The art of data retention in PostgreSQL

data egret

Your remote PostgreSQL DBA team

Stefan FERCOT

contact@dataegret.com

SECURING YOUR POSTGRESQL DATABASE AVAILABILITY AND HIGH PERFORMANCE

- **Migration**
- Performance **audit**
- **Cloud Cost** management
- **Backup & restore**
- **Architecture** review
- **DataOps/CDC** projects
- **Consulting** for data science and analytics teams
- **PostgreSQL Courses** for DBA and Developers

Contributing sponsor to PostgreSQL

Open Alliance
For PostgreSQL Education

OSB Open Source Business ALLIANCE
Bundesverband Für digitale Souveränität e.V.
MITGLIED

EXPERTISE

Senior DBA with 10+ years of experience in PostgreSQL administration.

DEVELOPMENT

PostgreSQL Contributors involved in new PostgreSQL feature and extention development.

CUSTOMISATION

Flexible approach and dedicated team focused on success of your project.

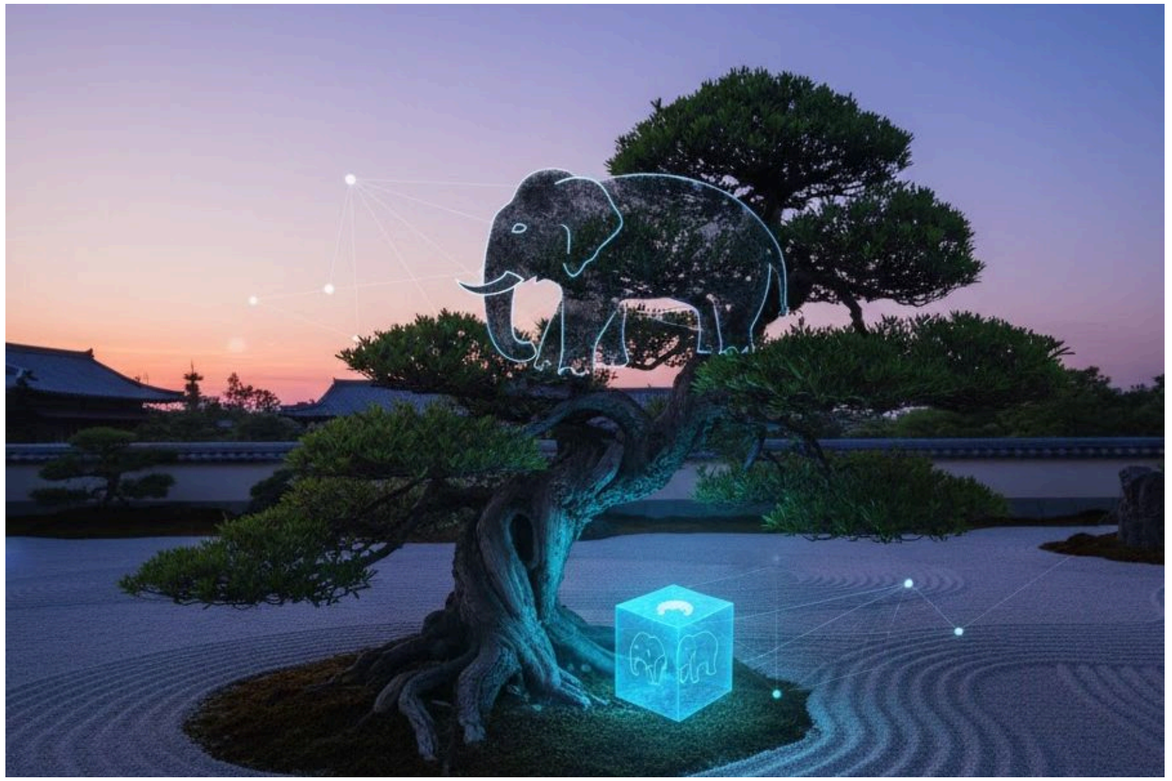
COMMUNITY

Recognised as Significant Contributing Sponsor to PostgreSQL.

