Talk slides and markdown

- `github.com/sfc-gh-echristensen/bluebox`



What's New in

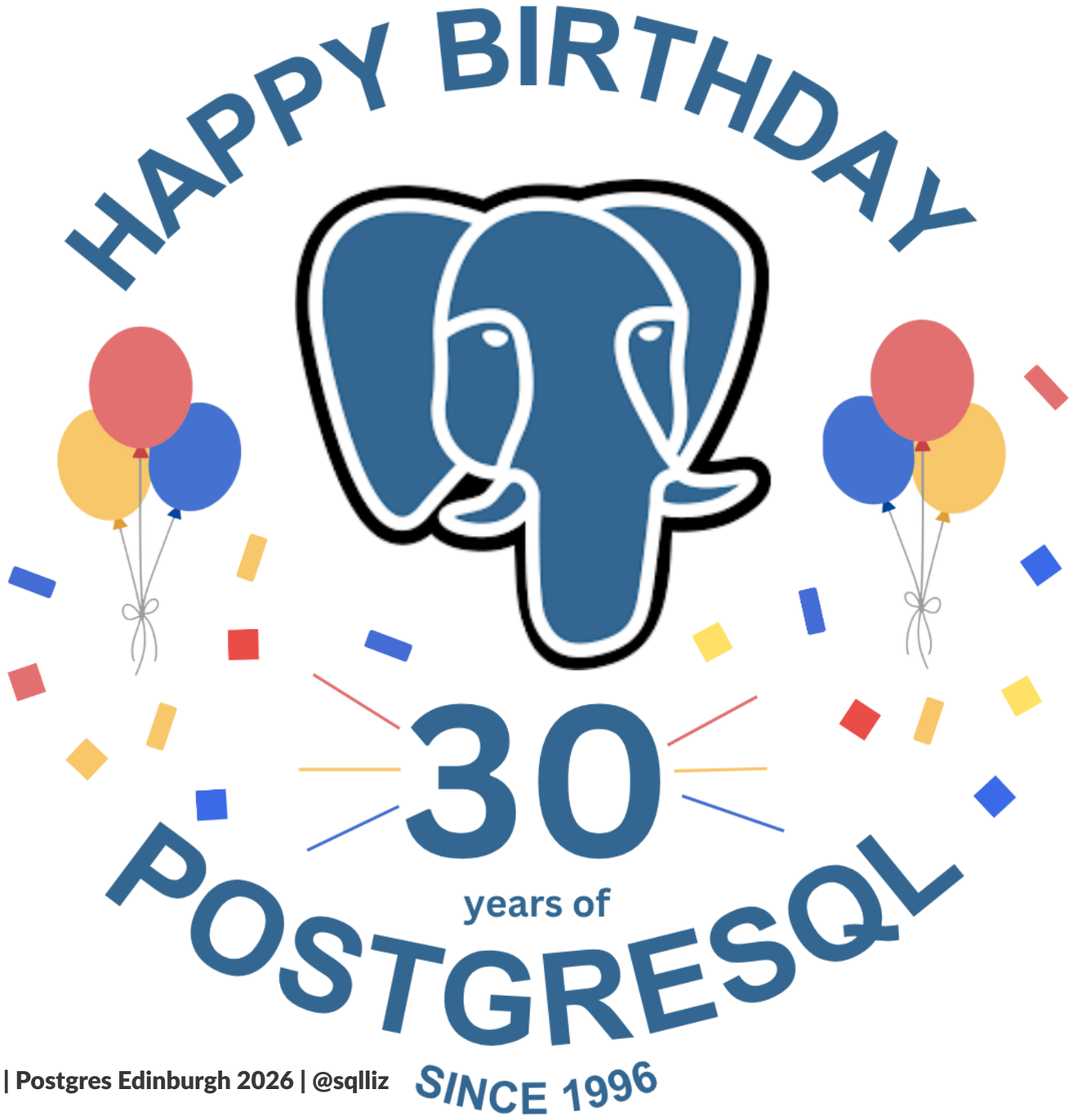
PostgreSQL 18 & 19

Edinburgh Postgres, Oct 2026

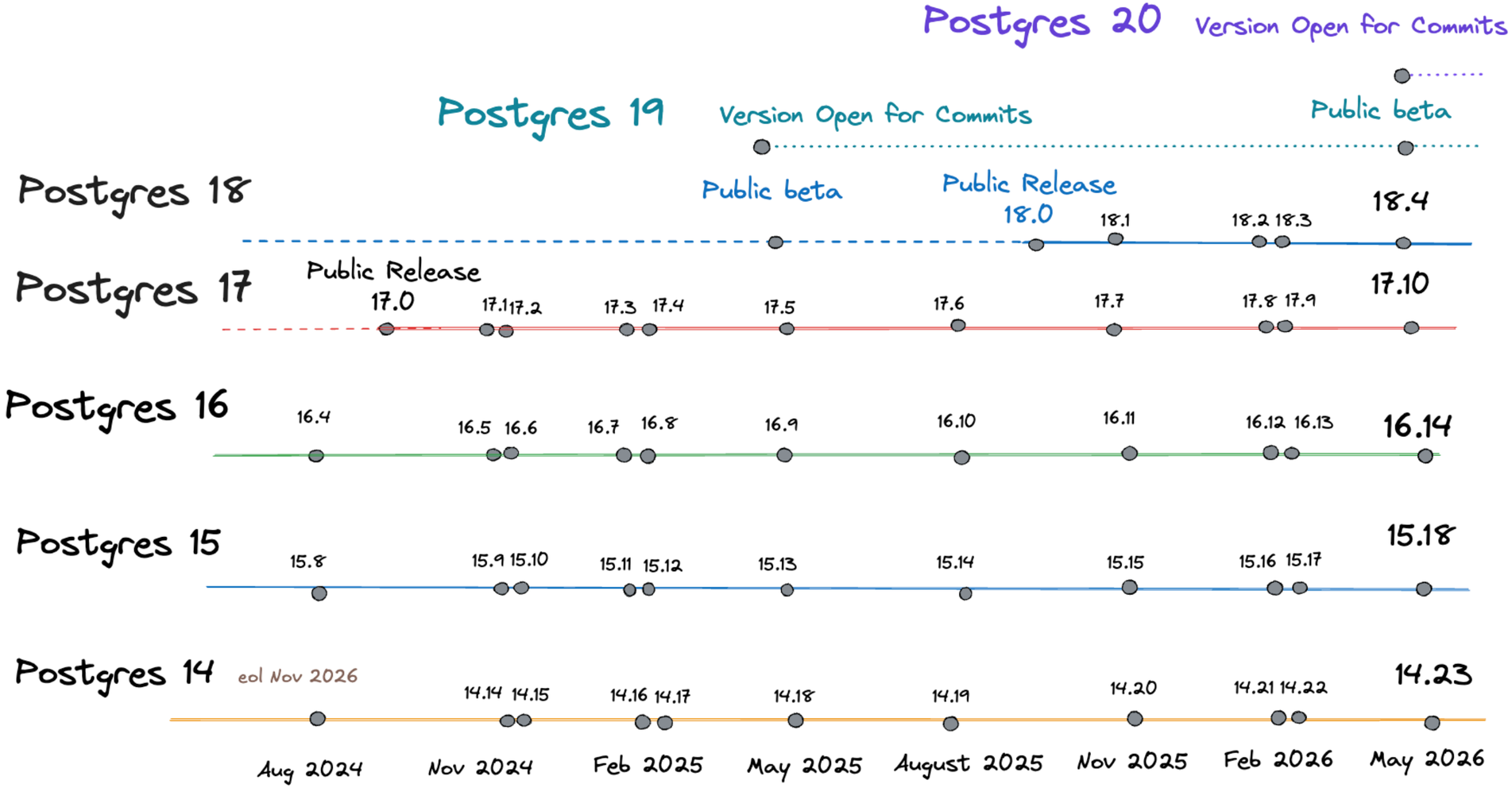
Elizabeth Christensen
Snowflake

About Me

- Elizabeth Christensen
- OSS Advocate at Snowflake
- Postgres contributor and pgUS Board member
- Learn more at Postgres Meetup for All


HAPPY BIRTHDAY
30
years of
POSTGRESQL
SINCE 1996

Postgres Versions Timeline



Postgres 19

- has been slightly delayed
- has had a large number of reverts (this is a healthy project!)

