 kB	Section 4

8. Limitations



<https://www.flickr.com/photos/pensiero/>

Partitioning Limitations

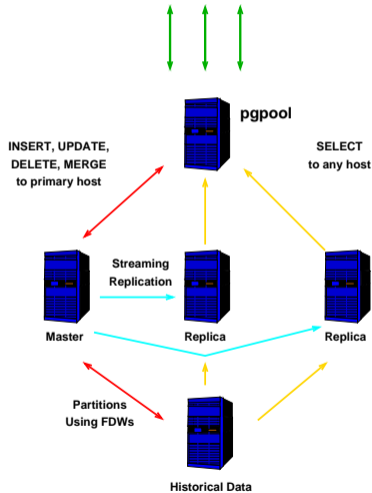
- Adding partitions
 - requires locking
 - partitions can be created, populated, and then attached to reduce locking
 - attached partition rows are verified unless CHECK constraints match partition bounds
 - requires DEFAULT partition scans unless CHECK constraints make it unnecessary
 - foreign table rows are not verified
- Removing partitions
 - requires locking
 - partitions can be detached CONCURRENTLY to reduce locking
- Function calls on columns cannot be pruned unless specified in the range
 - function calls on constants often can
- Unique indexes on partitioned tables must include the partition key columns
 - no global indexes, see https://momjian.us/main/blogs/pgblog/2020.html#July_1_2020
- CONCURRENT index creation on *partitioned* tables is not supported
 - CONCURRENT index creation on *partitions* is supported, and can then be attached

9. Complex Architectures

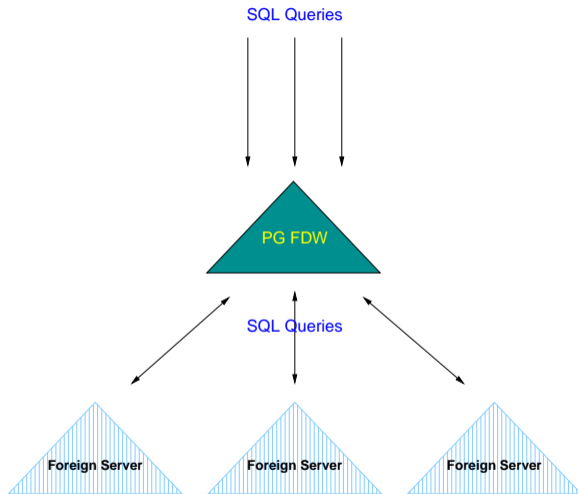


<https://www.flickr.com/photos/alexdrop/>

Foreign Servers for Archive Data



Sharding



<https://momjian.us/main/presentations/performance.html#sharding>

Conclusion



<https://momjian.us/presentations>

<https://www.flickr.com/photos/maxbraun/>