Stefan Fercot

- PostgreSQL Expert **@Data Egret**
  - PostgreSQL consulting, support, and training
- pgBackRest contributor and advocate
- Recognized PostgreSQL significant contributor
- Also known as pgstef
- https://pgstef.github.io

The art of data retention in PostgreSQL



Balancing performance, cost, and discipline in growing databases

Every database starts small

- A few tables
- A few thousand rows
- A mix of tests and early production traffic

The calm before the storm

- Everything still fitted in memory
- Everything was fast
- Nobody was worried

Six months later

The hidden cost of big tables

- More data to scan, even with indexes
- VACUUM and autovacuum become heavier
- Cache efficiency drops
- Backups and replication slow down
- Higher risk of downtime 🔥

How can we save disk space when data grows?

- Data always grows faster than we expect
- Can PostgreSQL compress our way out of this?

Compression: the first idea

- PostgreSQL does have a form of compression: **TOAST**
- But TOAST is not a traditional compression system
  - TOAST = *The Oversized Attribute Storage Technique*
- It exists to handle **large values that do not fit in a page**

How PostgreSQL stores data

- PostgreSQL stores data in fixed-size **pages** (8 KB by default)
- If a row value is too large to fit in that page, TOAST comes in

What TOAST does

- Compresses large values using an internal algorithm
- If still too big, moves them to a **TOAST table** (`pg_toast_XXX`)
- The main table stores only a **pointer** instead of the full data

TOAST and compression

- Default: `pglz`
- Recommended: `lz4`
  - Faster in most cases
  - Similar compression ratio
- How to enable `lz4`:
  - `default_toast_compression = 'lz4';`
  - or `ALTER TABLE t1 ALTER COLUMN c2 SET COMPRESSION lz4;`

Example

```
CREATE TABLE compress_lz4 (
    id         serial PRIMARY KEY,
    details    text COMPRESSION lz4
);

INSERT INTO compress_lz4 (details)
SELECT
    repeat(md5(random()::text), 5000)   -- ~160 KB strings
FROM generate_series(1, 1_000_000) i;
```

Table	Rows	Size
compress_lz4	1,000,000	710 MB
compress_pglz	1,000,000	1.9 GB

Why compression alone is not enough

- Only helps for big text, JSONB, or bytea columns
- Does nothing for regular numeric or short text data
- Does not control table growth or vacuum overhead
- You still need to manage what stays and what goes

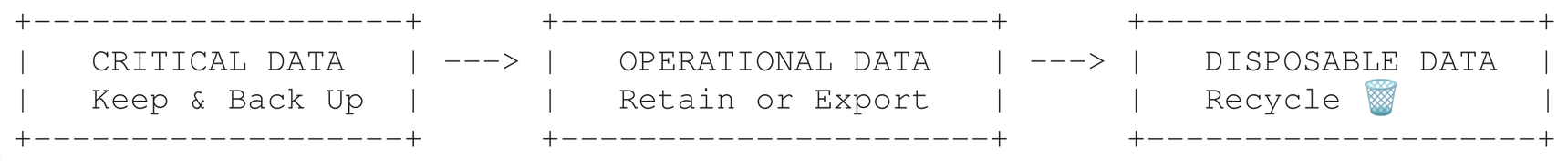
> *Compression saves bytes, not discipline.*

From chaos to structure

The real path: data retention

1. The problem: data keeps growing
2. Principles: classify data and plan retention
3. Techniques: partitioning, archiving
4. The way of the database: discipline and foresight