See more about it on the Snowflake Blog:
"What's Happening with Postgres 19"

Helping test new Postgres versions is an OSS contribution

- Community distributions for any OS
- Docker
- Postgres.app for Mac

Postgres versions get better every version

- 3,000+ commits per major version
- 279 authors
- Lots of performance under the hood

Agenda

1. **Exciting SQL Features** — new syntax you'll actually use
2. **Stuff for Free** — performance you get just by upgrading
3. **Easier Management** — REPACK, Oauth
4. **Experimenal features** - plan hints, graph querys
5. **The Postgres Herd Expands** — graphs extensions, lakehouse extensions, analytics extensions

Talk slides and markdown

- `github.com/sfc-gh-echristensen/bluebox`

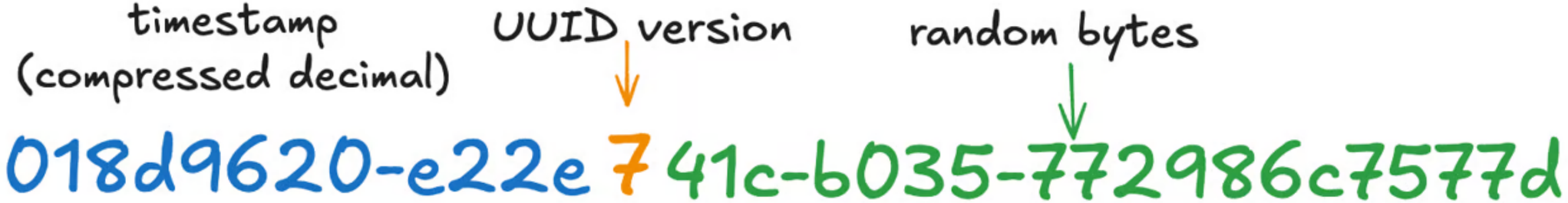


Bluebox sample Postgres data

- Test data set to work with these features
- "RedBox" style buisness, dvd rentals, sites
- Event more in Postgres Full Day Training

Exciting SQL Features

UUID v7 — PG 18


timestamp
(compressed decimal)
UUID version
random bytes
018d9620-e22e 7 41c-b035-772986c7577d

UUID v7 — PG 18

```
SELECT uuidv7();
-- 01a07255-1a44-7ef9-8d5d-0eaa8f0e8c01

-- Use as a default on the Bluebox customer table
CREATE TABLE bluebox.customer_event (
    event_id uuid DEFAULT uuidv7() PRIMARY KEY,
    customer_id int REFERENCES bluebox.customer(customer_id),
    event_type text,
    created_at timestamptz DEFAULT now()
);
```

Virtual Generated Columns

New in Postgres 18

```
CREATE TABLE products (
    id serial PRIMARY KEY,
    price numeric,
    tax_rate numeric DEFAULT 0.099995,
    total_price numeric GENERATED ALWAYS AS (price* (1 + tax_rate)) VIRTUAL
);
```

- Creates a column based on other data
- Not stored on disk, created on the fly
- Immutable functions, no sub-queries

Virtual Generated Columns — PG 18

```
-- Bluebox: auto-classify films by rating
CREATE TABLE bluebox.film_summary (
    film_id bigint PRIMARY KEY,
    title text NOT NULL,
    vote_average real,
    tier text GENERATED ALWAYS AS (
        CASE
            WHEN vote_average >= 8 THEN 'top-rated'
            WHEN vote_average >= 6 THEN 'solid'
            ELSE 'underdog'
        END
    ) VIRTUAL
);
```

Virtual Generated Columns — PG 18

```
                 title                  | vote_average |   tier
----------------------------------------+--------------+----------
 Paths of Glory                         |          8.3 | top-rated
 A Streetcar Named Desire               |          7.6 | solid
 Ocean's Eleven                         |          6.4 | solid
 Canadian Bacon                         |          5.7 | underdog
```

- Computed on read, not stored, zero storage overhead
- Perfect for extracting JSON keys, computed fields, classifications
- Not indexable — use `STORED` or expression indexes for that

OLD/NEW in RETURNING — PG 18

```
-- Bluebox: update a film rating and see old vs new
UPDATE bluebox.film_summary
SET vote_average = 9.0
WHERE title = 'Paths of Glory'
RETURNING OLD.title, OLD.vote_average AS old_rating, OLD.tier AS old_tier,
          NEW.vote_average AS new_rating, NEW.tier AS new_tier;
```

OLD/NEW in RETURNING — PG 18

```
     title       | old_rating | old_tier   | new_rating | new_tier
-----------------+------------+------------+------------+-----------
 Paths of Glory |        8.3 | top-rated  |          9 | top-rated
```

- See before and after in one statement
- Works with `UPDATE`, `DELETE`, and `INSERT`
- Audit logging, triggers, change tracking, event sourcing — all in one round trip

INSERT … ON CONFLICT … RETURNING — PG 19

```
-- Bluebox: add to watchlist, or return existing entry
INSERT INTO bluebox.watchlist (customer_id, film_id)
VALUES (1717, 555285)
ON CONFLICT (customer_id, film_id) DO SELECT
RETURNING customer_id, film_id, added_at;
```

