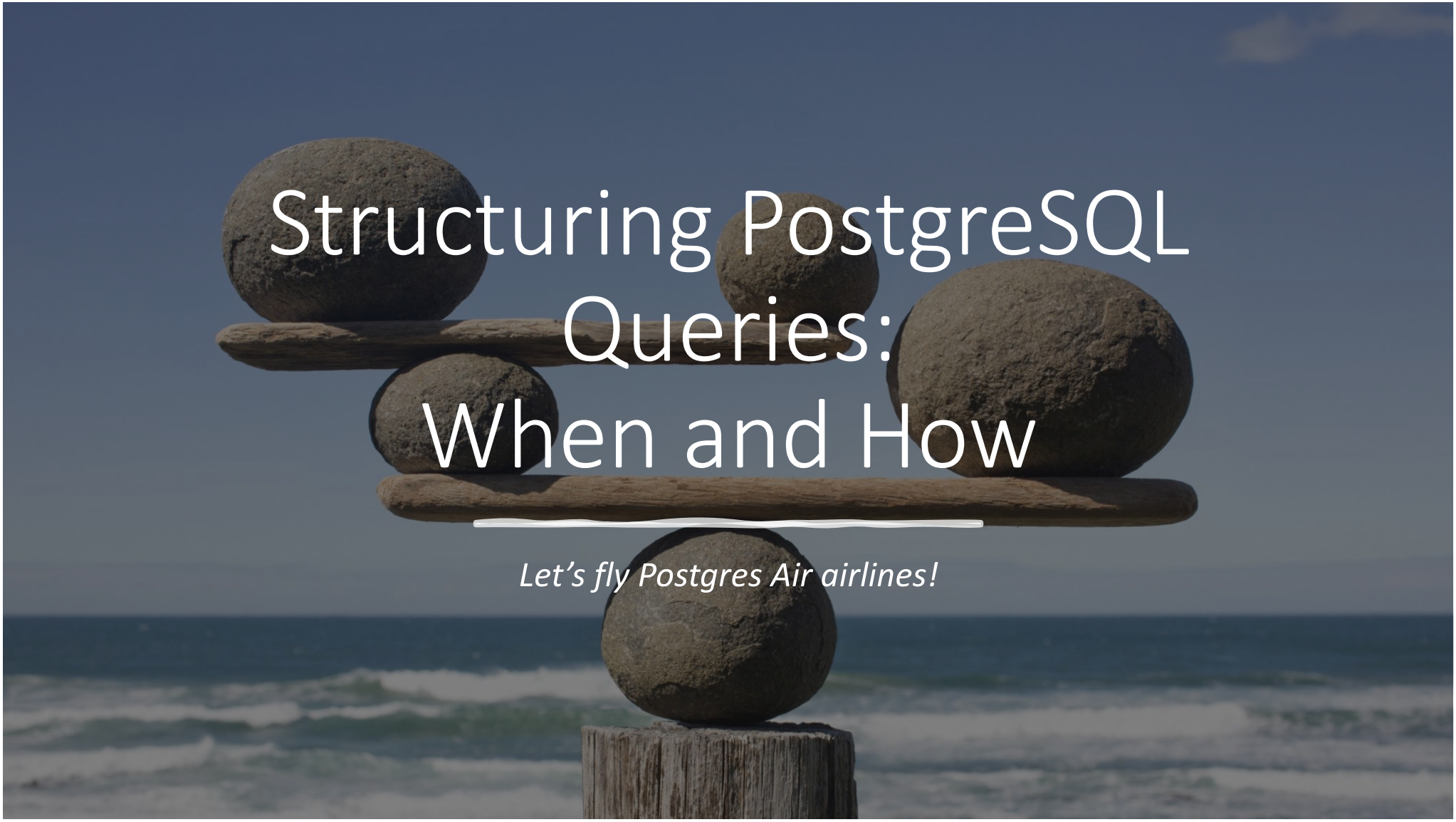


Structuring PostgreSQL Queries: When and How

Let's fly Postgres Air airlines!

Who am I

Hettie Dombrovskaya

- Database Architect at DRW Holdings, Chicago IL
- Founder and President of Prairie Postgres NFP
- Illinois Prairie Postgres User Group Organizer
- ACM Chicago Chapter Communications Chair
- LPI Board of Directors member

