credativ

PostgreSQL Native Partitioning

From Ancient Monolithic Tables to Scalable Better Future

Josef Machytka <josef.machytka@credativ.de>

2025-08-27 - credativ Tech Talk

credativ GmbH

- Founded 1999 in Jülich, Germany
- Close ties to Open-Source Community
- More than 40 Open-Source experts
- Consulting, development, training, support (3rd-level / 24x7)
- Open-Source infrastructure with Linux, Kubernetes, Proxmox
- Open-Source databases with PostgreSQL
- DevSecOps with Ansible, Puppet, Terraform and others
- Since 2025 independent owner-managed company again

Josef Machytka

- Professional Service Consultant - PostgreSQL specialist at credativ GmbH
- 33+ years of experience with different databases
- PostgreSQL (13y), BigQuery (7y), Oracle (15y), MySQL (12y), Elasticsearch (5y), MS SQL (5y)
- 10+ years of experience with Data Ingestion pipelines, Data Analysis, Data Lake and Data Warehouse
- 3+ years of practical experience with different LLMs / AI / ML including architecture and principles
- From Czechia, living now 12 years in Berlin

- LinkedIn: linkedin.com/in/josef-machytka
- Medium: medium.com/@josef.machytka
- YouTube: youtube.com/@JosefMachytka

- GitHub: github.com/josmac69/conferences_slides
- ResearchGate: researchgate.net/profile/Josef-Machytka
- Academia.edu: academia.edu/JosefMachytka
- sessionize: sessionize.com/josefmachytka

All My Slides:

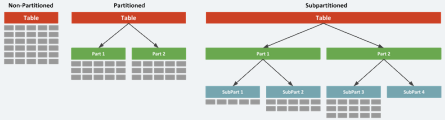
Recoded talks:

credativ

Why we need Partitioning?

Partitioning in General

- Splits large tables into smaller, more manageable partitions
- Assigns rows to partitions using a defined partition key
- Better query and maintenance performance via smaller tables
- Scans and processes only relevant partitions for each query
- Reduces index size and speeds up index lookups
- Enables efficient parallel query execution across partitions
- Supports targeted backup and restore of individual partitions
- Allows fast data lifecycle management by dropping partitions



Picture from Oracle documentation

Partitioning & PostgreSQL MVCC

- PostgreSQL MVCC turns UPDATEs into DELETE + INSERT
- Large tables with frequent updates can bloat quickly
- Heap pages fill with dead tuples, slowing scans
- Changes are randomly distributed across the table
- Large index scans on big table cause massive random IO
- Indexes only know tuple CTIDs, not visibility
- Every scan must check visibility of all tuples in table
- Huge tables lead to slower VACUUM and ANALYZE operations
- VACUUM must scan entire table to reclaim space

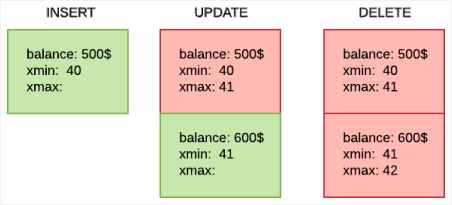
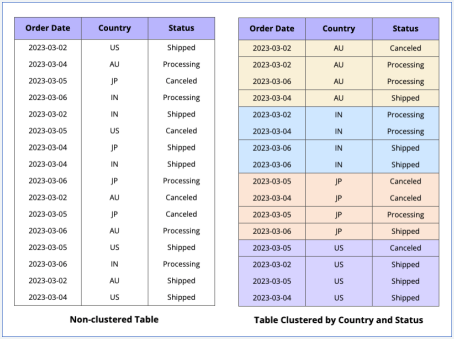


Image from the article Snapshot Isolation in PostgreSQL

Clustering

- Physically orders table data on disk by specified column(s)
- Rows with similar values are stored together
- Improves range queries by reducing random disk IO
- Most effective for immutable datasets
- Widely used in Data Warehouses and Data Lakehouses
- Partitioning & Clustering are complementary techniques
- Partitioning divides data into smaller tables
- Clustering organizes data within those tables
- Together they enhance query performance and manageability



Order Date	Country	Status
2023-03-02	US	Shipped
2023-03-04	AU	Processing
2023-03-05	JP	Canceled
2023-03-06	IN	Processing
2023-03-02	IN	Shipped
2023-03-05	US	Canceled
2023-03-04	JP	Shipped
2023-03-04	IN	Shipped
2023-03-06	JP	Processing
2023-03-02	AU	Canceled
2023-03-05	JP	Canceled
2023-03-06	AU	Processing
2023-03-05	US	Shipped
2023-03-06	IN	Processing
2023-03-02	AU	Shipped
2023-03-04	US	Shipped

