

I Learned About PostgreSQL From That....



Google Cloud



About Me

Blog: shaneborden.com

X: @sborden76

LinkedIn: <https://www.linkedin.com/in/shaneborden/>



**Shane
Borden**

Database Black Belt
Solutions Consultant
Google Cloud



I Learned About Flying From That...

How decades of cockpit close-calls, flight discipline, and candid debriefs translate directly to PostgreSQL reliability.

The 80-Year Aviation Tradition: Since 1939, FLYING Magazine has published firsthand accounts from pilots who encountered unexpected emergencies, systemic failures, and near-misses. The core ethos: *admit mistakes openly so others survive them.*

The PostgreSQL Parallel: High-throughput databases operate at the edge of performance envelopes. When things go wrong, it's rarely a total mystery—it's usually an edge-case condition where assumptions collided with reality.

Core Engineering Lessons (Cockpit to Database):

1. Problem Solving Culture: Never assume. Systematically inspect execution plans, fetch sizes, and lock states before deploying under load.

2. Early Warning Indicators: Recognize subtle degradation (subtransaction locks, high water marks, vacuum lag) before cascading stalls occur.

3. Blameless Culture: Every production failure is a teaching case study that builds stronger architectural resilience across the team.



Aviation Rule #1: Aviate, Navigate, Communicate — then Optimize.

“Good judgment comes from experience, and a lot of that comes from bad judgment.”

— **Legendary Aviation Adage**

The guiding philosophy of FLYING's ILAFFT series & database postmortems

Agenda

Subtransaction Cliff

Exception blocks can drive severe locking contention under high concurrency

Partition Pruning Why Isn't It Happening

Sometimes changing the query changed the plan dramatically

Google Cloud

Ghost Writes and Vacuum

Simple constraint violations can cause vacuum work to increase dramatically

High Water Mark Contention + Vacuum

Releasing space sometimes can cause big problems

JDBC Fetch Sizes + Parallelism

Is the default good enough?
What is that impact?

Data Type Mismatch & Table Scans

Not all data types are equal

Agenda

Parameter Adjustments

Sometimes the default just isn't enough

Beware Correlated Columns

Columns with highly correlated indexed columns can affect the plan

Use a Connection Pooler

Applications tend to overuse connections



Subtransaction Cliff

Hidden Savepoint Allocation

```
FOR r IN SELECT * FROM batch_data LOOP
BEGIN
    INSERT INTO orders
        VALUES (r.id);
EXCEPTION
    WHEN OTHERS THEN
-- Allocates 1 sub-txid per iteration!
END;
END LOOP;
```

Each **BEGIN...EXCEPTION** block creates an explicit subtransaction (SubXID) under the hood.

What's Happening Under the Covers?

- **First 64 Sub-Transaction IDs are stored in memory**
- **Anything over 65 is spilled to disk (Until PG 17)**

Concurrent queries attempting visibility checks must now acquire global SLRU locks (**SubtransSLRUlock**), causing sudden 100% CPU lock contention.

What's the Impact?

- **More users, the slower it gets**
- **More subtransactions, more to manage**
- **More subtransactions, more WAL**



Subtransaction Cliff

What can lead to SubTransaction Usage?

- Calling Nested Procedures:
 - ● Free. No new XIDs are used. They share the parent's transaction.
- BEGIN ... END (No Exception Block):
 - ● Free. Just organization.
- COMMIT:
 - ● Expensive. Burns a main Transaction ID (finite resource, leads to Vacuum Freeze).
- EXCEPTION BLOCK:
 - ● Dangerous. Creates a subtransaction (performance killer).

The Fix

```
-- Atomic, set-based execution
INSERT INTO telemetry (id, metric, val)
  VALUES ($1, $2, $3)
ON CONFLICT (id) DO UPDATE
  SET val = EXCLUDED.val;
```

Native engine support resolves lock conflicts at the tuple level without allocating SubXID or touching SLRU.