PG DATA 2026

https://2026.pg-data.org/

June 4-5 2026

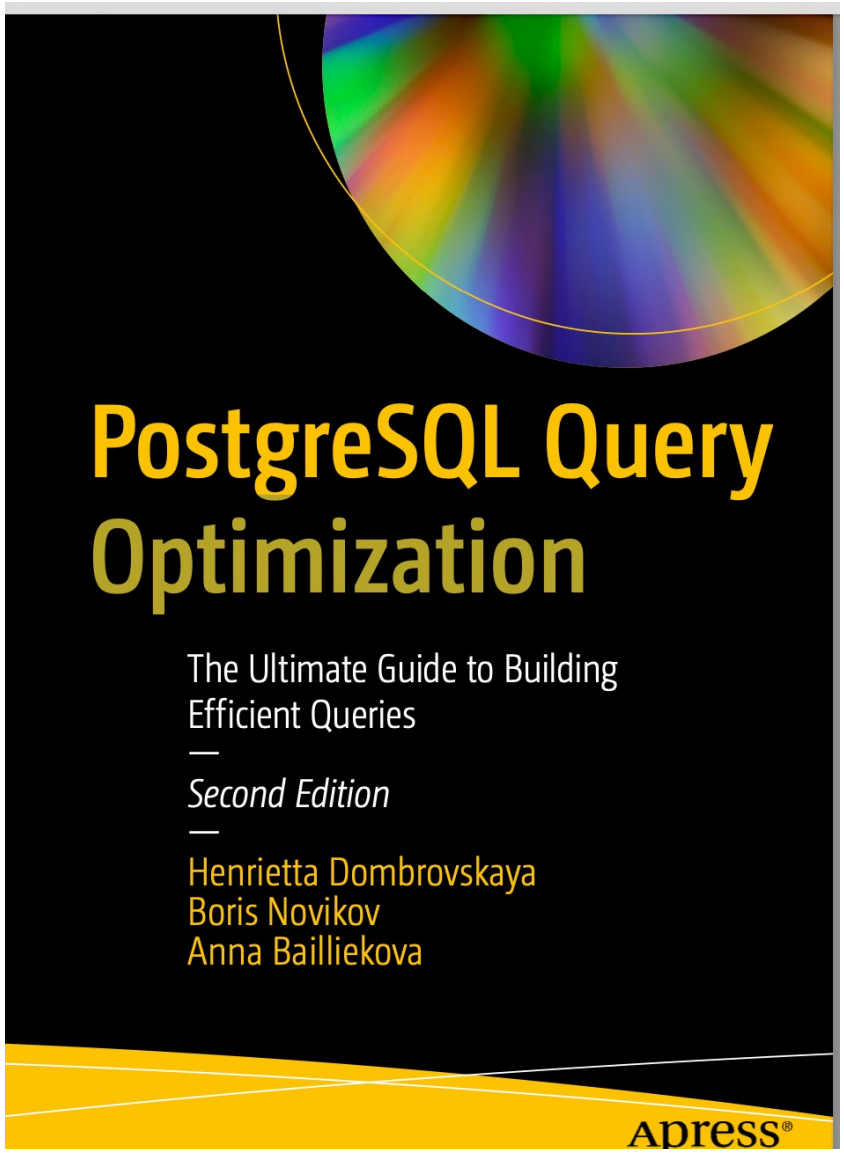
Loyola Water Tower Campus in Chicago

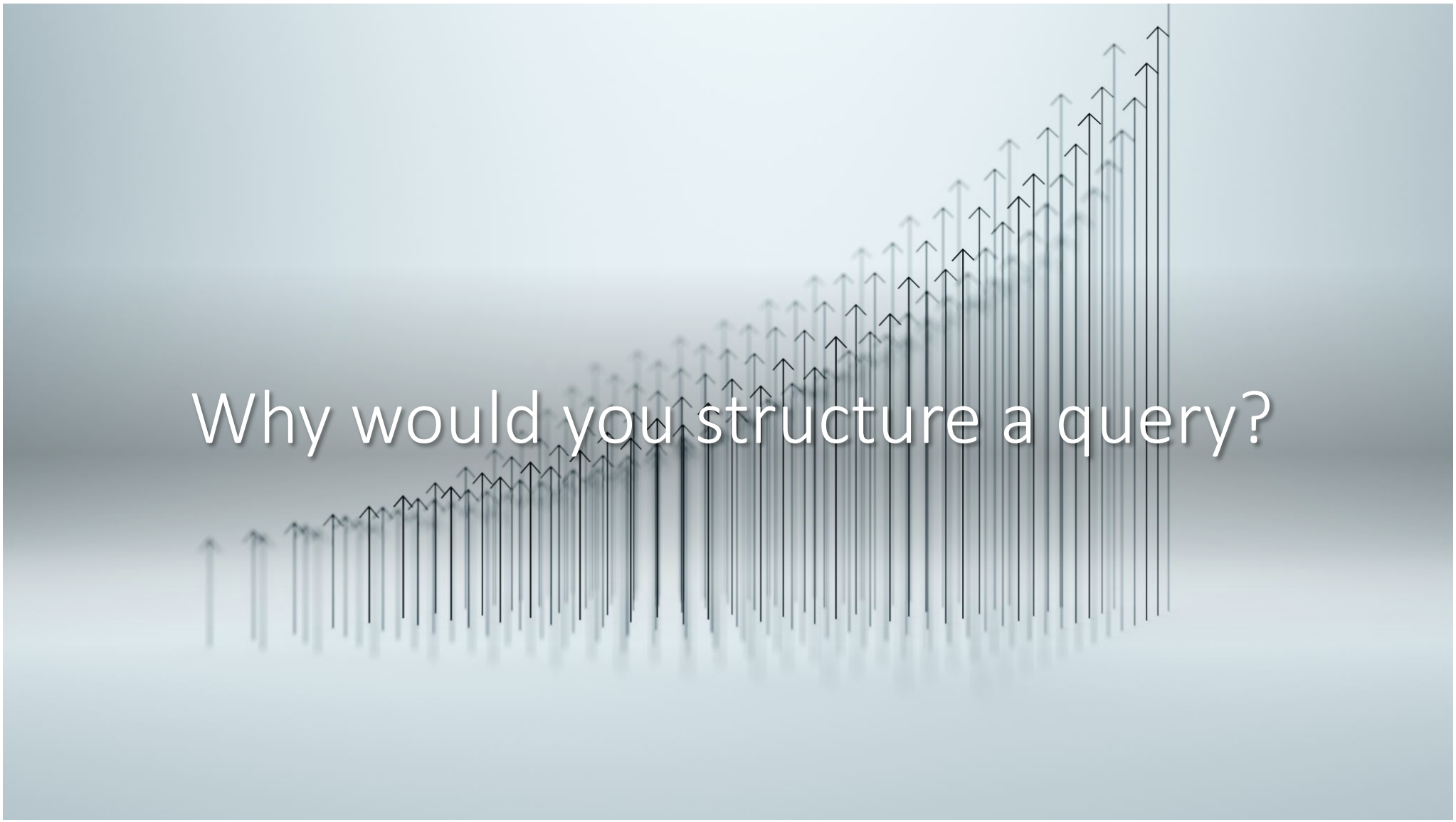
Registration is open

PostgreSQL Query Optimization, Apress 2024 (2nd edition)

postgres_air database:

https://github.com/hettie-d/postgres_air



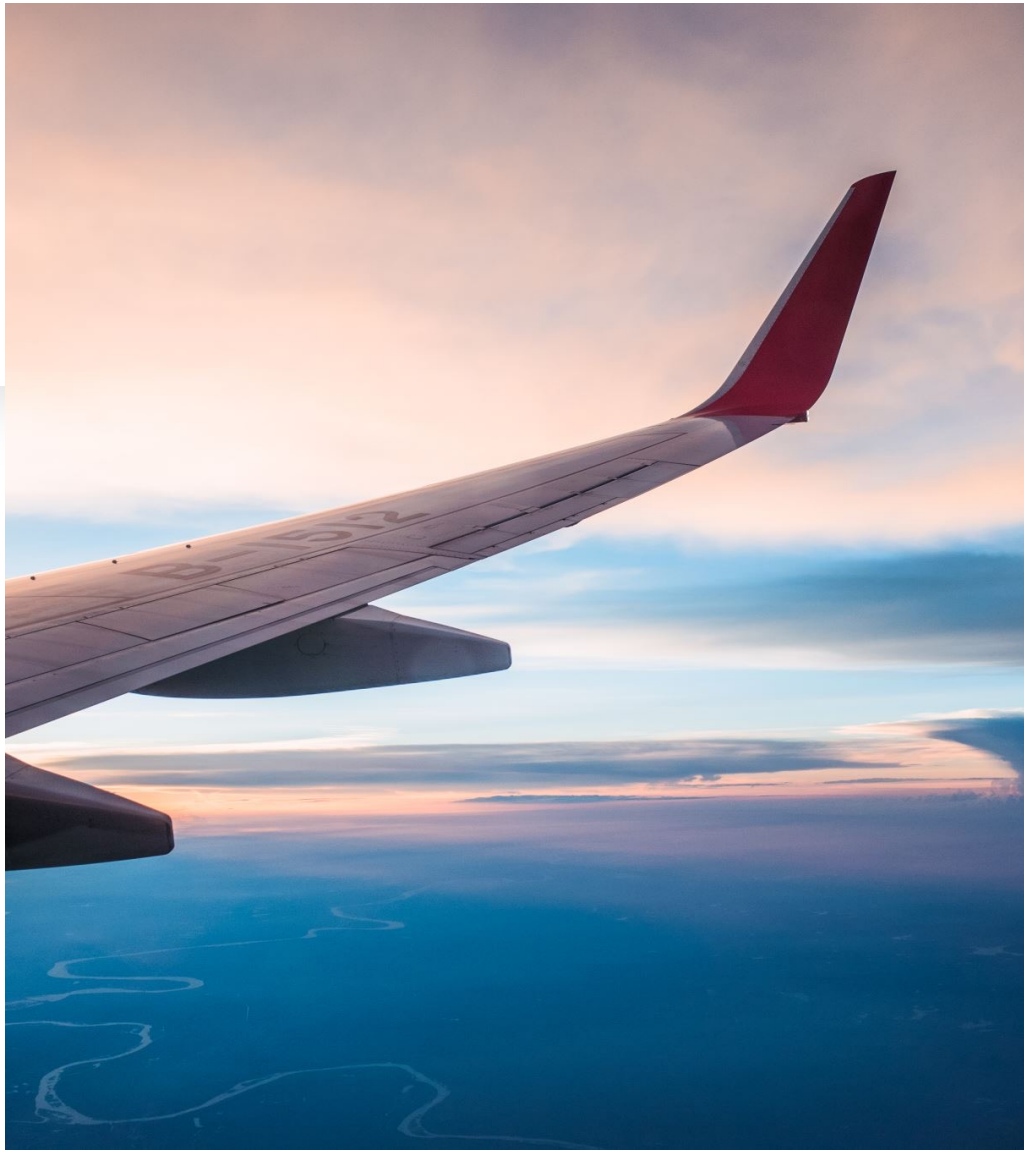

Why would you structure a query?

Code Factoring:

- More readable
- More manageable
- Easier to make changes
- Reusable

What about SQL?

```
SELECT
 ROW( a.application_id ,
    email_address ,
    a.loan_purpose_id ,
        (select loan_purpose from common.loan_purpose where loan_purpose_id=a.loan_purpose_id),
    desired_loan_amount,
    a.income_source_id ,
        (select income_source from common.income_source where income_source_id = a.income_source_id),
    a.product ,
    a.brand ,
    language_id,
    application_number,
    coalesce(coalesce((select customer_status_id from identity.user_account u
              where u.user_account_id = a.user_account_id and application_created_at<@u.effective
                    and now()<@u.asserted),
    (select  customer_status_id  from identity.user_account
           where user_account_id=a.user_account_id
                 and now()<@asserted
--                    and upper(asserted)='infinity'
         order by lower(effective) limit 1)),1),
    a.application_sub_status_id ,
    (select application_sub_status from common.application_sub_status where application_sub_status_id = a.application_sub_status_id),
    (select application_source_type from common.application_source_type ast where ast.application_source_type_id = a.application_source_type_id),
    application_source_id,
    selected_loan_contract_calculation_id,
    user_reported_marketing_channel_id,
    final_disclosure_at ,
    final_disclosure_flag ,
    row(
    primary_addr_city ,
    primary_addr_state_id ,
(select state_name from common.states where state_id=a.primary_addr_state_id),
    primary_addr_line1 ,
    primary_addr_line2 ,
    primary_addr_zip ,
    a.time_at_address_id ,
    (select time_at_address from common.time_at_address where time_at_address_id=a.time_at_address_id),
    own_property,
    rent )::origination.address_record,
    row(primary_phone ,
    primary_phone_type_id ,
        (select phone_type from common.phone_type where phone_type_id=primary_phone_type_id),
    secondary_phone ,
    secondary_phone_type_id ,
    (select phone_type from common.phone_type where phone_type_id=secondary_phone_type_id))::origination.phone_record,
    row (ssn ,
    dob ,
    first_name  ,
    last_name  ,
    middle_initial,
    a.user_account_id,
    (select person_id from identity.user_account where user_account_id = a.user_account_id and now()<@effective and now()<@asserted),
    govt_id,
    issuing_state_id)::origination.userinfo_record,
    (select row(adjusted_income_frequency ,
    max_offer_amount)::origination.pricing_info_record from origination.balance_pricing b where b.application_id = a.application_id and now()<@effective
and now()<@asserted),
    row(employer_name ,
    employer_phone,
    a.job_title,
    a.industry_id,
    a.income_type_id,
    a.take_home_amount,
    a.gross_income_per_frequency,
    a.gross_monthly_income,
    a.employment_duration_id ,
        (select employment_duration from common.employment_duration where employment_duration_id = a.employment_duration_id),
    a.payment_frequency_id ,
    payment_frequency,
    payment_frequency_code,
    pay_period ,
    pay_day1 ,
    pay_day2 ,
    pay_dow ,
    last_paycheck_date,
    first_paycheck_date,
    second_paycheck_date,
    first_payment_date,
    second_payment_date,
    schedule_aligned):: origination.employer_record,
    row(a.bank_account_type_id,
        (select bank_account_type from common.bank_account_type where bank_account_type_id=a.bank_account_type_id),
    bank_name ,
    name_on_account ,
    is_checking ,
    routing_number,
    account_number,
    account_number_confirm ,
    a.payment_method_id ,
        (select payment_method from common.payment_method where payment_method_id=a.payment_method_id))::origination.bank_record,
    (select
    array_agg( row(a1.application_sub_status_id,
    s1.application_sub_status,
    a1. effective)::origination.app_history_record)
    from (select application_id,
    application_sub_status_id,
    min(lower(effective)) as effective from origination.application
    group by 1,2
    order by min(lower(effective))) a1
    join common.application_sub_status s1 using(application_sub_status_id)
    where a1.application_id=a.application_id
    ) ,
    application_created_at ,
```



I can’t understand what this query is doing!

It’s loo long!

Repeated SQL!

LET’S DO SOMETHING!


Temporary Tables

Temporary tables

```
CREATE TEMP TABLE interim_results AS
  SELECT …

Often, we see:

CREATE TEMP TABLE T1 AS SELECT  * FROM A ;
CREATE TEMP TABLE T2 AS SELECT t1.* , b.* FROM T1
  JOIN B WHERE c=d;
CREATE TEMP TABLE T3 AS SELECT b.*, e.* FROM B
  JOIN e WHERE h=g;
CREATE TEMP TABLE T4 AS SELECT t1.*, t2.* FROM T1
JOIN T2 WHERE…
```

What's the problem?