Non-clustered Table

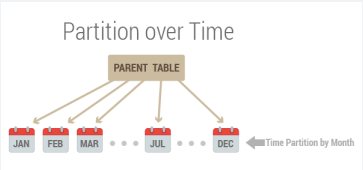
Order Date	Country	Status
2023-03-02	AU	Canceled
2023-03-02	AU	Processing
2023-03-06	AU	Processing
2023-03-04	AU	Shipped
2023-03-02	IN	Processing
2023-03-04	IN	Processing
2023-03-06	IN	Shipped
2023-03-06	IN	Shipped
2023-03-05	JP	Canceled
2023-03-04	JP	Canceled
2023-03-05	JP	Processing
2023-03-06	JP	Shipped
2023-03-05	US	Canceled
2023-03-02	US	Shipped
2023-03-05	US	Shipped
2023-03-04	US	Shipped

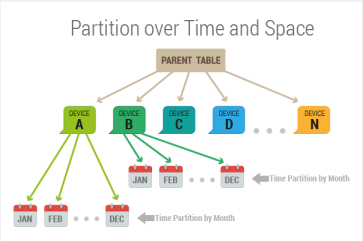
Table Clustered by Country and Status

Image from the article Google BigQuery Data Structure

Typical Use Cases

- *Time-series data* - monthly, weekly, daily, hourly partitions
- *Multi-tenant applications* - separate partitions for each tenant
- *Quick access to specific data* - range or list partitioning
- *Data collected from multiple sources* - hash partitioning
- *Data Warehouse & Data Lakehouse* - multiple dimensions
- *Data archiving & Data Retention Policies (GDPR)* - time-based partitioning





Images from the article Scaling IoT Time-Series Data in Postgres

credativ

Worst Case Scenarios

Microservices Logs Ingestion

- Production logs from thousands of microservices in single table
- Daily ingestion exceeded 1 TB with unpredictable spikes
- Multiple indexes required for analytics and reporting queries
- Retention policy kept recent data for several days
- Deletion of old data used time and complex filtering criteria
- High write and delete rates caused huge table and index bloat

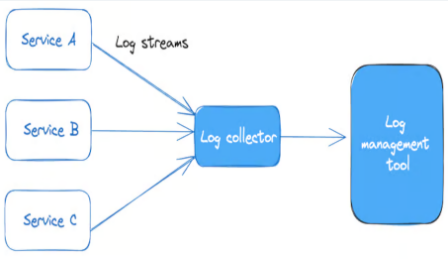


Image from the article Logging in Microservices

Real Life Worst Case Example

- PostgreSQL splits large tables into 1 GB segments
- 1 TB table creates 1000+ segment files
- Multi-GB indexes span hundreds of additional files
- Traffic spikes cause unpredictable data volume surges
- Peak days see extreme spikes in insert rates
- Large deletes may run for hours
- Autovacuum must reclaim space in table and indexes
- Bloat severely degrades query and write speed

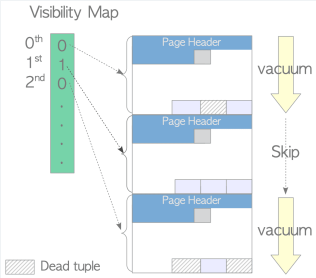


Image from the article Vacuum in PostgreSQL

Blaming PostgreSQL: A Common Pitfall

credativ

- Performance issues are often blamed on PostgreSQL itself
- Technical debt accumulates when proper design is postponed
- Production systems rarely allow downtime for major changes
- Feature development takes priority over infrastructure fixes
- Teams hesitate without clear partitioning strategy guidance
- ”Why doesn’t PostgreSQL handle large tables automatically?”
- NoSQL databases are often seen as magical scaling solutions
- Reality: No database can compensate for poor design choices

credativ

Automatic Partitioning

Magic Comes with a Price

- Some commercial databases offer automatic partitioning features
- Snowflake organizes tables into internal micro-partitions
- Micro-partitioning is fully managed and not user-configurable

- Oracle Autonomous Database provides automatic partitioning
- Skips tables with manual partitioning and analyses others
- Automatic analysis may consume significant system resources

- Both solutions are proprietary and involve high licensing costs

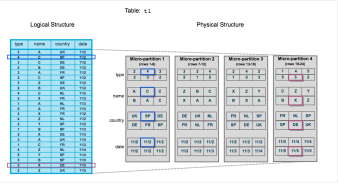


Image from the article Snowflake documentation: Micro-partitions & Data Clustering

Open Source Solutions

credativ

- Open source databases primarily support manual partitioning
- Semi-automatic tools exist, built on manual partitioning foundations
- Users define initial setup and partitioning strategy
- Tools automate partition creation and routine maintenance tasks
- PostgreSQL: *pg_partman* for general partition management
- Time-series data: *TimescaleDB* extension provides automation

How to Implement Partitioning Later?