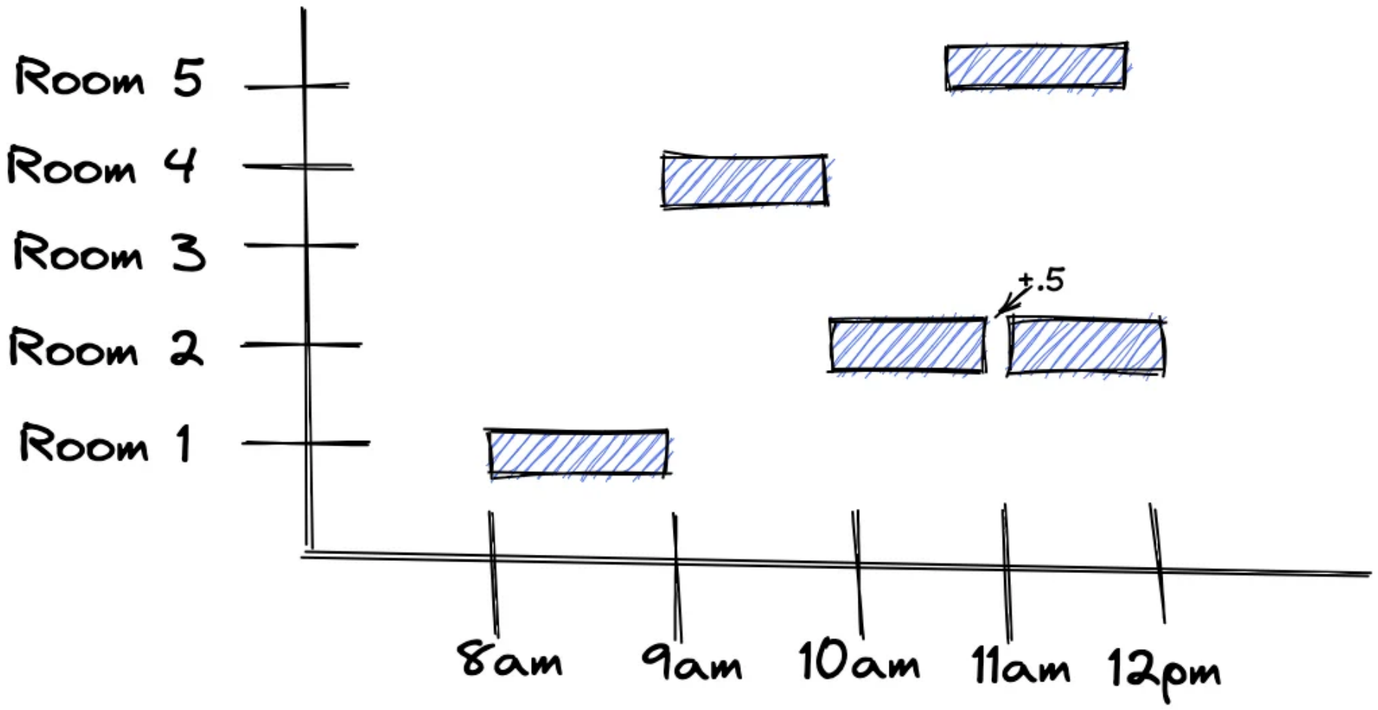
INSERT ... ON CONFLICT RETURNING — PG 19

```
-- First run:  inserts, returns new row with current timestamp
-- Second run: conflict, returns the same row with original timestamp
 customer_id | film_id |           added_at
-------------+---------+-------------------------------
        1717 |  555285 | 2026-09-05 16:10:13.709734+00
```

- Upsert-or-fetch in a single statement, one round trip
- No more `SELECT` + `INSERT` race conditions
- No more `ON CONFLICT DO UPDATE SET col = col` hacks

WITHOUT OVERLAPS — PG 18

Temporal range constraints



Temporal the old way

```
ALTER TABLE bluebox.rental_schedule ADD CONSTRAINT rental_no_overlap
EXCLUDE USING GIST (
    box (
        point(
            extract(epoch from lower(rental_period)),
            inventory_id
        ),
        point(
            extract(epoch from upper(rental_period)) - 0.5,
            inventory_id + 0.5
        )
    )
    WITH &&
);
```

WITHOUT OVERLAPS — PG 18

```
CREATE EXTENSION IF NOT EXISTS btree_gist;

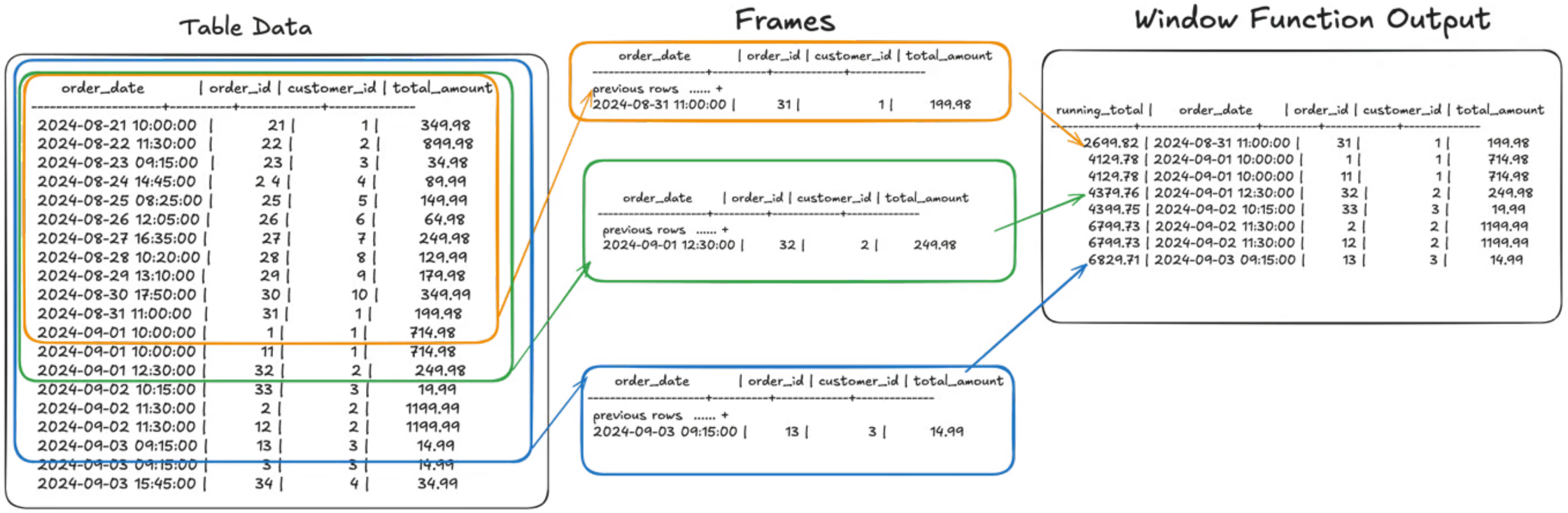
-- Bluebox: prevent double-booking an inventory item
CREATE TABLE bluebox.rental_schedule (
    inventory_id int,
    rental_period tstzrange,
    customer_id int,
    PRIMARY KEY (inventory_id, rental_period WITHOUT OVERLAPS)
);
```

WITHOUT OVERLAPS — The New B-tree GiST Index

PG 18 adds a native **B-tree GiST** index type — a hybrid that supports both equality (=) and range overlap (&&) in a single index.

- One index handles both:
  - inventory_id B-tree equality
  - rental_period && range - GiST overlap
- The `btree_gist` contrib extension is required
- Works in both `PRIMARY KEY` and `UNIQUE` constraints

IGNORE NULLS in Window Functions — PG 19


Table Data
order_date | order_id | customer_id | total_amount
2024-08-21 10:00:00 | 21 | 1 | 349.98
2024-08-22 11:30:00 | 22 | 2 | 899.98
2024-08-23 09:15:00 | 23 | 3 | 34.98
2024-08-24 14:45:00 | 24 | 4 | 89.99
2024-08-25 08:25:00 | 25 | 5 | 149.99
2024-08-26 12:05:00 | 26 | 6 | 64.98
2024-08-27 16:35:00 | 27 | 7 | 249.98
2024-08-28 10:20:00 | 28 | 8 | 129.99
2024-08-29 13:10:00 | 29 | 9 | 179.98
2024-08-30 17:50:00 | 30 | 10 | 349.99
2024-08-31 11:00:00 | 31 | 1 | 199.98
2024-09-01 10:00:00 | 1 | 1 | 714.98
2024-09-01 10:00:00 | 11 | 1 | 714.98
2024-09-01 12:30:00 | 32 | 2 | 249.98
2024-09-02 10:15:00 | 33 | 3 | 19.99
2024-09-02 11:30:00 | 2 | 2 | 1199.99
2024-09-02 11:30:00 | 12 | 2 | 1199.99
2024-09-03 09:15:00 | 13 | 3 | 14.99
2024-09-03 09:15:00 | 3 | 3 | 14.99
2024-09-03 15:45:00 | 34 | 4 | 34.99
Frames
order_date | order_id | customer_id | total_amount
previous rows ..... +
2024-08-31 11:00:00 | 31 | 1 | 199.98
order_date | order_id | customer_id | total_amount
previous rows ..... +
2024-09-01 12:30:00 | 32 | 2 | 249.98
order_date | order_id | customer_id | total_amount
previous rows ..... +
2024-09-03 09:15:00 | 13 | 3 | 14.99
Window Function Output
running_total | order_date | order_id | customer_id | total_amount
2699.82 | 2024-08-31 11:00:00 | 31 | 1 | 199.98
4129.78 | 2024-09-01 10:00:00 | 1 | 1 | 714.98
4129.78 | 2024-09-01 10:00:00 | 11 | 1 | 714.98
4379.76 | 2024-09-01 12:30:00 | 32 | 2 | 249.98
4399.75 | 2024-09-02 10:15:00 | 33 | 3 | 19.99
6799.73 | 2024-09-02 11:30:00 | 2 | 2 | 1199.99
6799.73 | 2024-09-02 11:30:00 | 12 | 2 | 1199.99
6829.71 | 2024-09-03 09:15:00 | 13 | 3 | 14.99