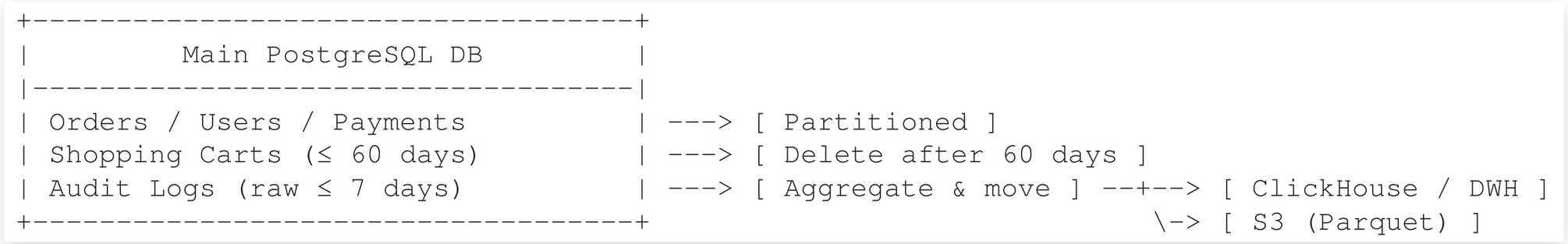
Not all data is equal



Defining data lifecycles

- Decide how long to keep each data type
- Define where it goes when it becomes old
- Set clear rules for safe deletion
- Keep only what is useful or required

Case study: an e-commerce platform



Partitioning in PostgreSQL

- Helps with:
  - Faster queries on recent or relevant data
  - Easier clean-up
    - Drop old partitions instead of running large DELETEs
  - Reduced bloat and lower VACUUM overhead

Declarative partitioning

- **Partitioned table**
  - Defines only the structure
  - Indexes and constraints are automatically propagated to partitions
- **Partitions**
  - Each partition is a regular table and can be queried directly
  - Supports default partitions
  - Sub-partitions are also supported
  - Partitions can be attached or detached

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

List partitioning

- The table is divided into partitions by listing specific key values
- Each partition contains rows matching those values
- The partition key must be a single column

Example

```
CREATE TABLE customers (
  id serial,
  country text,
  name text
) PARTITION BY LIST (country);

CREATE TABLE customers_eu PARTITION OF customers FOR VALUES IN ('FR', 'DE', 'ES');
CREATE TABLE customers_us PARTITION OF customers FOR VALUES IN ('US', 'CA');
CREATE TABLE customers_other PARTITION OF customers DEFAULT;
```

Hash partitioning

- Distributes data evenly across partitions
- Uses a **modulus** and **remainder** to decide where each row goes
- Each partition holds rows where `hash(key) % modulus = remainder`

Example

```
CREATE TABLE customers (
  id serial,
  country text,
  name text
) PARTITION BY HASH (country);

CREATE TABLE customers_p0 PARTITION OF customers FOR VALUES WITH (modulus 3, remainder 0);
CREATE TABLE customers_p1 PARTITION OF customers FOR VALUES WITH (modulus 3, remainder 1);
CREATE TABLE customers_p2 PARTITION OF customers FOR VALUES WITH (modulus 3, remainder 2);
```

Range partitioning

- The table is divided into ranges based on one or more key columns
- Each range must be unique and cannot overlap with another
- Range bounds are inclusive at the lower end and exclusive at the upper end
- Use `MINVALUE` or `MAXVALUE` for open-ended ranges

Example

```sql
-- Create the parent table and define range partitioning on order_date
CREATE TABLE orders (
  order_id      bigint GENERATED BY DEFAULT AS IDENTITY,
  customer_id   bigint NOT NULL,
  order_date    date NOT NULL,
  total_amount  numeric NOT NULL,
  PRIMARY KEY (order_date, order_id)
) PARTITION BY RANGE (order_date);

-- Create monthly partitions for Q1 2025
CREATE TABLE orders_2025_01 PARTITION OF orders
  FOR VALUES FROM ('2025-01-01') TO ('2025-02-01');

CREATE TABLE orders_2025_02 PARTITION OF orders
  FOR VALUES FROM ('2025-02-01') TO ('2025-03-01');

CREATE TABLE orders_2025_03 PARTITION OF orders
  FOR VALUES FROM ('2025-03-01') TO ('2025-04-01');

-- Create an index on the parent table (simple, not concurrent)
CREATE INDEX orders_customer_id_idx ON orders (customer_id);
```

The benefits of partitioning

- Faster queries
- Easier clean-up
- Better long-term performance

> *“Partitions follow your business logic: by month, by region, by customer, making data easier to manage and maintain.”*

But old data never leaves on its own

- Old partitions are:
  - Still on disk
  - Still vacuumed
  - Still backed up
  - Still costing money