- Indexes
- Statistics
- Excessive I/O.
- Most importantly: block the optimizer by

  "locking" the order of operations

Example of inefficient usage

```
CREATE TEMP TABLE flights_totals AS
SELECT bl.flight_id,
       departure_airport,
      (avg(price))::numeric (7,2) AS avg_price,
       count(DISTINCT passenger_id) AS num_passengers
FROM booking b
JOIN booking_leg bl USING (booking_id)
JOIN flight f USING (flight_id)
JOIN passenger p USING (booking_id)
GROUP BY 1,2;
```

---**15 min; 500,000 rows.**

```
SELECT * FROM flights_totals
WHERE departure_airport='ORD’
```

--- **10,000 rows**

Without temp table

```
SELECT * FROM
(SELECT bl.flight_id,
        departure_airport,
        (avg(price))::numeric (7,2) AS avg_price,
        count(DISTINCT passenger_id) AS num_passengers
FROM booking b
JOIN booking_leg bl USING (booking_id)
JOIN flight f USING (flight_id)
JOIN passenger p USING (booking_id)
GROUP BY 1,2) a
WHERE departure_airport='ORD'
```

--- **1 min**


Common Table Expressions
(CTE)

Common Table Expressions (CTE)

Common Table Expressions: temporary tables that exist just for one query.

WITH SELECT|NSERT|UPDATE|DELETE

SELECT|NSERT|UPDATE|DELETE

CTE Example

```
WITH flights_totals AS(
SELECT bl.flight_id,
       departure_airport,
       (avg(price))::numeric (7,2) AS avg_price,
       count(DISTINCT passenger_id) AS num_passengers
FROM booking b
JOIN booking_leg bl USING (booking_id)
JOIN flight f USING (flight_id)
JOIN passenger p USING (booking_id)
GROUP BY 1,2)
  SELECT flight_id,
         avg_price,
         num_passengers
  FROM flights_totals
  WHERE departure_airport='ORD'
```

How does the execution plan look like?

For PG versions < 12 – same as TEMP TABLE

- Why? By design. Optimization fence (especially with INSERT/UPDATE/DELETE,recursion) + JOIN_COLLAPSE_LIMIT within CTE

For PG 12 and up:

- No recursion AND used once: breaking optimization fence
- More than once – old behavior
- Also, you can force old behavior:

```
WITH flights_totals AS MATERIALIZED ( SELECT …
```

Execution Plan with CTE inlining

	QUERY PLAN text
1	Subquery Scan on flights_totals (cost=899526.09..935621.26 rows=962538 width=26)
2	-> GroupAggregate (cost=899526.09..925995.88 rows=962538 width=30)
3	Group Key: bl.flight_id, f.departure_airport
4	-> Sort (cost=899526.09..901932.43 rows=962538 width=18)
5	Sort Key: bl.flight_id
6	-> Nested Loop (cost=9545.27..784126.73 rows=962538 width=18)
7	Join Filter: (b.booking_id = p.booking_id)
8	-> Nested Loop (cost=9544.84..532480.61 rows=334023 width=26)
9	-> Hash Join (cost=9544.41..367021.97 rows=334023 width=12)
10	Hash Cond: (bl.flight_id = f.flight_id)
11	-> Seq Scan on booking_leg bl (cost=0.00..310506.66 rows=17893566 width=8)
12	-> Hash (cost=9384.99..9384.99 rows=12753 width=8)
13	-> Bitmap Heap Scan on flight f (cost=243.26..9384.99 rows=12753 width=8)
14	Recheck Cond: (departure_airport = 'ORD'::bpchar)
15	-> Bitmap Index Scan on flight_departure_airport (cost=0.00..240.07 rows=12753 width=0)
16	Index Cond: (departure_airport = 'ORD'::bpchar)
17	-> Index Scan using booking_pkey on booking b (cost=0.43..0.50 rows=1 width=14)
18	Index Cond: (booking_id = bl.booking_id)
19	-> Index Scan using passenger_booking_id on passenger p (cost=0.43..0.64 rows=9 width=8)
20	Index Cond: (booking_id = bl.booking_id)

	QUERY PLAN text
1	CTE Scan on flights_totals (cost=14081839.16..15242007.67 rows=257815 width=26)
2	Filter: (departure_airport = 'ORD'::bpchar)
3	CTE flights_totals
4	-> GroupAggregate (cost=12663855.42..14081839.16 rows=51563045 width=30)
5	Group Key: bl.flight_id, f.departure_airport
6	-> Sort (cost=12663855.42..12792763.04 rows=51563045 width=18)
7	Sort Key: bl.flight_id, f.departure_airport
8	-> Hash Join (cost=1151589.89..2886328.12 rows=51563045 width=18)
9	Hash Cond: (b.booking_id = bl.booking_id)
10	-> Hash Join (cost=314001.30..846710.40 rows=16313907 width=22)
11	Hash Cond: (p.booking_id = b.booking_id)
12	-> Seq Scan on passenger p (cost=0.00..334787.07 rows=16313907 width=8)
13	-> Hash (cost=215591.02..215591.02 rows=5661302 width=14)
14	-> Seq Scan on booking b (cost=0.00..215591.02 rows=5661302 width=14)
15	-> Hash (cost=526548.02..526548.02 rows=17893566 width=12)
16	-> Hash Join (cost=26607.46..526548.02 rows=17893566 width=12)
17	Hash Cond: (bl.flight_id = f.flight_id)
18	-> Seq Scan on booking_leg bl (cost=0.00..310506.66 rows=17893566 width=8)
19	-> Hash (cost=15398.76..15398.76 rows=683176 width=8)
20	-> Seq Scan on flight f (cost=0.00..15398.76 rows=683176 width=8)

Execution plan with forced materialization: condition is not pushed down

Pros and Cons of CTEs

Pros:

- More manageable code
- Performance

Cons:

- Not reusable


Views: To Use or Not To Use

What Is a View?

A view is a database object that stores a query that defines a virtual table.

View can be used in SELECT statements the same way as tables.

Is this true?