IGNORE NULLS in Window Functions — PG 19

```sql
-- Bluebox: film budgets are sparse — fill forward the last known budget
SELECT title, release_date,
    CASE WHEN budget > 0 THEN budget END AS budget,
    LAST_VALUE(CASE WHEN budget > 0 THEN budget END) IGNORE NULLS OVER (
        ORDER BY release_date
        ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
    ) AS last_known_budget
FROM bluebox.film
WHERE release_date BETWEEN '2020-01-01' AND '2020-06-30'
ORDER BY release_date LIMIT 12;
```

IGNORE NULLS in Window Functions — PG 19

```
             title                 | release_date |  budget   | last_known_budget
-----------------------------------+--------------+-----------+------------------
 Legion of Super-Heroes            | 2020-01-01   |           |
 Spinster                          | 2020-01-01   |           |
 Underwater                        | 2020-01-10   |  50000000 |          50000000
 VHYes                             | 2020-01-12   |           |          50000000
 Dolittle                          | 2020-01-17   | 175000000 |         175000000
 A Fall from Grace                 | 2020-01-17   |           |         175000000
```

- Works with `LAG`, `LEAD`, `FIRST_VALUE`, `LAST_VALUE`, `NTH_VALUE`
- Sparse data / gap-filling — no more `LATERAL` subquery workarounds

SQL Features Recap

Feature	PG Version
UUID v7	18
Virtual Generated Columns	18
OLD/NEW in RETURNING	18
INSERT ON CONFLICT DO SELECT	19
WITHOUT OVERLAPS	18
IGNORE NULLS (window functions)	19

Postgres Goodies You Get for Free

Just upgrade. No code changes.

Postgres Asynchronous i/o

Postgres 18+

What does it mean? 2-3x faster reads

How? Disk reads start but other tasks can be completed while requests are waiting

Configs

io_method=worker	async workers interact the disk to fetch data for shared buffers
io_method=io_uring	utilizes Linux io_uring support

Async I/O — PG 18

2-3x faster reads

- Disk reads can now happen asynchronously — other work continues while waiting for I/O
- Linux: native `io_uring` support (kernel 5.1+)
- Works out of the box — no configuration needed
- Biggest wins on sequential scans and bitmap heap scans

Async I/O — The Reality Check

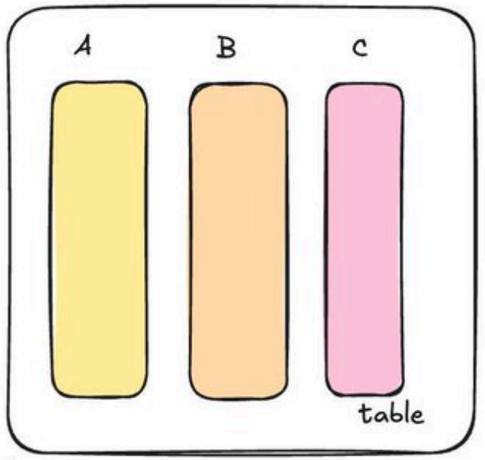
- **Most reads hit buffer cache** — async I/O only helps when Postgres goes to disk
- **Lots of other disk and storge perf** nvme, EBS, others of other stuff affects this too
- **More coming** This took 5 years of engineering to rebuild the I/O path. Async writes, WAL, and Direct I/O are next.

pganalyze benchmarks · *PlanetScale benchmarks*

Postgres Multi-column Indexes

Use one or more columns in the same index

```
CREATE INDEX idx_abc ON measurements (a, b, c);
```



Queries only all the columns ordered left to right will use the index

✓ `SELECT * FROM list WHERE a = 10`

✓ `SELECT * FROM list WHERE a = 10 AND b = 50;`

✓ `SELECT * FROM list WHERE a = 10 AND b = 50 AND c = 100;`

Postgres < 18 Queries missing the left most column will not use the index

✗ `SELECT * FROM measurements WHERE b = 50;`

✗ `SELECT * FROM measurements WHERE b = 50 AND c=75;`

Postgres 18+ any columns can be included in the index. This is part of the new skip scan.

✓ `SELECT * FROM measurements WHERE b = 50;`

✓ `SELECT * FROM measurements WHERE b = 50 AND c=75;`

B-tree Skip Scans — PG 18

```
-- 3-column index
CREATE INDEX idx_inv_3col ON bluebox.inventory (store_id, status_id, film_id);

-- PG 18: skip scans over both leading columns to reach film_id
SELECT inventory_id
FROM bluebox.inventory
WHERE film_id = 555285;

--  Bitmap Index Scan on idx_inv_3col
--    Index Cond: (film_id = 555285)
```

Index Build Improvements — PG 18

- **Parallel GIN index creation** — GIN indexes (JSONB, arrays, full-text search) now build in parallel
- **Sorted GiST builds** — range-type GiST indexes build faster with sorted input
- **Incremental sort for index builds** — less memory pressure during large index operations

These are all "just upgrade" improvements — existing `CREATE INDEX` statements get faster automatically.

Performance Recap

Improvement	PG Version	Benefit
Async I/O	18	2-3x faster reads
B-tree Skip Scans	18	Composite indexes work for more queries
Other index improvements	18	Faster JSONB/FTS index creation, faster ranges
Lots of perofrmance	Every version	Lots of performance improvements in every version

Zero code changes. Just upgrade.