Subtransaction Cliff

How Bad Can It Get???

- **Versions < 17 Limited to a 64 SLRU space**
- **SLRU Gets Worse with Concurrency**
- **Testing Confirmed Behavior**
 - Baseline / Single Exception / 48 Sub-Transaction / 128 Sub-Transaction

Test	Avg Time Per Exec	Min Exec Time	Max EXEC Time
1 User x 50 iterations	2137ms	2069ms	3477ms
4 Users x 50 iterations	3171ms	2376ms	4080ms
10 Users x 50 iterations	9041ms	4018ms	47880ms



“Ghost Writes” and Vacuum

Zero Rows Affected **CAN** Cause Dead Tuples

- **Deceptive Failure:** What seems like a simple constraint violation is **NOT** harmless.
- **Early Heap Write:** Inserts / Updates write the row to heap **before** verifying PK / FK existence.
- **Vacuum Overhead:** Simple errors in the PostgreSQL log can cause **lots of vacuum work**.

PostgreSQL Server Log Output:

```
ERROR: insert or update on table "fk_violation_test" violates foreign key constraint  
"fk_violation_test_product_id_fk"
```

```
DETAIL: Key (product_id)=(202) is not present in table "product".
```

```
ERROR: duplicate key value violates unique constraint "pk_violation_test_pkey"
```

```
DETAIL: Key (id)=(0) already exists.
```



“Ghost Writes” and Vacuum

Set Up the Test

```
/* Create the table: */

CREATE TABLE public.pk_violation_test (
    id int PRIMARY KEY,
    value numeric,
    product_id int,
    effective_date timestamp(3)
);

/* Insert some mocked up data */
INSERT INTO public.pk_violation_test VALUES (
    generate_series(0,10000),
    random()*1000,
    random()*100,
    current_timestamp(3));
```

Verify No Dead Tuples

```
/* Verify that there are no dead tuples: */
```

```
SELECT
```

```
    relname,
    n_live_tup,
    n_dead_tup
```

```
FROM
```

```
    pg_stat_all_tables
```

```
WHERE
```

```
    relname = 'pk_violation_test';
```

relname	n_live_tup	n_dead_tup
-----+-----+-----		
pk_violation_test	100001	0



“Ghost Writes” and Vacuum

A Simple PK Violation Causes the Dead Row Verify No Dead Tuples

```
/* Perform a simple insert: */
```

```
INSERT INTO pk_violation_test  
VALUES (0, 44.33893489873, 46, now());
```

```
ERROR:  duplicate key value violates unique  
constraint "pk_violation_test_pkey"  
DETAIL:  Key (id)=(0) already exists.  
Time: 1.292 ms
```

```
/* Verify that there are no dead tuples: */
```

```
SELECT  
    relname,  
    n_live_tup,  
    n_dead_tup  
FROM  
    pg_stat_all_tables  
WHERE  
    relname = 'pk_violation_test';
```

relname	n_live_tup	n_dead_tup
-----	+-----	+-----
pk_violation_test	100001	1



“Ghost Writes” and Vacuum

EXCEPTION Doesn't fix this!

```
BEGIN
  INSERT INTO .. VALUES ..;
EXCEPTION
  WHEN unique_violation THEN
    -- try an update
    UPDATE .. SET .. ;
END;
```

Use ON CONFLICT / MERGE when able

```
BEGIN
  INSERT INTO .. VALUES ..
  ON CONFLICT
    DO NOTHING | UPDATE | DELETE;
END;

OR

BEGIN
  MERGE INTO ...
  USING ... ON ...
  WHEN MATCHED THEN
    UPDATE ...
  WHEN NOT MATCHED THEN
    INSERT ...
END;
```



JDBC Fetch Sizes + Parallelism

Fetch Size Can Affect Performance

- Fetches All Rows At Once (Default)
 - Out of Memory Risk **HIGH**
- Other JDBC Drivers Vary
- Depending on Type of Application This Could be **GOOD** or **BAD**
- 100% OLTP? - Leave it **ALONE**
- Mixed / Analytical Workload / Memory Concerns?
 - **MAYBE** - Consider Setting fetchSize

Why No Parallelism?