View example

```
CREATE VIEW flight_stats AS
SELECT bl.flight_id,
       departure_airport,
      (avg(price))::numeric (7,2) AS avg_price,
       count(DISTINCT passenger_id) AS num_passengers
FROM booking b
JOIN booking_leg bl USING (booking_id)
JOIN flight f USING (flight_id)
JOIN passenger p USING (booking_id)
GROUP BY 1,2
```

	QUERY PLAN text
2	Group Key: bl.flight_id, f.departure_airport
3	-> Sort (cost=2574.94..2575.85 rows=367 width=18)
4	Sort Key: f.departure_airport
5	-> Nested Loop (cost=6.85..2559.30 rows=367 width=18)
6	-> Nested Loop (cost=6.41..1602.52 rows=128 width=26)
7	-> Index Scan using flight_pkey on flight f (cost=0.42..8.44 rows=1 width=8)
8	Index Cond: (flight_id = 222183)
9	-> Nested Loop (cost=5.99..1592.80 rows=128 width=22)
10	-> Bitmap Heap Scan on booking_leg bl (cost=5.55..511.20 rows=128 width=8)
11	Recheck Cond: (flight_id = 222183)
12	-> Bitmap Index Scan on booking_leg_flight_id (cost=0.00..5.52 rows=128 width=0)
13	Index Cond: (flight_id = 222183)
14	-> Index Scan using booking_pkey on booking b (cost=0.43..8.45 rows=1 width=14)

```
SELECT *
FROM flight_stats
WHERE flight_id=222183
```

	QUERY PLAN text
1	Hash Join (cost=12641030.68..14717215.42 rows=3663700 width=34)
2	Hash Cond: (bl.flight_id = f.flight_id)
3	-> GroupAggregate (cost=12630373.17..14053302.95 rows=51742901 width=30)
4	Group Key: bl.flight_id, f_1.departure_airport
5	-> Sort (cost=12630373.17..12759730.42 rows=51742901 width=18)
6	Sort Key: bl.flight_id, f_1.departure_airport
7	-> Hash Join (cost=1080689.84..2817439.82 rows=51742901 width=18)
8	Hash Cond: (b.booking_id = bl.booking_id)
9	-> Hash Join (cost=243101.21..775846.44 rows=16318749 width=22)
10	Hash Cond: (p.booking_id = b.booking_id)
11	-> Seq Scan on passenger p (cost=0.00..334860.49 rows=16318749 width=8)
12	-> Hash (cost=145003.98..145003.98 rows=5643298 width=14)
13	-> Seq Scan on booking b (cost=0.00..145003.98 rows=5643298 width=14)
14	-> Hash (cost=526548.06..526548.06 rows=17893566 width=12)
15	-> Hash Join (cost=26607.50..526548.06 rows=17893566 width=12)
16	Hash Cond: (bl.flight_id = f_1.flight_id)
17	-> Seq Scan on booking_leg bl (cost=0.00..310506.66 rows=17893566 width=8)
18	-> Hash (cost=15398.78..15398.78 rows=683178 width=8)
19	-> Seq Scan on flight f_1 (cost=0.00..15398.78 rows=683178 width=8)
20	-> Hash (cost=10052.84..10052.84 rows=48373 width=4)
21	-> Bitmap Heap Scan on flight f (cost=760.25..10052.84 rows=48373 width=4)
22	Recheck Cond: ((actual_departure >= '2023-08-01 00:00:00-05'::timestamp with time zone) AND (actual_departure <= '2023-08-14 00
23	-> Bitmap Index Scan on flight_actual_departure (cost=0.00..748.15 rows=48373 width=0)
24	Index Cond: ((actual_departure >= '2023-08-01 00:00:00-05'::timestamp with time zone) AND (actual_departure <= '2023-08-14 00

```
SELECT * FROM flight_stats fs
        JOIN
        (SELECT flight_id
         FROM flight
         WHERE actual_departure
              BETWEEN ‘2023-08-01’
                   AND ‘2023-08-14') fl
         ON fl.flight_id=fs.flight_id
```

Exec time: 10 min

Rewrite without the view

```
SELECT bl.flight_id,
       departure_airport,
      (avg(price))::numeric (7,2) AS avg_price,
       count(DISTINCT passenger_id) AS num_passengers
FROM booking b
JOIN booking_leg bl USING (booking_id)
JOIN flight f USING (flight_id)
JOIN passenger p USING (booking_id)
WHERE actual_departure between ‘2023-08-01' AND ‘2023-08-14'
GROUP BY 1,2
```

exec time: 3 min

Restrictions on flight are applied first

	QUERY PLAN text
1	GroupAggregate (cost=1773339.64..1874091.39 rows=3663700 width=30)
2	Group Key: bl.flight_id, f.departure_airport
3	-> Sort (cost=1773339.64..1782498.89 rows=3663700 width=18)
4	Sort Key: bl.flight_id, f.departure_airport
5	-> Hash Join (cost=654792.33..1223638.13 rows=3663700 width=18)
6	Hash Cond: (p.booking_id = b.booking_id)
7	-> Seq Scan on passenger p (cost=0.00..334860.49 rows=16318749 width=8)
8	-> Hash (cost=630294.22..630294.22 rows=1266969 width=26)
9	-> Hash Join (cost=390159.18..630294.22 rows=1266969 width=26)
10	Hash Cond: (b.booking_id = bl.booking_id)
11	-> Seq Scan on booking b (cost=0.00..145003.98 rows=5643298 width=14)
12	-> Hash (cost=368135.07..368135.07 rows=1266969 width=12)
13	-> Hash Join (cost=10657.51..368135.07 rows=1266969 width=12)
14	Hash Cond: (bl.flight_id = f.flight_id)
15	-> Seq Scan on booking_leg bl (cost=0.00..310506.66 rows=17893566 width=8)
16	-> Hash (cost=10052.84..10052.84 rows=48373 width=8)
17	-> Bitmap Heap Scan on flight f (cost=760.25..10052.84 rows=48373 width=8)
18	Recheck Cond: ((actual_departure >= '2023-08-01 00:00:00-05'::timestamp with time zone) AND (actual_departure <= '2023-08-14 00:00:00-05'
19	-> Bitmap Index Scan on flight_actual_departure (cost=0.00..748.15 rows=48373 width=0)
20	Index Cond: ((actual_departure >= '2023-08-01 00:00:00-05'::timestamp with time zone) AND (actual_departure <= '2023-08-14 00:00:00-05

Views with column transformation

```
CREATE VIEW flight_departure AS
SELECT
      bl.flight_id,
       departure_airport,
       coalesce(actual_departure, scheduled_departure)::date
                  AS  departure_date,
       count(DISTINCT passenger_id) AS num_passengers
 FROM booking b
 JOIN booking_leg bl USING (booking_id)
 JOIN flight f USING (flight_id)
 JOIN passenger p USING (booking_id)
GROUP BY 1,2,3
```

Not all columns are equal

```
SELECT flight_id,
       num_passengers
FROM flight_departure
WHERE flight =22183

---1 sec

SELECT flight_id,
       num_passengers
FROM flight_departure
WHERE departure_date= ‘2023-08-01’
--2 min
```