Easier Management

REPACK (PG 19)

```
REPACK (CONCURRENTLY) my_table;
```

REPACK — The Bloat Problem

- MVCC multi-version concurrency control
- `INSERT` `UPDATE` create dead rows
- `VACUUM` reclaims space for reuse, `AUTOVACUUM` runs this for you
- Tuning autovacuum can help
- Extension like `pg_repack` and `pg_squeeze`

REPACK — Query internal tables for Postgres bloat

```sql
-- Find your bloated tables
SELECT schemaname, relname,
    pg_size_pretty(pg_relation_size(schemaname || '.' || relname)) AS size,
    n_live_tup, n_dead_tup,
    CASE WHEN n_live_tup > 0
         THEN round((n_dead_tup::float / n_live_tup::float)::numeric, 4)
    END AS dead_tup_ratio
FROM pg_stat_user_tables
ORDER BY dead_tup_ratio DESC NULLS LAST;
```

REPACK — Three Modes (PG 19)

```
-- Mode 1: Basic (ACCESS EXCLUSIVE lock, like VACUUM FULL)
REPACK my_table;

-- Mode 2: Online rebuild (table stays accessible!)
REPACK (CONCURRENTLY) my_table;

-- Mode 3: Online + physical reorder by index
REPACK (CONCURRENTLY) my_table USING INDEX my_idx;
```

REPACK CONCURRENTLY

new in Postgres 19



Snapshot creation

Take consistent snapshot of table
Create logical replication slot at same *LSN*

Copy Phase

- Copy old table to new table
- Concurrent changes captured by replication slot
- Rebuild indexes

Final Swap (brief read/write lock)

- Acquire lock
- Apply any remaining *WAL* changes
- Swap relfilenodes (old file ↔ new file)
- Drop old table + indexes
- Drop replication slot
- Release lock

Vaccuum vs REPACK

When to use what:

- Autovacuum is fine for the vast majority of use cases
- Tune autovaccum
- Reach for repack later when that doesn work

Autovacuum Parallel Workers — PG 19

Large tables get maintained faster.

```
-- Set at server level
autovacuum_max_parallel_workers = 4

-- Or per-table
ALTER TABLE big_table SET (autovacuum_max_parallel_workers = 2);
```

- New scoring system prioritizes which tables get vacuumed first
- Parallel workers split the index cleanup phase

OAuth 2.0 Authentication — PG 18

Postgres now supports OAuth 2.0 authentication

- Works with Okta, Keycloak, Entra ID, Auth0!
- No more managing Postgres-specific passwords for SSO environments

OAuth 2.0 Authentication — PG 18

```
-- Snowflake Postgres auth setup
ALTER POSTGRES INSTANCE my_postgres_instance SET
  AUTHENTICATION_AUTHORITY = EXTERNAL_OAUTH,
  AUTHENTICATION_PROPERTIES = (
    ISSUER = 'https://sts.windows.net/<tenant-id>/v2.0',
    AUDIENCE = 'api://<RESOURCE_APP_ID>',
    ACCEPTED_TOKENS = (
      -- Human/delegated flow: identity from 'email' claim, scope in 'scp' claim
      (MAPPING_CLAIM = 'email', SCOPES = ('postgres.login.scope')),

      -- Application/client credentials flow: identity from 'appid' claim, scope in 'roles' claim
      (MAPPING_CLAIM = 'appid', SCOPES_CLAIM = 'roles', SCOPES = ('Postgres.Login.Role'))
    )
  );
```

MD5 Passwords Deprecated — PG 18

MD5 password hashing is officially deprecated in PG 18 — will be fully removed by PG 21

- `scram-sha-256` has been the default since PG 14, but old users may still have MD5 hashes
- **Find MD5 users:** `SELECT rolname FROM pg_authid WHERE rolpassword LIKE 'md5%';`
- **Fix them:** `ALTER ROLE myuser PASSWORD 'newpassword';` (re-hashing with same password works too —

Experimental Features

New ideas for Postgres. Not sure where they'll go.

Postgres has never had hints!

PostgreSQL's optimizer uses `pg_statistic` to make decisions:
- Row counts and page counts per table
- Most common values and their frequencies
- Histograms of value distribution
- Correlation (physical vs. logical ordering)
- NULL fraction

It is very good. You almost never need hints.

pg*plan*advice — PG 19

```
CREATE EXTENSION pg_plan_advice;

-- Override with your own advice
SET pg_plan_advice.advice =
  'JOIN_ORDER(o c) NESTED_LOOP_MEMOIZE(c) INDEX_SCAN(o idx_orders_customer)';
```

EXPLAIN to get plan advice

```
EXPLAIN (COSTS OFF, PLAN_ADVICE)
SELECT c.name, c.tier, count(*) AS order_count, sum(o.amount) AS total
FROM customers c
JOIN orders o ON o.customer_id = c.id
WHERE c.tier = 'VIP'
GROUP BY c.name, c.tier;
```

```
 Generated Plan Advice:
   JOIN_ORDER(o c)
   HASH_JOIN(c)
   SEQ_SCAN(o c)
   GATHER_MERGE((c o))
```

pg*stash*advice — Plan Pinning

Pin advice to query IDs — no application code changes.

```
CREATE EXTENSION pg_stash_advice;

-- Create a named stash
SELECT pg_create_advice_stash('production_fixes');

-- Get the query ID from EXPLAIN
EXPLAIN (VERBOSE, PLAN_ADVICE) SELECT ...;
-- Query Identifier: 9122549731181782750

-- Pin advice to that query
SELECT pg_set_stashed_advice(
    'production_fixes',
    9122549731181782750,
    'JOIN_ORDER(c o) NESTED_LOOP_MEMOIZE(o) INDEX_SCAN(o idx_orders_customer)'
);
```

pg*plan*advice — PG 19 - Scoping

- session
- user
- db
- query_id

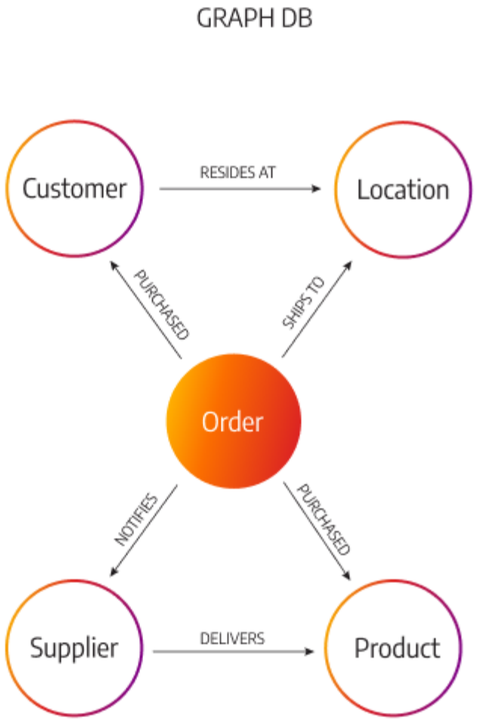
Query Plan Hints — When you might use it

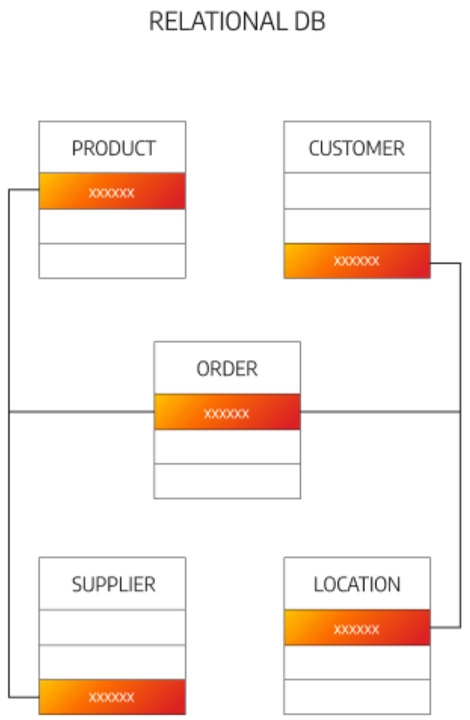
1. **Opaque functions** — PL/pgSQL, PostGIS functions → planner defaults to 33% selectivity estimate
2. **Third-party apps/SQL you can't modify** — ORMs, reporting tools

Graphs in Postgres?