- Retrofitting native partitioning requires careful planning
- Existing table must attach as a partition to new parent
- May require data migration or complete table rewrites
- Table inheritance offers simpler retrofitting approach
- Inheritance useful when partitioning strategy unclear
- Create new partitions for future data with routing rules
- Practical examples will be demonstrated in other talk

- Crunchy Bridge Guide: Partitioning Existing Tables in PostgreSQL

```
BEGIN
    ALTER TABLE probes_measurements
        RENAME TO probes_measurements_old_data;

    CREATE TABLE probes_measurements (
        id SERIAL PRIMARY KEY,
        -- all other columns ....
        inserted_at TIMESTAMP WITH TIME ZONE
            DEFAULT clock_timestamp()
    ) PARTITION BY RANGE
        (measurement_timestamp);

    ALTER TABLE probes_measurements
        ATTACH PARTITION probes_measurements_old_data
        FOR VALUES FROM (MINVALUE) TO ('2024-11-13');

    CREATE TABLE probes_measurements_20241113
        PARTITION OF probes_measurements
        FOR VALUES FROM ('2024-11-13') TO ('2024-11-14');
END;
```

credativ

PostgreSQL Partitioning Overview

PostgreSQL Partitioning Overview

- PostgreSQL supports table inheritance and declarative partitioning
- Table inheritance is legacy, not recommended for new applications
- It is more flexible, but requires manual management
- Declarative partitioning automates maintenance operations
- Each method has specific use cases and operational trade-offs
- Partitioning behavior is influenced by configuration settings
- Performance degrades significantly with excessive partitions

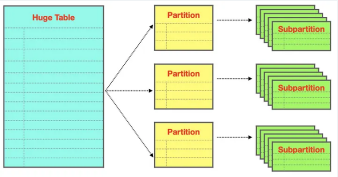
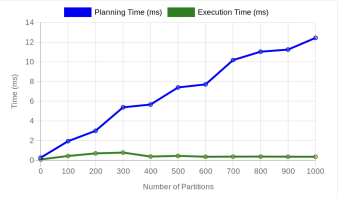


Image from the article Partitioned and Inherited Tables in PostgreSQL

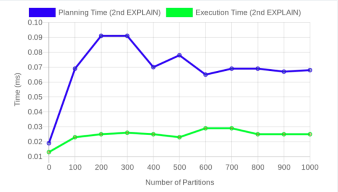
Partitioning: Scalability and Planning Overheads

- Performance can degrade with hundreds/thousands of partitions
- Query planning time increases significantly
- Planning can exceed execution time for complex queries
- Many partitions may cause session memory exhaustion
- Each partition adds metadata overhead in the planner's cache

Cold cache



Warm cache



Images from the article Postgres experiment: number of partitions vs. planning time

Efficiency of Indexes in Partitioned Tables

- Maintenance operations are more efficient on smaller partition indexes
- GIN indexes on large, unpartitioned tables badly degrade performance
- Indexes on partitions are smaller, enabling faster index scans
- Partitioning can yield 5x–10x faster queries in practical benchmarks

- Benchmarking PostgreSQL: The Hidden Cost of Over-Indexing
- How Partial and Covering Indexes Affect Update Performance in PostgreSQL

credativ

PostgreSQL Native Partitioning

PostgreSQL Native Partitioning

- Parent table is defined with the `PARTITION BY` clause
- Partition key columns must be included in the PRIMARY KEY
- Parent table is virtual - contains no data, only metadata
- Specifies columns and constraints for all partitions
- Child partitions inherit all columns and constraints from parent
- Only leaf partitions on the lowest level physically store data
- Each partitioning level can define a DEFAULT partition
- DML operations are automatically routed to the correct partition