Execution plan

	QUERY PLAN text
1	Subquery Scan on flight_departure (cost=512344.83..521400.17 rows=258724 width=12)
2	-> GroupAggregate (cost=512344.83..518812.93 rows=258724 width=20)
3	Group Key: bl.flight_id, f.departure_airport, ((COALESCE(f.actual_departure, f.scheduled_departure))::da…
4	-> Sort (cost=512344.83..512991.64 rows=258724 width=16)
5	Sort Key: bl.flight_id, f.departure_airport
6	-> Nested Loop (cost=18858.24..484660.18 rows=258724 width=16)
7	Join Filter: (b.booking_id = p.booking_id)
8	-> Nested Loop (cost=18857.80..416906.41 rows=89471 width=36)
9	-> Hash Join (cost=18857.37..376334.93 rows=89471 width=28)
10	Hash Cond: (bl.flight_id = f.flight_id)
11	-> Seq Scan on booking_leg bl (cost=0.00..310506.66 rows=17893566 width=8)
12	-> Hash (cost=18814.67..18814.67 rows=3416 width=24)
13	-> Seq Scan on flight f (cost=0.00..18814.67 rows=3416 width=24)
14	Filter: ((COALESCE(actual_departure, scheduled_departure))::date = '2023-08-01'::d…
15	-> Index Only Scan using booking_pkey on booking b (cost=0.43..0.45 rows=1 width=8)
16	Index Cond: (booking_id = bl.booking_id)
17	-> Index Scan using passenger_booking_id on passenger p (cost=0.43..0.64 rows=9 width=8)
18	Index Cond: (booking_id = bl.booking_id)

Even worse:

```
SELECT flight_id
        FROM flight_departure
        WHERE departure_airport='ORD’
```

Why it is even worse?

We do not even need to use the view!

Runs 1 min 42 sec

Execution plan

	QUERY PLAN text
1	Subquery Scan on flight_departure (cost=886771.95..918054.44 rows=962538 width=4)
2	-> GroupAggregate (cost=886771.95..908429.06 rows=962538 width=20)
3	Group Key: bl.flight_id, f.departure_airport, ((COALESCE(f.actual_departure, f.scheduled_departure))::date)
4	-> Sort (cost=886771.95..889178.30 rows=962538 width=12)
5	Sort Key: bl.flight_id, ((COALESCE(f.actual_departure, f.scheduled_departure))::date)
6	-> Nested Loop (cost=9545.27..774662.60 rows=962538 width=12)
7	Join Filter: (b.booking_id = p.booking_id)
8	-> Nested Loop (cost=9544.84..520610.13 rows=334023 width=36)
9	-> Hash Join (cost=9544.41..367021.97 rows=334023 width=28)
10	Hash Cond: (bl.flight_id = f.flight_id)
11	-> Seq Scan on booking_leg bl (cost=0.00..310506.66 rows=17893566 width=8)
12	-> Hash (cost=9384.99..9384.99 rows=12753 width=24)
13	-> Bitmap Heap Scan on flight f (cost=243.26..9384.99 rows=12753 width=24)
14	Recheck Cond: (departure_airport = 'ORD'::bpchar)
15	-> Bitmap Index Scan on flight_departure_airport (cost=0.00..240.07 rows=12753 width=0)
16	Index Cond: (departure_airport = 'ORD'::bpchar)
17	-> Index Only Scan using booking_pkey on booking b (cost=0.43..0.46 rows=1 width=8)
18	Index Cond: (booking_id = bl.booking_id)
19	-> Index Only Scan using passenger_booking_id on passenger p (cost=0.43..0.64 rows=9 width=4)
20	Index Cond: (booking_id = bl.booking_id)

Without the view

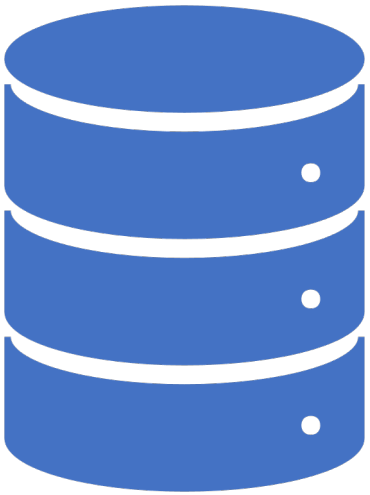
```
SELECT flight_id
FROM flight
WHERE
   departure_airport='ORD’
   AND flight_id IN
        (SELECT flight_id FROM booking_leg)

---3 sec
```

Why we ever want to use views?!

Code reuse!

Materialized Views



What is a materialized view?

A query definition

+

a table to store the results of the query at the time it is run.

`CREATE MATERIALIZED VIEW` saves both the query and the table

Materialized view

```
CREATE MATERIALIZED VIEW flight_departure_mv AS
SELECT
  bl.flight_id,
  departure_airport,
  coalesce(actual_departure,
                scheduled_departure)::date departure_date,
  count(DISTINCT passenger_id) AS num_passengers
FROM booking b
JOIN booking_leg bl USING (booking_id)
JOIN flight f USING (flight_id)
JOIN passenger p USING (booking_id)
GROUP BY 1,2,3
```

Indexes on the materialized view

```
CREATE UNIQUE INDEX flight_departure_flight_id
ON flight_departure_mv(flight_id);
--
CREATE INDEX flight_departure_dep_date
ON flight_departure_mv(departure_date);
--
CREATE INDEX flight_departure_dep_airport
ON flight_departure_mv(departure_airport);
```

Query Example

```
SELECT
  flight_id,
   num_passengers
FROM flight_departure_mv
WHERE departure_date_= '2023-08-01'
```

--**60 ms**

Refreshing materialized view

```
REFRESH MATERIALIZED VIEW flight_departure_mv
```

```
REFRESH MATERIALIZED VIEW CONCURRENTLY
flight_departure_mv
```

- Needs unique index
- Takes longer

There is no incremental refresh in PostgreSQL

When to create?

- How often does the data in the base tables change?
- How critical is it to have the most recent data?
- How often do we need to select this data (or rather how many reads per one refresh are expected)?
- How many different queries will use this data?