Many real-world problems are graph problems:
- Fraud detection, transaction networks
- Recommendations, co-purchases, co-actors
- Permission graphs, who can access what
- Supply chains, parts → assemblies → products

Graph vs relational





Graphs in Postgres?

Before PG 19: Recursive CTEs — powerful but not really graphs

PG 19 Reverted feature: Property graph queries over your existing tables

Apage Age Extension: Full Cypher (Neo4j) inside Postgres

CREATE PROPERTY GRAPH (now PG 20)

The graph is a **view** over existing tables — no new storage, no ETL.

```
CREATE PROPERTY GRAPH movie_graph
  VERTEX TABLES (
    person KEY (person_id)
      PROPERTIES (person_id, name, popularity),
    film KEY (film_id)
      PROPERTIES (film_id, title, release_date)
  )
  EDGE TABLES (
    film_cast KEY (film_id, person_id)
      SOURCE KEY (person_id) REFERENCES person (person_id)
      DESTINATION KEY (film_id) REFERENCES film (film_id)
      PROPERTIES (film_character)
  );
```

GRAPH_TABLE Query (now PG 20)

```
-- Find all actors who appeared in the same film as Tom Hanks
SELECT *
FROM GRAPH_TABLE (movie_graph
    MATCH (a IS person)-[IS film_cast]->(f IS film)
          <-[IS film_cast]-(b IS person)
    WHERE a.name = 'Tom Hanks'
      AND b.person_id != a.person_id
    COLUMNS (
        a.name AS actor_a,
        b.name AS actor_b,
        f.title AS shared_film
    )
)
ORDER BY shared_film
LIMIT 10;
```

CTEs to get the same query as graph

```
-- Find all actors who appeared in the same film as Tom Hanks
SELECT p2.name AS costar, f.title
FROM bluebox.film_cast fc1
JOIN bluebox.film_cast fc2 ON fc2.film_id = fc1.film_id
  AND fc2.person_id != fc1.person_id
JOIN bluebox.person p1 ON p1.person_id = fc1.person_id
JOIN bluebox.person p2 ON p2.person_id = fc2.person_id
JOIN bluebox.film f ON f.film_id = fc1.film_id
WHERE p1.name = 'Tom Hanks'
LIMIT 10;
```

Apache AGE — Building the Graph

Unlike SQL/PGQ (a view), AGE needs its own graph store. Create and load from your tables:

```
LOAD 'age';
SET search_path = ag_catalog, "$user", public;

-- Create the graph
SELECT create_graph('movie_graph');

-- Load nodes from Bluebox tables
SELECT * FROM cypher('movie_graph', $$
    CREATE (a:Actor {person_id: id, name: name})
$$, (SELECT person_id AS id, name FROM bluebox.person)) AS (v agtype);

SELECT * FROM cypher('movie_graph', $$
    CREATE (f:Film {film_id: id, title: title})
$$, (SELECT film_id AS id, title FROM bluebox.film)) AS (v agtype);

-- Load edges
SELECT * FROM cypher('movie_graph', $$
    MATCH (a:Actor {person_id: pid}), (f:Film {film_id: fid})
    CREATE (a)-[:ACTED_IN]->(f)
$$, (SELECT person_id AS pid, film_id AS fid FROM bluebox.film_cast)) AS (e agtype);
```

Apache AGE — Cypher in Postgres

- openCypher (Neo4j's query language) as a Postgres extension.
- ASCII pattern matching

() = nodes (entities)
[] = relationships (edges)
-> / <- = direction

```
(a)-[:ACTED_IN]->(f)<-[:ACTED_IN]-(b)
```

Blog: Graph Queries in Postgres with Apache AGE

Apache AGE — Cypher in Postgres

```
SELECT * FROM cypher('movie_graph', $$
    MATCH (a:Actor)-[:ACTED_IN]->(f:Film)<-[:ACTED_IN]-(b:Actor)
    WHERE a.name = 'Tom Hanks'
    RETURN b.name AS costar, f.title AS film
    ORDER BY f.title
    LIMIT 10
$$) AS (costar agtype, film agtype);
```

Apache AGE — Variable-Length Paths

PG 19 Property graph is not variable length

```
-- Six Degrees of Kevin Bacon
SELECT * FROM cypher('movie_graph', $$
    MATCH path = (a:Actor)-[:ACTED_IN*1..4]-(b:Actor)
    WHERE a.name = 'Tom Hanks'
      AND b.name = 'Kevin Bacon'
    RETURN length(path) AS degrees, path
    LIMIT 5
$$) AS (degrees agtype, path agtype);
```

Apache AGE — Variable-Length Paths

`[:ACTED_IN*1..4]` — traverse 1 to 4 hops. SQL/PGQ requires hard-coding each depth.

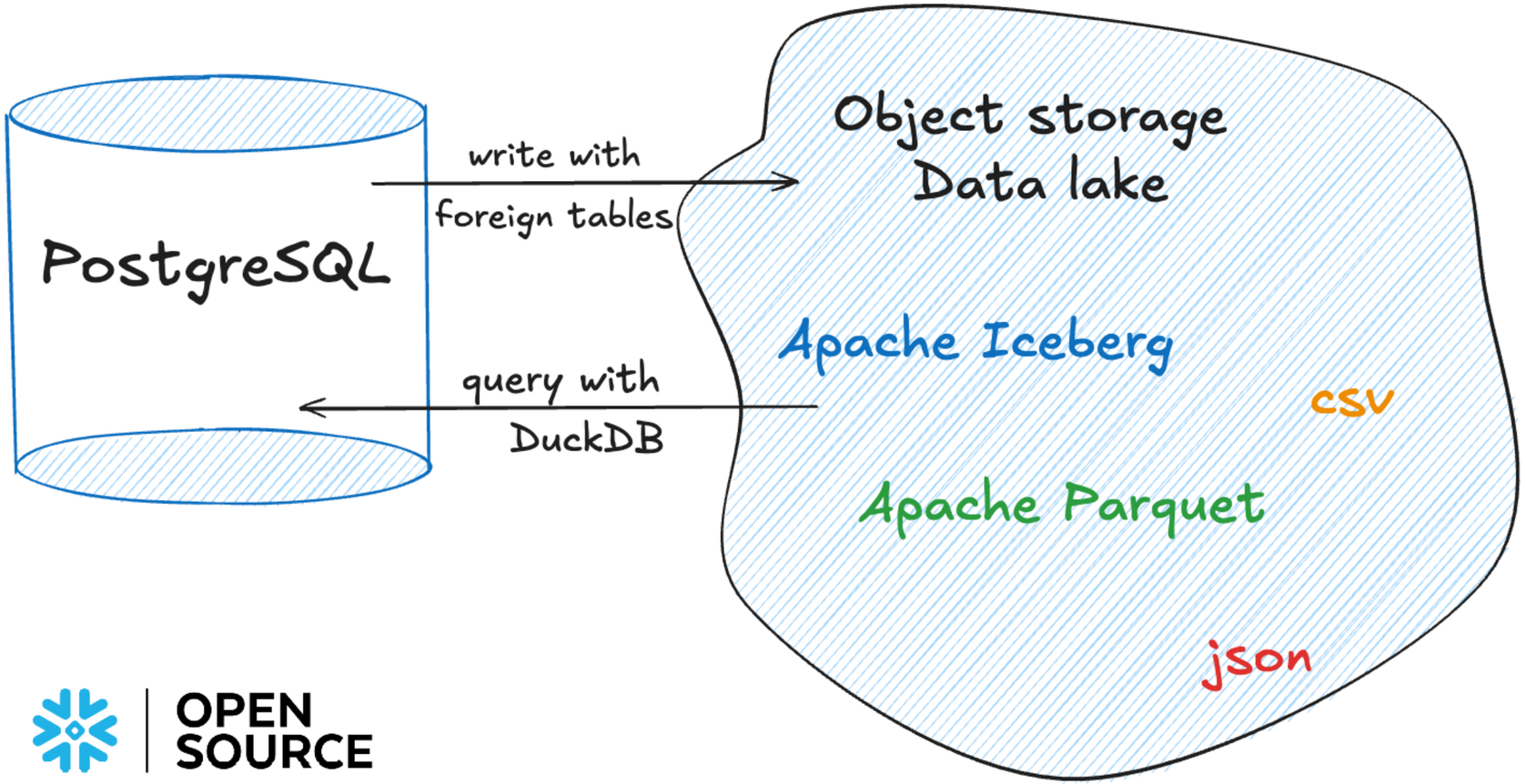
```
 degrees | path
---------+-------------------------------------------------------------
       2 | Tom Hanks -> Forrest Gump <- Gary Sinise -> Apollo 13 <- Kevin Bacon
       2 | Tom Hanks -> That Thing You Do! <- Marc McClure -> Apollo 13 <- Kevin Bacon
       2 | Tom Hanks -> News of the World <- Ray McKinnon -> Apollo 13 <- Kevin Bacon
```