Cleaning up old partitions

- Old partitions can be:
  - Exported
  - Detached
  - Dropped

```
-- Option 1: Export with COPY
COPY orders_2025_01 TO '/archive/orders_2025_01.csv' CSV;

-- Option 2: Export with pg_dump
pg_dump -Fc -Z 3 -t orders_2025_01 yourdb > orders_2025_01.sql

-- Detach or drop after export
ALTER TABLE orders DETACH PARTITION orders_2025_01 CONCURRENTLY;
DROP TABLE orders_2025_01;
```

What about less active but still accessible data

- Keep it accessible, just not on your fastest storage
- Common approaches:
  - Separate PostgreSQL **tablespace**
  - Separate PostgreSQL **instance**
  - **External object storage** (for example, S3 or Azure Blob)
  - **Data warehouse** (for example, ClickHouse or BigQuery)

Option A: another tablespace in the same cluster

- Move older partitions to slower or cheaper disks
- Queries still use the same tables and indexes
- Requires minimal application changes

```
-- Create a tablespace on cheaper storage
CREATE TABLESPACE orders_archive LOCATION '/mnt/archive/pg';

-- Move a partition to the archive tablespace
ALTER TABLE orders_2025_02 SET TABLESPACE orders_archive;
```

Understanding tablespaces

- Physical storage for database objects (not a logical concept)
- Represented by a normal directory outside **PGDATA**, linked symbolically
- Common use cases:
  - Distributing I/O load and data volume
  - Enforcing storage quotas via the filesystem
  - Isolating temporary or sorting operations

Option B: external archiving

FDW	Reads from	Example use case
file_fdw	CSV or TSV files	Logs and simple exports
postgres_fdw	Other PostgreSQL databases	Cross-database archiving or reporting
pg_parquet	Parquet with S3, GCS, or Azure	Object storage
pg_lake	Parquet, CSV, JSON, Iceberg	Analytics and AI workloads

```
CREATE FOREIGN TABLE hits ()
SERVER pg_lake options (path 'az://your_container_name/hits.parquet');

SELECT count(*) FROM hits;
```

Putting it together: a simple retention workflow

- **Keep**: data that is legally or operationally required
- **Move**: less active data to slower or cheaper storage
- **Archive**: historical data that should live outside PostgreSQL
- **Drop**: data that is no longer needed and safe to remove

Automating retention policies

- Define a clear retention period per table or dataset (for example, 12 months)
- Always archive before dropping data (export or offload)
- Monitor logs or alerts to track when partitions are removed
- Test automation in staging before running it in production

Tools for automation

Tool	Purpose
pg_partman	Automatically create and drop partitions based on time or ID
pg_cron / cron	Schedule SQL tasks and scripts
pg_timetable	PostgreSQL-native job scheduler with advanced workflows
COPY / pg_dump	Export data before dropping

Production pitfalls to avoid

Operation	Common issue
`ALTER TABLE ... SET COMPRESSION`	No effect on existing data unless the table is rewritten (e.g. with `CLUSTER`)
Rewriting large tables	CPU-intensive; run during off-hours
Index without `CONCURRENTLY`	Blocks reads and writes; can block traffic
Too many partitions	Slows query planning and autovacuum
Dropping large partitions	May cause locks or replication lag

The way of the database

Where to go from here:

1. **Understand your data**
2. **Partition** what grows continuously
3. **Set retention rules**: delete or archive
4. **Automate** lifecycle tasks
5. **Monitor** and adjust over time

Discipline is scalability

> *“Like the code of Bushido, discipline brings freedom.*
> *Freedom from chaos, slow queries, and sleepless nights.”*

Data retention is not about deletion; it is about **preserving what truly matters**.

Questions and feedback

contact@dataegret.com

Need some help with PostgreSQL? ;-)
*Get in touch with **Data Egret** to talk about PostgreSQL*
and, in particular, about training.