Good example – mview for yesterday's flights

```
CREATE MATERIALIZED VIEW flight_departure_prev_day AS
SELECT
   bl.flight_id,
   departure_airport,
   coalesce(actual_departure,
              scheduled_departure)::date departure_date,
    count(DISTINCT passenger_id) AS num_passengers
FROM booking b
JOIN booking_leg bl USING (booking_id)
JOIN flight f USING (flight_id)
JOIN passenger p USING (booking_id)
WHERE (actual_departure BETWEEN CURRENT_DATE -1 AND CURRENT_DATE)
      OR (actual_departure IS NULL AND scheduled_departure
              BETWEEN CURRENT_DATE -1 AND CURRENT_DATE)
GROUP BY 1,2,3
```

Not so good example

```
CREATE MATERIALIZED VIEW passenger_passport AS
SELECT
   cf.passenger_id,
   coalesce(max(CASE WHEN custom_field_name ='passport_num’
     THEN custom_field_value ELSE NULL END),'') AS passport_num,
   coalesce(max(CASE WHEN custom_field_name='passport_exp_date’
     THEN custom_field_value ELSE NULL END),'') AS passport_exp_date,
   coalesce(max(CASE WHEN custom_field_name ='passport_country’
     THEN custom_field_value ELSE NULL END),'') AS passport_country
FROM custom_field cf
GROUP BY 1
```

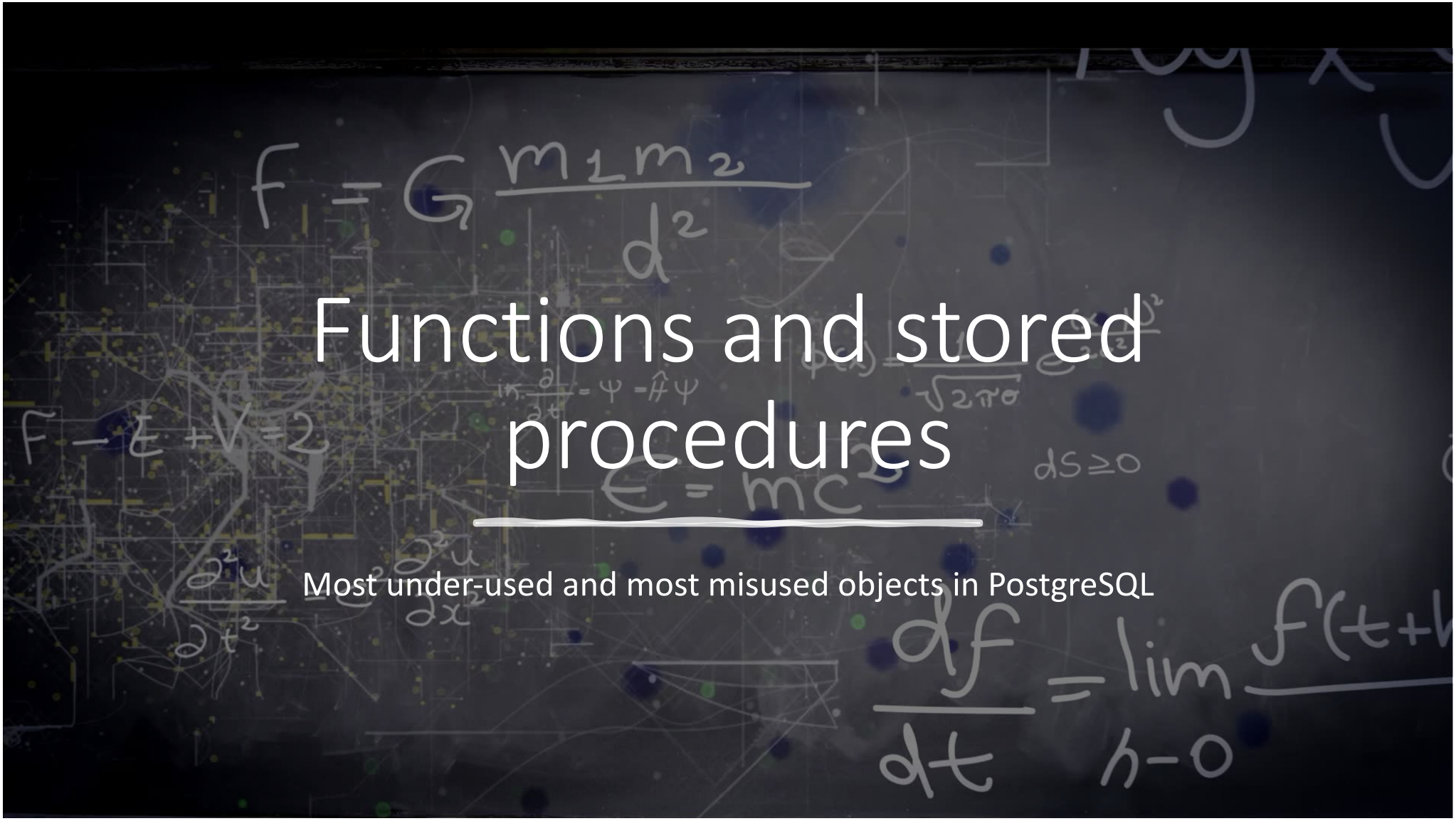
Why ”not so good”?

- Passport information can be added at any time => continues refresh
- MVIEW keeps growing

Materialized views should be optimized

Nobody is happy with a report running for six hours, no matter how infrequently

“Monday Morning Madness”



Functions and stored procedures

Most under-used and most misused objects in PostgreSQL

Functions and stored procedures

In contrast to other RDBMS, plpgsql functions:

- Are stored in the form of source code
- Are interpreted, not compiled
- No checks for existence of tables, columns or other functions
- No execution plan is saved
- Only when the execution path reaches a specific command, it is analyzed and *prepared statement* is created. If will be reused in the *same session*.
- That means, that you *might not discover* some errors in the conditional code

Functions are atomic

```
CREATE OR REPLACE FUNCTION num_passengers(p_flight_id int) RETURNS
integer
AS
$$BEGIN
RETURN (
        SELECT count(*) FROM booking_leg bl
                JOIN booking b USING (booking_id)
                JOIN passenger p
                USING (booking_id)
WHERE flight_id=p_flight_id);
```

Using in SELECT

With function:

```
SELECT flight_id,
   num_passengers(flight_id) AS num_pass
FROM flight f
   WHERE departure_airport='ORD'
   AND scheduled_departure
   BETWEEN ‘2023-07-05' AND ‘2023-07-13’
```

Execution time: 3.5 sec

Without function:

```
SELECT f.flight_id,
count(*) AS num_pass
FROM booking_leg bl
JOIN booking b USING (booking_id)
JOIN n passenger p
          USING (booking_id)
JOIN flight f USING (flight_id)
WHERE  departure_airport='ORD'
AND scheduled_departure BETWEEN
    ‘2023-07-05' AND ‘2023-07-13'
GROUP BY 1
```

Execution time: 900 ms

What if a function returns a set of records?

```
CREATE OR REPLACE FUNCTION boarding_passes_flight (p_flight_id int)
   RETURNS SETOF boarding_pass_record. AS $body$
BEGIN
RETURN QUERY
  SELECT pass_id,  bp.booking_leg_id,  flight_no,
     departure_airport,  arrival_airport,
     last_name ,  first_name ,
     seat,  boarding_time
  FROM flight f
  JOIN booking_leg bl USING (flight_id)
  JOIN boarding_pass bp USING(booking_leg_id)
  JOIN passenger USING (passenger_id)
  WHERE bl.flight_id=p_flight_id;
END;
$body$  LANGUAGE plpgsql;
```