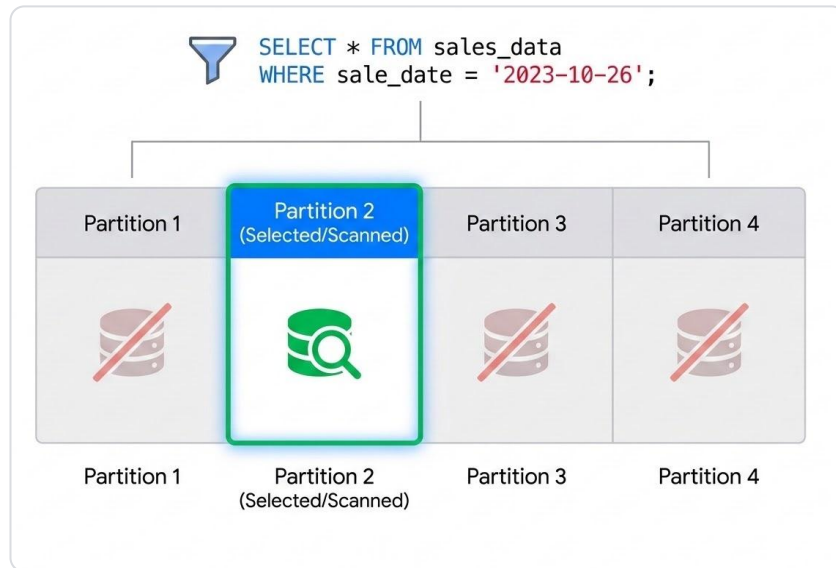
- Do Not Set maxRowSize
- Use LIMIT Instead
- LIMIT is Applied After All Parallel Nodes are Finished + Known at Planning Time
 - Parallel **ENABLED**
- When maxRowSize is Set, its, the Value is Unknown at Planning Time
 - Parallel **DISABLED**



Partition Pruning Why Isn't It Happening??

Placement of Partition Predicates are Important!

- The planner can only push quals down into a subquery, it cannot pull quals from a subquery into the outer query.
 - This is why CTEs won't help prune partitions
- If Partitions Cannot Be Pruned, Planning Time Can Be **ELEVATED**
 - Sometimes as much or more than Execution Time





Partition Pruning Why Isn't It Happening??

No Pruning

```
WITH top_emp AS (  
  SELECT  
    emp.id,  
    emp.hire_date  
  FROM  
    employees.employee_part emp  
    JOIN employees.salary sal  
      ON (sal.employee_id = emp.id)  
  WHERE  
    emp.hire_date = '1985-01-01'::date)  
SELECT  
  d.employee_id, d.department_id, d.hire_date  
FROM  
  top_emp e,  
  employees.department_employee_part d  
WHERE  
  e.hire_date = d.hire_date and  
  e.id = d.employee_id;  
Planning Time: 4.111 ms
```

Pruning

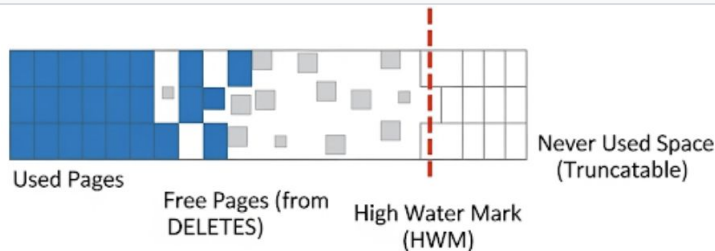
```
SELECT  
  employee_id, department_id, hire_date  
FROM (  
  SELECT  
    d.employee_id, d.department_id, d.hire_date,  
    count(1)  
  FROM  
    employees.employee_part emp  
    JOIN employees.salary sal  
      ON (sal.employee_id = emp.id)  
    JOIN employees.department_employee_part dept  
      ON (emp.hire_date = dept.hire_date  
          and emp.id = dept.employee_id)  
  WHERE  
    emp.hire_date = '1985-01-01'::date  
  GROUP BY  
    dept.employee_id, dept.department_id,  
    dept.hire_date) a;  
Planning Time: 0.460 ms
```