More graphs at KCDC

Beyond Vector Search: Graph Algorithms for Smarter RAG Context
Nathan Smith
Friday 3:30

The Postgres Herd Expands

pg_lake: Postgres for lakehouse



OPEN SOURCE

pg_lake — Postgres as a Lakehouse

```
-- This is an Iceberg table. Stored as Parquet in S3.
CREATE TABLE sensors (
    sensor_id   int,
    sensor_type text,
    model       text,
    installed_at timestamp
) USING iceberg;
```

Any Iceberg-compatible tool can read this data: Snowflake, Spark, Trino, DuckDB, Athena.

pg_lake — Query External Data on S3

Amazon customer reviews — real Parquet, public S3, no auth needed. Schema inferred.

```
-- Point a foreign table at public Amazon reviews
CREATE FOREIGN TABLE amazon_video_reviews ()
SERVER pg_lake
OPTIONS (path 's3://amazon-reviews-pds/parquet/product_category=Digital_Video_Download/');

SELECT product_title, star_rating, review_headline
FROM amazon_video_reviews
ORDER BY total_votes DESC
LIMIT 10;
```

pg_lake — Join S3 Data with Postgres Tables

Public sentiment from S3 + rental data from Postgres in one query.

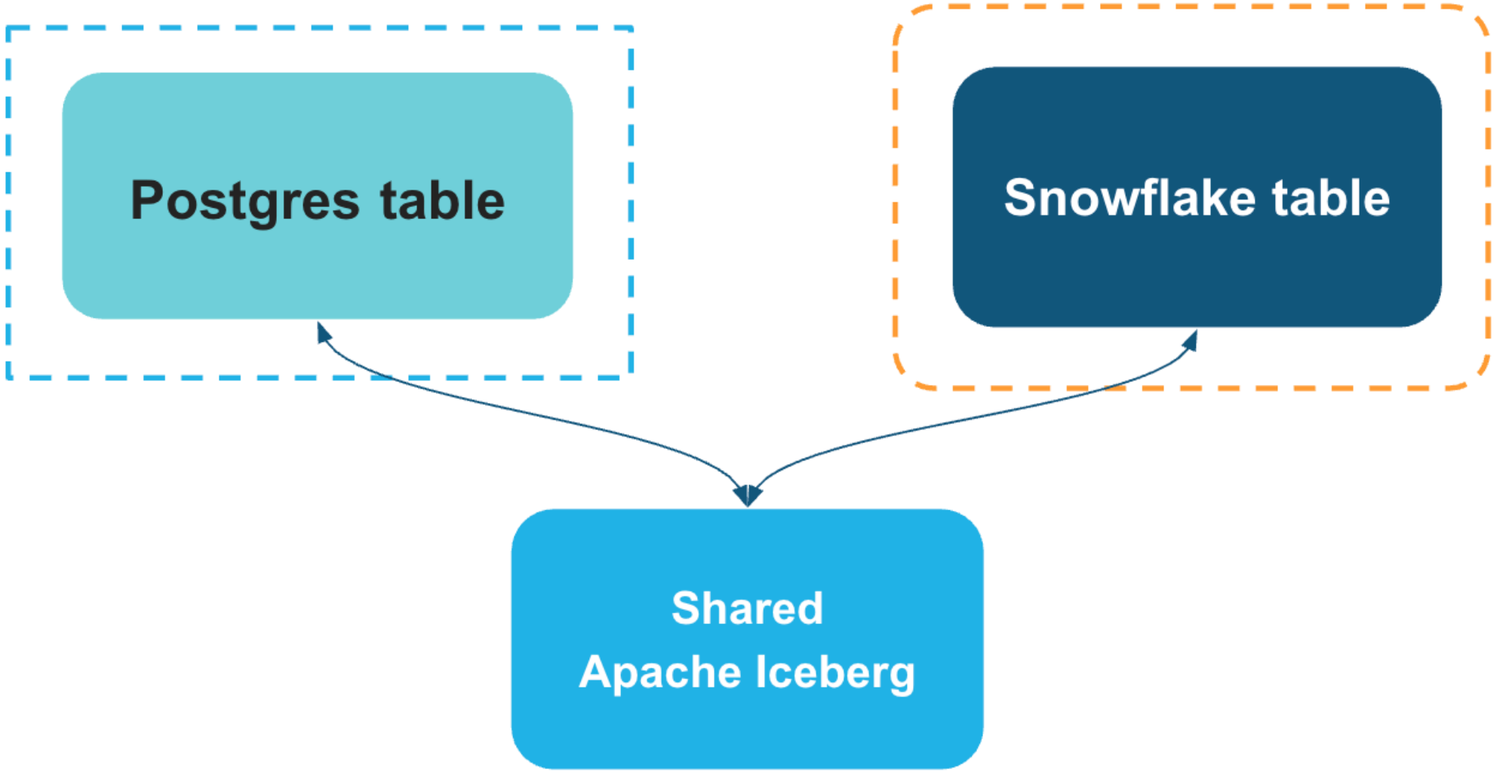
```
-- Films with great Amazon reviews but low Bluebox rentals
SELECT f.title,
       round(avg(r.star_rating), 1) AS avg_amazon_stars,
       count(r.*) AS review_count,
       coalesce(rentals.cnt, 0) AS bluebox_rentals
FROM amazon_video_reviews r
JOIN bluebox.film f ON f.title = r.product_title
LEFT JOIN (
    SELECT i.film_id, count(*) AS cnt
    FROM bluebox.rental ren
    JOIN bluebox.inventory i ON i.inventory_id = ren.inventory_id
    GROUP BY i.film_id
) rentals ON rentals.film_id = f.film_id
GROUP BY f.title, rentals.cnt
HAVING avg(r.star_rating) >= 4
ORDER BY bluebox_rentals ASC
LIMIT 10;
```

pg_lake — Incremental Pipelines

```
-- Schedule with pg_cron: move new data to Iceberg every minute
SELECT cron.schedule('move-to-iceberg', '* * * * *',
    $$SELECT incremental.execute_pipeline('move_metrics_to_iceberg')$$
);

-- pg_incremental apend only stream
SELECT incremental.create_time_interval_pipeline(
    pipeline_name     := 'move_metrics_to_iceberg',
    time_interval     := '1 minute',
    source_table_name := 'metrics_staging',
    start_time        := (SELECT min(ts) FROM metrics_staging),
    command           := $$
        INSERT INTO metrics_iceberg (ts, device_id, metric_name, value)
        SELECT ts, device_id, metric_name, value
        FROM metrics_staging WHERE ts >= $1 AND ts < $2
    $$
);
```