Can we?...

```
SELECT
  departure_airport,
  scheduled_departure,
  actual_departure,
  boarding_time,
  last_name,
  first_name,
  seat
FROM boarding_passes_flight('2156') p
JOIN flight f USING (flight_no)
```

DON'T DO THAT!

```
CREATE OR REPLACE FUNCTION age_category (p_age int)
RETURNS TEXT language plpgsql AS
$body$
BEGIN
    RETURN (case
            WHEN p_age <= 2 then 'Infant'
            WHEN p_age <=12 then 'Child'
            WHEN p_age < 65 then 'Adult'
            ELSE 'Senior' END);
END; $body$;
```

```
SELECT passenger_id, age_category(age)
FROM passenger
LIMIT 5000000
```

Execution time 25 sec

```
SELECT passenger_id,
     CASE
          WHEN age <= 2 then 'Infant'
          WHEN age <=12 then 'Child'
          WHEN age < 65 then 'Adult'
          ELSE 'Senior’
     END from passenger LIMIT 5000000
```

Execution time 9 sec

How to do it the right way?

```
CREATE OR REPLACE FUNCTION age_category_dyn (p_age text)
   RETURNS text language plpgsql AS
$body$
BEGIN
    RETURN ($$CASE
                WHEN $$||p_age ||$$ <= 2 THEN 'Infant'
                WHEN $$||p_age ||$$<= 12 THEN 'Child'
                WHEN $$||p_age ||$$< 65 THEN 'Adult'
           ELSE 'Senior'
END$$);
END; $body$;
```

This function returns a part of code, not the value.

Next…

```
CREATE OR REPLACE FUNCTION passenger_age_category_select (p_limit int)
RETURNS setof passenger_age_cat_record
AS
$body$
BEGIN
RETURN QUERY
EXECUTE $$SELECT
          passenger_id,
          $$||age_category_dyn('age')||$$ AS age_category
FROM passenger LIMIT $$ ||p_limit::text
;
END;
$body$ LANGUAGE plpgsql;
```

This function executes generated SQL.

Success!

```
SELECT * FROM
passenger_age_category_select (5000000)
```

Execution time: 11 sec

+

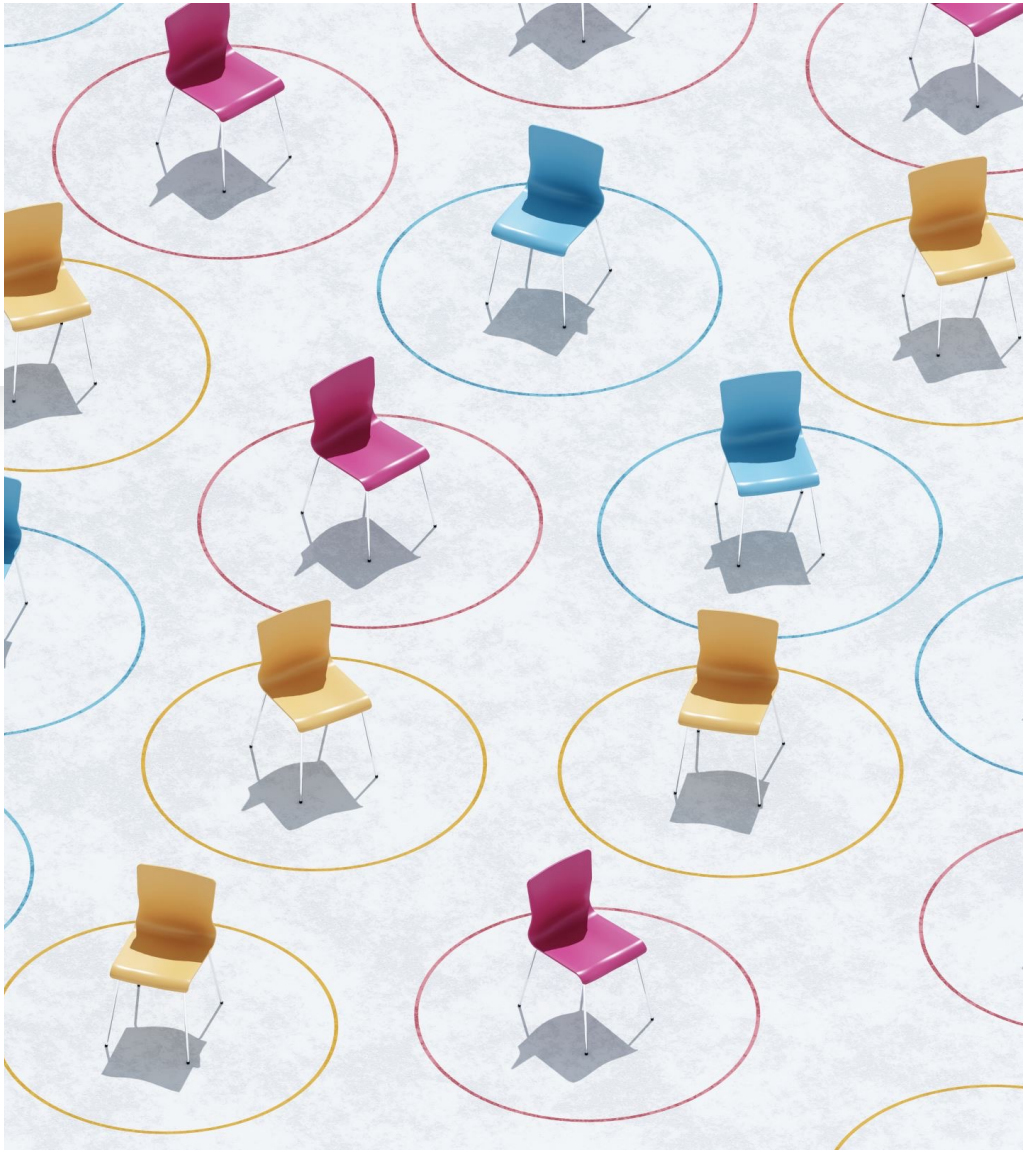
Code factoring

Summary

There are different ways to structure SQL queries.

Be aware of pros and cons of:

- TEMP tables
- CTEs
- Views
- Materialized views
- Functions
- Dynamic SQL




It’s difficult!
And Claude can’t do that!

LinkedIn:

GitHub:

https://github.com/hettie-d

Prairie Postgres

prairiepostgres.org

PG DATA 2026

2026.pg-data.org


Q&A Time!