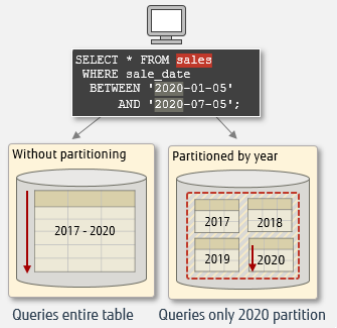


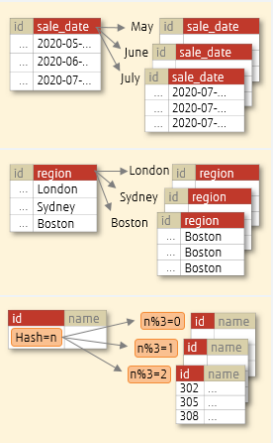
Image from the article
PostgreSQL Insider - Partitioning Overview

Partitioning Methods

- **Range partitioning**:
  - Partitions data into non-overlapping intervals
  - Ideal for time series, numeric ranges, or sequential data
  - Each partition covers a range: lower bound inclusive, upper bound exclusive
  - Example: 0 to 10 means: $0 \leq x < 10$

- **List partitioning**:
  - Assigns explicit values to each partition
  - Suitable for categorical data, such as ENUMs or specific IDs

- **Hash partitioning**:
  - Distributes rows evenly using a hash function
  - Each partition defined by modulus and remainder of the hash value
  - Useful for balancing data when no natural range or list exists



Images from the article
PostgreSQL Insider - Partitioning Overview

Example of Range Partitioning

```
CREATE TABLE orders_range (
    order_id SERIAL,
    customer_id INT NOT NULL,
    order_date DATE NOT NULL,
    status TEXT NOT NULL
) PARTITION BY RANGE (order_date);

CREATE TABLE orders_range_2022 PARTITION OF orders_range
    FOR VALUES FROM ('2022-01-01') TO ('2023-01-01');

CREATE TABLE orders_range_2023 PARTITION OF orders_range
    FOR VALUES FROM ('2023-01-01') TO ('2024-01-01');

CREATE TABLE orders_range_2024 PARTITION OF orders_range
    FOR VALUES FROM ('2024-01-01') TO ('2025-01-01');

CREATE TABLE orders_range_default PARTITION OF orders_range DEFAULT;
```

Example of List Partitioning

```
CREATE TABLE orders_list (
    order_id SERIAL,
    customer_id INT NOT NULL,
    order_date DATE NOT NULL,
    status TEXT NOT NULL
) PARTITION BY LIST (customer_id);

CREATE TABLE orders_list_customer_1 PARTITION OF orders_list
    FOR VALUES IN (1);

CREATE TABLE orders_list_customer_2 PARTITION OF orders_list
    FOR VALUES IN (2);

CREATE TABLE orders_list_customer_3 PARTITION OF orders_list
    FOR VALUES IN (3);

CREATE TABLE orders_list_default PARTITION OF orders_list DEFAULT;
```

Example of Hash Partitioning

```
CREATE TABLE orders_hash (
    order_id SERIAL,
    customer_id INT NOT NULL,
    order_date DATE NOT NULL,
    status TEXT NOT NULL
) PARTITION BY HASH (customer_id);

CREATE TABLE orders_hash_customer_1 PARTITION OF orders_hash
    FOR VALUES WITH (MODULUS 4, REMAINDER 0);

CREATE TABLE orders_hash_customer_2 PARTITION OF orders_hash
    FOR VALUES WITH (MODULUS 4, REMAINDER 1);

CREATE TABLE orders_hash_customer_3 PARTITION OF orders_hash
    FOR VALUES WITH (MODULUS 4, REMAINDER 2);

CREATE TABLE orders_hash_customer_4 PARTITION OF orders_hash
    FOR VALUES WITH (MODULUS 4, REMAINDER 3);
```

PostgreSQL Native Partitioning

credativ

- Partitions can be attached or detached without data movement
- Each partition can reside on a different tablespace for storage optimization
- Query planner prunes partitions using partition key predicates
- Partition-wise joins and aggregates enable parallel processing
- SPLIT PARTITION and MERGE PARTITION are not yet supported natively

```
ALTER TABLE orders_range
    ATTACH PARTITION orders_range_2021
    FOR VALUES FROM ('2021-01-01') TO ('2022-01-01');

ALTER TABLE orders_range
    DETACH PARTITION orders_range_2019;
```

Columns, Constraints, Indexes

credativ

- Columns & Constraints
  - Columns or constraints added to parent are propagated to all partitions
  - Dropping a column or constraint from parent removes it from partitions

- Indexes
  - Indexes created on parent are automatically created on all partitions
  - New partitions inherit all parent indexes at creation time
  - Attaching a table as partition creates any missing required indexes
  - Dropping an index on parent removes it from all partitions
  - Indexes created via parent cannot be dropped individually on partitions
  - Indexes created directly on a partition exist only on that partition