High Water Mark Contention + Vacuum

Freespace Truncation Consequences

- **Other DBMS:** Do not manage High Water Mark under normal operations
- **Exclusive Lock:** Truncation requires an ACCESS EXCLUSIVE lock
- **Table Workload:** Frequently inserted / deleted tables are most susceptible
- **Double Whammy:** Gets replayed on a read replica



Management & Mitigation

Disable automatic truncation on active tables:

- `ALTER TABLE [table name / partition name] SET (vacuum_truncate = false);`

Controlled truncation during maintenance windows:

- `VACUUM (verbose, truncate true) [schema].[table name];`

PostgreSQL Server Log Output:

```
INFO: table "large_table": truncated 302534 to 302233 pages
```



Data Type Mismatch Causes Table Scans

Too Many Combinations To Assume Conversion

Update App Types During Migrations

- Many “number” Data Types Across Engines:
 - PostgreSQL vs. SQL Server vs. Oracle
 - numeric(x) / numeric(x,y) / numeric / decimal
 - smallint / bigint / int
 - float / real / double precision / money
- Performance Impact:
 - Data type mismatches force implicit casts
 - Suppresses index usage -> **TABLE SCAN**

- For Migrations Make Sure You Update App Types Too
 - **NO CHANGE** Required
 - int vs. numeric
 - long vs. numeric
 - Always **REQUIRES CHANGE**
 - float vs. numeric



Parameter Adjustments

Some Defaults Just Are Not Enough

Memory & I/O

effective_io_concurrency

Default: **1**

maintenance_work_mem

Default: **64MB**

work_mem

Default: **4MB**

WAL & Connections

max_wal_size

Default: **1GB**

checkpoint_timeout

Default: **5min**

max_connections

Default: **100**

Planner & Lock Flags

enable_partitionwise_join

Default: **off**

enable_partitionwise_aggregate

Default: **off**

log_lock_waits

Default: **off**



Beware Correlated Columns

Highly Correlated Columns Can Affect Plans

- **Sometimes the Planner** Can be Tricked into Thinking an Index That Supports the Sort is Cheaper Than the Index That Supports a Filtered Lookup
- **Statistics Infer** that Filtered Values are Close Together Based on Order By Correlation:
 - Doesn't understand that the filtered column could be very far into the B-Tree structure

What's the Fix?

- **Look for Queries** with ORDER BY / LIMIT and High Variance in Execution Time
- **Invalidate the ORDER BY:**
 - **ORDER BY (order_column + 0)**
- **Use a CTE** to Enforce the Filter First
- **Create a Composite Index** to Support Both the Filter and the ORDER BY:
 - **CREATE INDEX** on (filter_col, order_col) ***
Best OPTION



Use Connection Pooling

Memory Wastage is REAL

- **Applications Tend to Oversize** Their App Side Connection Pools
- **Each Connection Uses Private Memory:**
 - ~ **7 to 10mb** Just to Connect
 - + **WORK_MEM**
 - + Factor for **hash_mem_multiplier**
- **Connection Memory is not Released** Until Connection Is Closed
- **Less Connection Memory = More Shared Buffers Memory**

Finding the Best Setting:

- **Understand Current Average** Idle vs Active Connections
- **2x - 3x of the CPU / vCPU Count** is a Good Place to Start for Max Pool Size
- **Understand How Many are Pooled Together (for PgBouncer)**
- **Alternatives?**
 - pgCat
 - pgDog
 - Pgpool-II
 - Odyssey

Thank you

Google Cloud