Bonus appendix

- Partitioning an existing table
- Indexing partitioned tables
- Creating and using tablespaces
- Automating retention with `pg_cron`

Partitioning an existing table

1. Create an empty partitioned copy
2. Add a trigger to sync new inserts, updates, and deletes
3. Backfill data in safe batches
4. Build indexes concurrently if needed
5. Switch over in a single transaction

Partitioning an existing table (Adyen approach)

1. Rename the original table as a *mammoth* partition
2. Create a new parent partitioned table with the original name
3. Add the *mammoth* table as the first child
4. Create empty replacement partitions
5. Backfill data into the new partitions in safe batches
6. Detach the *mammoth* and attach the new partitions
7. Validate constraints afterwards if needed

Adyen: Efficiently RePartitioning Large Tables in PostgreSQL

Indexing partitioned tables

- Index on the parent table:
  - Automatically applies to all partitions (existing and future)
  - Cannot be created concurrently, which can block reads and writes

```
CREATE INDEX ON orders (customer_id);
```

Indexing partitions manually

- Manual control with `ON ONLY`:
  - Skips automatic propagation
  - Allows per-partition indexes to be created concurrently

```
CREATE INDEX ON ONLY orders (customer_id);
CREATE INDEX CONCURRENTLY ON orders_2025_01 (customer_id);
ALTER INDEX orders_customer_id_idx
  ATTACH PARTITION orders_2025_01_customer_id_idx;
```

Example: indexing range partitions

```sql
-- Define the index on the parent "ONLY"
CREATE INDEX orders_customer_id_idx ON ONLY orders (customer_id);

-- Create the indexes on each existing partition CONCURRENTLY
CREATE INDEX CONCURRENTLY orders_2025_01_customer_id_idx ON orders_2025_01 (customer_id);
CREATE INDEX CONCURRENTLY orders_2025_02_customer_id_idx ON orders_2025_02 (customer_id);
CREATE INDEX CONCURRENTLY orders_2025_03_customer_id_idx ON orders_2025_03 (customer_id);

-- Attach the per-partition indexes to the parent index
ALTER INDEX orders_customer_id_idx ATTACH PARTITION orders_2025_01_customer_id_idx;
ALTER INDEX orders_customer_id_idx ATTACH PARTITION orders_2025_02_customer_id_idx;
ALTER INDEX orders_customer_id_idx ATTACH PARTITION orders_2025_03_customer_id_idx;

-- Tip: once the parent partitioned index exists,
-- any new partition you create will automatically receive a matching child index.
```

Creating and using tablespaces

```
-- Create a new tablespace
CREATE TABLESPACE ssd LOCATION '/mnt/ssd/pg';

-- Assign it to a database
CREATE DATABASE mydb TABLESPACE ssd;
ALTER DATABASE mydb SET default_tablespace TO ssd;

-- Assign it to a specific table
CREATE TABLE some_table (...) TABLESPACE ssd;

-- Move an existing table (requires an exclusive lock)
ALTER TABLE some_table SET TABLESPACE ssd;
```

Automating retention with `pg_cron`

- Automatically:
  - Detach old partitions (older than 12 months)
  - Export to `/archive`
  - Drop from the live database
- Scheduled with:

```
-- Schedule a nightly job at 03:00
SELECT cron.schedule(
  job_name   => 'detach_export_drop_old_partitions',
  schedule   => '0 3 * * *',
  command    => 'CALL detach_export_drop_old_partitions();'
);
```

Example: procedure to export and prune old partitions

```
CREATE OR REPLACE PROCEDURE detach_export_drop_old_partitions ()
LANGUAGE plpgsql
AS $$
DECLARE
  part RECORD;
BEGIN
  FOR part IN
    SELECT tablename FROM pg_tables
    WHERE schemaname = 'public' AND tablename LIKE 'orders_20%%'
    AND tablename < to_char(current_date - interval '12 months', '"orders_"YYYY_MM')
  LOOP
    RAISE LOG 'Retention: handling %', part.tablename;
    -- 1) Detach from the parent table
    EXECUTE format('ALTER TABLE orders DETACH PARTITION %I;', part.tablename);
    -- 2) Export the partition to CSV (server-side path)
    EXECUTE format('COPY %I TO ''/archive/%I.csv'' WITH CSV HEADER;', part.tablename, part.tablename);
    -- 3) Drop the detached table
    EXECUTE format('DROP TABLE IF EXISTS %I CASCADE;', part.tablename);
  END LOOP;
END;
$$;
```