Useful Queries - All partitioned parent tables

```
SELECT
    n.nspname                       AS schema_name,
    c.relname                       AS table_name,
    pt.partstrat                    AS strategy,      -- 'r' = range, 'l' = list, 'h' = ↩
 hash
    pt.partnatts                    AS key_cols,
    pg_get_partkeydef(c.oid)        AS partition_key
FROM pg_partitioned_table AS pt
JOIN pg_class      AS c ON c.oid = pt.partrelid
JOIN pg_namespace  AS n ON n.oid = c.relnamespace
WHERE n.nspname NOT IN ('pg_catalog', 'information_schema')
ORDER BY 1, 2;

 schema_name |  table_name  | strategy | key_cols |   partition_key
-------------+--------------+----------+----------+--------------------
 public      | orders_hash  | h        |        1 | HASH (customer_id)
 public      | orders_list  | l        |        1 | LIST (customer_id)
 public      | orders_range | r        |        1 | RANGE (order_date)
```

Query - All children and their parent tables

credativ

```
SELECT
    pn.nspname                      AS parent_schema,
    p.relname                       AS parent_table,
    cn.nspname                      AS partition_schema,
    c.relname                       AS partition_table
FROM pg_inherits AS i
JOIN pg_class     AS p  ON p.oid  = i.inhparent
JOIN pg_namespace AS pn ON pn.oid = p.relnamespace
JOIN pg_class     AS c  ON c.oid  = i.inhrelid
JOIN pg_namespace AS cn ON cn.oid = c.relnamespace
WHERE p.relkind = 'p'               -- parent is a partitioned table
ORDER BY parent_schema, parent_table, partition_schema, partition_table;
```

```
 parent_schema | parent_table | partition_schema |    partition_table
---------------+--------------+------------------+-----------------------
 public        | orders_hash  | public           | orders_hash_customer_1
 public        | orders_hash  | public           | orders_hash_customer_2
 public        | orders_hash  | public           | orders_hash_customer_3
 public        | orders_hash  | public           | orders_hash_customer_4
 public        | orders_list  | public           | orders_list_customer_1
 public        | orders_list  | public           | orders_list_customer_2
 public        | orders_list  | public           | orders_list_customer_3
 public        | orders_list  | public           | orders_list_default
 public        | orders_range | public           | orders_range_2022
 public        | orders_range | public           | orders_range_2023
 public        | orders_range | public           | orders_range_2024
 public        | orders_range | public           | orders_range_default
```

Query - Inspect the partition tree

```
SELECT
       t.relid::regclass      AS rel_name,
       t.parentrelid::regclass AS parent_name,
       t.level,
       t.isleaf
FROM pg_partition_tree('public.orders_list'::regclass) AS t
ORDER BY t.level, rel_name;

        rel_name         | parent_name | level | isleaf
-------------------------+-------------+-------+--------
 orders_list             |             |     0 | f
 orders_list_customer_1  | orders_list |     1 | t
 orders_list_customer_2  | orders_list |     1 | t
 orders_list_customer_3  | orders_list |     1 | t
 orders_list_default     | orders_list |     1 | t
```

Query - Bounds for all partitions of all parent tables

credativ

```
SELECT
    child.oid::regclass                       AS partition_name,
    pg_get_expr(child.relpartbound, child.oid) AS partition_bound
FROM pg_inherits AS i
JOIN pg_class     AS parent ON parent.oid = i.inhparent
JOIN pg_class     AS child  ON child.oid  = i.inhrelid
WHERE parent.relkind = 'p'
ORDER BY partition_name;
```

```
     partition_name     |                 partition_bound
------------------------+-------------------------------------------------
 orders_range_2022      | FOR VALUES FROM ('2022-01-01') TO ('2023-01-01')
 orders_range_2023      | FOR VALUES FROM ('2023-01-01') TO ('2024-01-01')
 orders_range_2024      | FOR VALUES FROM ('2024-01-01') TO ('2025-01-01')
 orders_list_customer_1 | FOR VALUES IN (1)
 orders_list_customer_2 | FOR VALUES IN (2)
 orders_list_customer_3 | FOR VALUES IN (3)
 orders_hash_customer_1 | FOR VALUES WITH (modulus 4, remainder 0)
 orders_hash_customer_2 | FOR VALUES WITH (modulus 4, remainder 1)
 orders_hash_customer_3 | FOR VALUES WITH (modulus 4, remainder 2)
 orders_hash_customer_4 | FOR VALUES WITH (modulus 4, remainder 3)
 orders_list_default    | DEFAULT
 orders_range_default   | DEFAULT
```