Use Case: Shared Iceberg Table Between Transactions and Analytics



© 2026 Snowflake Inc. All Rights Reserved

Postgres as a Lakehouse is 🔥

- **pg_lake** (Snowflake) — native Iceberg tables via new table access method, DuckDB engine, PG is the catalog. Open source, self-hostable.
- **pg_duckdb** (MotherDuck) — embeds DuckDB inside Postgres for analytical acceleration + lake file I/O. The engine, not the car — no catalog, no sync, no platform integration.
- **pg_mooncake** (acquired by Databricks) — columnar storage engine, Parquet/Iceberg, DuckDB-powered analytics. Now part of

Postgres features for analytics

- Window functions
- CTEs
- HyperLogLog
- Apache DataSketches

Apache DataSketches — The COUNT(DISTINCT) Problem

Sketches are probabilistic data structures that give approximate answers in constant time and constant space.

- ~1-4 KB per sketch regardless of input size
- Error bounds are well-understood (typically < 2%)
- **Mergeable** — union daily sketches to get weekly uniques without re-scanning

*My blog: Approximate Answers in PostgreSQL

Apache DataSketches — Five Sketch Types

Sketch	Purpose	Key Functions
CPC	Compact distinct counting (~30% smaller than HLL)	`cpc_sketch_build()`, `cpc_sketch_get_estimate()`
HLL	Classic distinct counting, broadly interoperable	`hll_sketch_build()`, `hll_sketch_get_estimate()`
Theta	Distinct counting with **set operations**	`theta_sketch_intersection()`, `theta_sketch_a_not_b()`
KLL	Quantile/rank/histogram without sorting	`kll_float_sketch_get_quantile(0.5)` for median
Frequent Strings	Heavy-hitter / top-N detection	`frequent_strings_sketch_result_no_false_negatives()`

Apache DataSketches

```
CREATE EXTENSION datasketches;

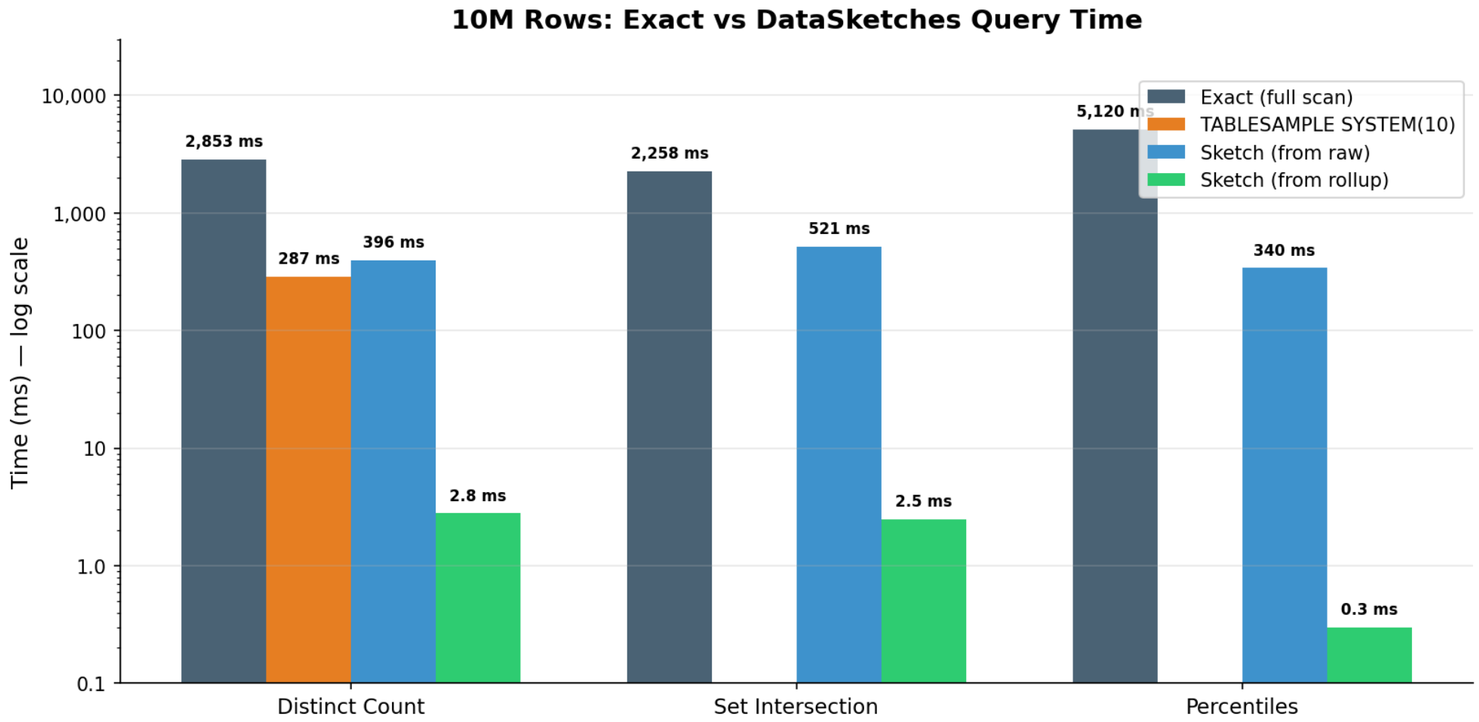
SELECT cpc_sketch_get_estimate(cpc_sketch_build(user_id))
FROM page_views;
```

DataSketches — Pre-Aggregate Then Merge

```
-- 1. Build sketches per dimension during ETL
CREATE TABLE campaign_sketches AS
SELECT
    campaign_id,
    date_trunc('day', event_time) AS day,
    count(*) AS impressions,
    sum(revenue) AS revenue,
    cpc_sketch_build(user_id) AS unique_users_sketch,
    theta_sketch_build(user_id) AS theta_sketch
FROM raw_events
GROUP BY campaign_id, day;

-- 2. Query: merge sketches (instant, regardless of raw data size)
SELECT
    campaign_id,
    sum(impressions) AS total_impressions,
    cpc_sketch_get_estimate(cpc_sketch_union(unique_users_sketch)) AS unique_users
FROM campaign_sketches
WHERE day BETWEEN '2026-01-01' AND '2026-01-31'
GROUP BY campaign_id;
```

DataSketches — Benchmarks (10M rows)



Pre-built rollups: **1,000x to 17,000x faster** than exact full scans.

Summary

PG 18 is out now — upgrade and get:
- Async I/O, UUID v7, skip scans, virtual columns, WITHOUT OVERLAPS

PG 19 beta available — test now:
- REPACK CONCURRENTLY, plan hints, SELECT RETURNING

Extensions keep expanding:
- pg_lake (lakehouse), Apache AGE (graphs), DataSketches (approximate analytics)

Please let me know what you think of this talk!



Questions?

Elizabeth Christensen — @sqlliz
Find me at the Postgres community booth upstairs

Thank you!

Resources

- PostgreSQL 18 Release Notes
- PostgreSQL 19 Beta Notes
- pg_lake on GitHub
- Apache AGE
- Apache DataSketches
- pg*plan*advice docs