Query - List only DEFAULT partitions

```
SELECT
    n.nspname               AS parent_schema,
    c.relname               AS parent_table,
    dp.oid::regclass        AS default_partition
FROM pg_partitioned_table AS pt
JOIN pg_class       AS c  ON c.oid = pt.partrelid
JOIN pg_namespace   AS n  ON n.oid = c.relnamespace
JOIN pg_class       AS dp ON dp.oid = pt.partdefid
WHERE pt.partdefid <> 0;

 parent_schema | parent_table |  default_partition
---------------+--------------+----------------------
 public        | orders_range | orders_range_default
 public        | orders_list  | orders_list_default
```

Query - List partitions and their sizes

credativ

```
SELECT
    t.relid::regclass                       AS partition_name,
    t.level,
    pg_size_pretty(pg_relation_size(t.relid))          AS heap_size,
    pg_size_pretty(pg_total_relation_size(t.relid))    AS total_size_incl_indexes
FROM pg_partition_tree('public.orders_list'::regclass) AS t
WHERE t.isleaf
ORDER BY t.level, partition_name;

     partition_name     | level | heap_size | total_size_incl_indexes
------------------------+-------+-----------+-------------------------
 orders_list_customer_1 |     1 | 0 bytes   | 8192 bytes
 orders_list_customer_2 |     1 | 0 bytes   | 8192 bytes
 orders_list_customer_3 |     1 | 0 bytes   | 8192 bytes
 orders_list_default    |     1 | 0 bytes   | 8192 bytes
```

credativ

PostgreSQL Partitioning Settings

PostgreSQL Partitioning Settings

credativ

- Assume shared_buffers, work_mem, maintenance_work_mem, effective_cache_size are tuned
- Parameter **enable_partition_pruning**
  - Default value = ON
  - Skips scanning partitions that cannot match query predicates
  - For table inheritance, check constraints are required for pruning
- Parameter **constraint_exclusion**
  - Default value = 'partition'
  - Controls exclusion of partitions and tables based on constraints
  - For declarative partitioning, prefer enable_partition_pruning

PostgreSQL Partitioning Settings – Related Parameters

credativ

- Additional parameters may impact partitioning performance and scalability

- Parameter **max_locks_per_transaction** - default 64
- Total locks = **max_locks_per_transaction** × (**max_connections** + **max_prepared_transactions**)
- High partition counts can exhaust lock slots, causing ”out of shared memory” errors

- Parameter **max_pred_locks_per_relation** - default 64
- Sets the maximum number of predicate locks per relation
- Insufficient slots may also trigger ”out of shared memory” errors

PostgreSQL Partitioning Settings – Related Parameters

- Parameter **max_files_per_process**
  - Default value = 1000
  - Sets the maximum number of files a backend process can open simultaneously
  - High partition/table counts may require increasing this value

- Parameter **default_statistics_target**
  - Default value = 100
  - Determines the number of rows sampled for table statistics
  - Formula: 300 × **default_statistics_target** rows sampled per column
  - The multiplier 300 is based on research for histogram accuracy (Ioannidis, 1998)
  - Higher values can improve query planning for partitioned tables
  - Larger targets increase ANALYZE time and statistics size

PostgreSQL Partitioning Settings

credativ

- Parameter **enable_partitionwise_aggregate**
  - Default value = OFF
  - Enables aggregation to run independently on each partition
  - Improves performance for GROUP BY and aggregate queries on partitions
- Parameter **enable_partitionwise_join**
  - Default value = OFF
  - Allows joins to be executed separately for matching partitions
  - Join condition must include all partitioning keys for partitionwise join
  - Reduces join costs for large partitioned tables with aligned partitioning
- Both settings may increase CPU and memory usage significantly
- Assess workload and resource impact before enabling in production

PostgreSQL Partitioning Settings - enable_partitionwise_aggregate

credativ

```
-- Create a partitioned table
CREATE TABLE sales (
    sale_id serial NOT NULL,
    sale_date date NOT NULL,
    amount numeric,
    PRIMARY KEY (sale_id, sale_date)
) PARTITION BY RANGE (sale_date);

-- Create partitions
CREATE TABLE sales_2023 PARTITION OF sales FOR VALUES FROM ('2023-01-01') TO ('2024-01-01');
CREATE TABLE sales_2024 PARTITION OF sales FOR VALUES FROM ('2024-01-01') TO ('2025-01-01');

-- Insert data into partitions
INSERT INTO sales_2023 (sale_date, amount) VALUES ('2023-05-01', 100);
INSERT INTO sales_2023 (sale_date, amount) VALUES ('2023-06-01', 150);
INSERT INTO sales_2024 (sale_date, amount) VALUES ('2024-02-01', 200);
INSERT INTO sales_2024 (sale_date, amount) VALUES ('2024-03-01', 250);
```

PostgreSQL Partitioning Settings - enable_partitionwise_aggregate

credativ

```
test=# set enable_partitionwise_aggregate = off;
SET
test=# explain SELECT sum(amount) AS total_sales FROM sales WHERE sale_date BETWEEN '2023-01-01' AND '2024-12-31';
                                        QUERY PLAN
-------------------------------------------------------------------------------------------
    Aggregate  (cost=56.09..56.10 rows=1 width=32)
    ->  Append  (cost=0.00..56.06 rows=12 width=32)
          ->  Seq Scan on sales_2023 sales_1  (cost=0.00..28.00 rows=6 width=32)
              Filter: ((sale_date >= '2023-01-01'::date) AND (sale_date <= '2024-12-31'::date))
          ->  Seq Scan on sales_2024 sales_2  (cost=0.00..28.00 rows=6 width=32)
              Filter: ((sale_date >= '2023-01-01'::date) AND (sale_date <= '2024-12-31'::date))



test=# set enable_partitionwise_aggregate = on;
SET
test=# explain SELECT sum(amount) AS total_sales FROM sales WHERE sale_date BETWEEN '2023-01-01' AND '2024-12-31';
                                              QUERY PLAN
-----------------------------------------------------------------------------------------------------
    Finalize Aggregate  (cost=56.08..56.09 rows=1 width=32)
    ->  Append  (cost=28.02..56.07 rows=2 width=32)
          ->  Partial Aggregate  (cost=28.02..28.03 rows=1 width=32)
              ->  Seq Scan on sales_2023 sales  (cost=0.00..28.00 rows=6 width=32)
                  Filter: ((sale_date >= '2023-01-01'::date) AND (sale_date <= '2024-12-31'::date))
          ->  Partial Aggregate  (cost=28.02..28.03 rows=1 width=32)
              ->  Seq Scan on sales_2024 sales_1  (cost=0.00..28.00 rows=6 width=32)
                  Filter: ((sale_date >= '2023-01-01'::date) AND (sale_date <= '2024-12-31'::date))
```

PostgreSQL Partitioning Settings - enable_partitionwise_join

credativ

```sql
-- Create a partitioned table for sales
CREATE TABLE sales (
    sale_id serial NOT NULL,
    sale_date date NOT NULL,
    amount numeric,
    PRIMARY KEY (sale_id, sale_date)
) PARTITION BY RANGE (sale_date);

-- Create partitions for sales
CREATE TABLE sales_2023 PARTITION OF sales FOR VALUES FROM ('2023-01-01') TO ('2024-01-01');
CREATE TABLE sales_2024 PARTITION OF sales FOR VALUES FROM ('2024-01-01') TO ('2025-01-01');

-- Create a partitioned table for customers
CREATE TABLE customers (
    customer_id serial NOT NULL,
    registration_date date NOT NULL,
    name text,
    PRIMARY KEY (customer_id, registration_date)
) PARTITION BY RANGE (registration_date);

-- Create partitions for customers
CREATE TABLE customers_2023 PARTITION OF customers FOR VALUES FROM ('2023-01-01') TO ('2024-01-01');
CREATE TABLE customers_2024 PARTITION OF customers FOR VALUES FROM ('2024-01-01') TO ('2025-01-01');

-- Insert data into sales partitions
INSERT INTO sales_2023 (sale_date, amount) VALUES ('2023-05-01', 100), ('2023-06-01', 150);
INSERT INTO sales_2024 (sale_date, amount) VALUES ('2024-02-01', 200), ('2024-03-01', 250);

-- Insert data into customers partitions
INSERT INTO customers_2023 (registration_date, name) VALUES ('2023-05-01', 'Alice'), ('2023-06-01', 'Bob');
INSERT INTO customers_2024 (registration_date, name) VALUES ('2024-02-01', 'Charlie'), ('2024-03-01', 'Dave');
```

PostgreSQL Partitioning Settings - enable_partitionwise_join

credativ

```
test=# set enable_partitionwise_join = off;

test=# explain SELECT s.sale_date, s.amount, c.name FROM sales s
JOIN customers c ON s.sale_date = c.registration_date
WHERE s.sale_date BETWEEN '2023-01-01' AND '2024-12-31';

                                                  QUERY PLAN
---------------------------------------------------------------------------------------------------------------
    Hash Join  (cost=56.21..122.65 rows=144 width=68)
    Hash Cond: (c.registration_date = s.sale_date)
    ->  Append  (cost=0.00..56.00 rows=2400 width=36)
          ->  Seq Scan on customers_2023 c_1  (cost=0.00..22.00 rows=1200 width=36)
          ->  Seq Scan on customers_2024 c_2  (cost=0.00..22.00 rows=1200 width=36)
    ->  Hash  (cost=56.06..56.06 rows=12 width=36)
          ->  Append  (cost=0.00..56.06 rows=12 width=36)
                ->  Seq Scan on sales_2023 s_1  (cost=0.00..28.00 rows=6 width=36)
                      Filter: ((sale_date >= '2023-01-01'::date) AND (sale_date <= '2024-12-31'::date))
                ->  Seq Scan on sales_2024 s_2  (cost=0.00..28.00 rows=6 width=36)
                      Filter: ((sale_date >= '2023-01-01'::date) AND (sale_date <= '2024-12-31'::date))
```

PostgreSQL Partitioning Settings - enable_partitionwise_join

credativ

```
test=# set enable_partitionwise_join = on;

test=# explain SELECT s.sale_date, s.amount, c.name FROM sales s
JOIN customers c ON s.sale_date = c.registration_date
WHERE s.sale_date BETWEEN '2023-01-01' AND '2024-12-31';

                                                    QUERY PLAN
-------------------------------------------------------------------------------------------------------------------
    Append  (cost=28.07..110.23 rows=72 width=68)
    ->  Hash Join  (cost=28.07..54.94 rows=36 width=68)
            Hash Cond: (c_1.registration_date = s_1.sale_date)
            ->  Seq Scan on customers_2023 c_1  (cost=0.00..22.00 rows=1200 width=36)
            ->  Hash  (cost=28.00..28.00 rows=6 width=36)
                ->  Seq Scan on sales_2023 s_1  (cost=0.00..28.00 rows=6 width=36)
                        Filter: ((sale_date >= '2023-01-01'::date) AND (sale_date <= '2024-12-31'::date))
    ->  Hash Join  (cost=28.07..54.94 rows=36 width=68)
            Hash Cond: (c_2.registration_date = s_2.sale_date)
            ->  Seq Scan on customers_2024 c_2  (cost=0.00..22.00 rows=1200 width=36)
            ->  Hash  (cost=28.00..28.00 rows=6 width=36)
                ->  Seq Scan on sales_2024 s_2  (cost=0.00..28.00 rows=6 width=36)
                        Filter: ((sale_date >= '2023-01-01'::date) AND (sale_date <= '2024-12-31'::date))
```

credativ

PostgreSQL Extensions for Partitioning

PostgreSQL Extensions for Partitioning

credativ

- *pg_partman* - Advanced partition management, supports time and integer ranges
- *TimescaleDB* - Time-series extension, automatic partitioning via hypertables

- *pg_repack* - Online table and index reorganization, enables clustering
- *pg_squeeze* - Automates bloat removal, supports online table clustering
- *Citus* - Distributed extension, enables sharding and multi-node partitioning

pg_repack / pg_squeeze

credativ

- Does not support partitioning
- But can be life saver in case of badly bloated tables/indexes
- Extension for online table and index reorganization
- Removes bloat from tables and indexes without locks
- Supports clustering tables based on an index
- Creates a new copy of the table with desired order
- Swaps the new table in place with minimal locking
- Requires minimal downtime during reorganization
- pg_squeeze is designed to replace pg_repack

pg_partman

credativ

- Automates creation and management of time- and number-based partitions
- Supports native declarative partitioning; inheritance deprecated
- Includes background worker for automatic partition maintenance
- Configure with `shared_preload_libraries = 'pg_partman_bgw'`
- Eliminates need for external schedulers like cron
- Requires PostgreSQL 14 or newer
- Recommended by AWS RDS for scalable partition management

TimescaleDB

credativ

- Extension for scalable time-series data on PostgreSQL
- Time-series tables are managed as ”hypertables”
- Hypertables are partitioned into chunks automatically
- Supports partitioning by time, or by time and space (e.g., device_id)
- Enables continuous aggregations via specialized materialized views
- Optimized for high ingest rates and analytical queries
- Provides native compression for historical data

TIMESCALE

Thank you for your attention!

All my slides

Recorded talks