

**WELCOME TO
VALENCIA**



SELECT 2026 FROM PGConf.EU;

PGCONF.EU 2026 · APP DEVELOPER TRACK

Same Problems, Different Defaults

What 25+ Years of SQL Server Taught Me When I Switched to PostgreSQL

Silvano Coriani

ABOUT THE SPEAKER

Who I am



Silvano Coriani

Microsoft

25+ YEARS WITH MICROSOFT SQL SERVER

Application development, data access, query performance, and production troubleshooting.

8 YEARS IN SQL CAT

Customer Advisory Team — working on some of the largest and most complex SQL Server projects on the planet.

DISTRIBUTED SYSTEMS DEVELOPER

Focused on the data access layer, query performance, and troubleshooting.

WORK CONTACTS

EMAIL scoriani@microsoft.com

GITHUB github.com/scoriani

Just two small examples of real-world stories encountered



THE INDEX MYSTERY

An 8 TB database. Very high resource utilization on the database server.

Roughly 80% of its indexes had never been used.



THE SLOW PAGE

**A web page that “felt” slow.
*It was quietly firing 63 separate queries to load one screen.***

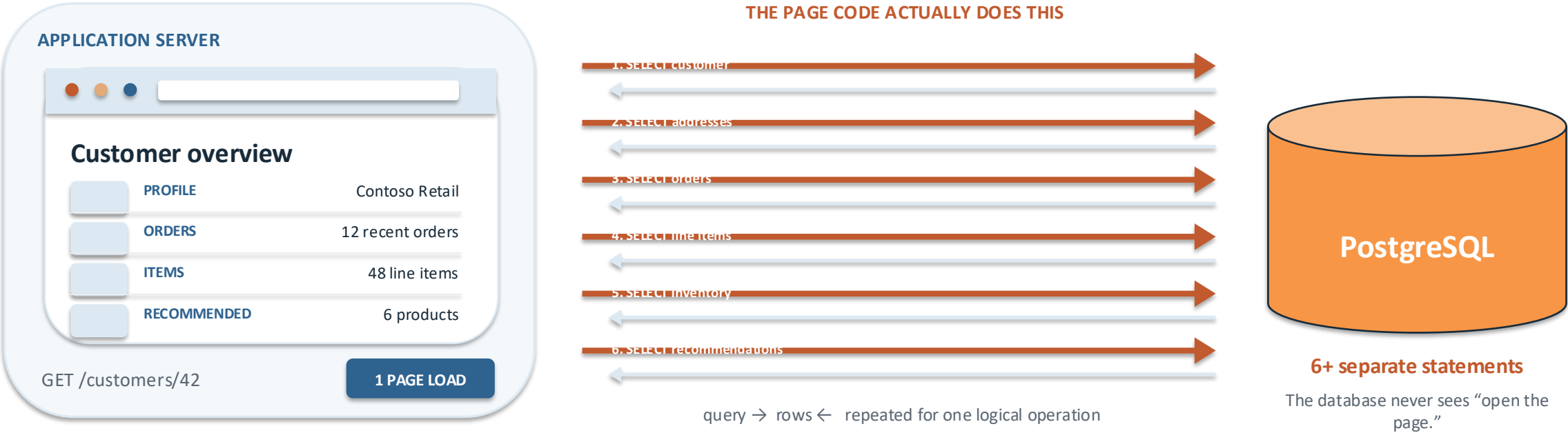
Both problems still exist in PostgreSQL. We'll come back to both by the end.

Agenda

- **Five patterns I hit with SQL Server that apply to PostgreSQL as well:**
 - Chatty applications
 - Accidental N+1 queries (flagship demo)
 - Parameterization surprises
 - Pagination that breaks down at scale
 - Schema changes that seemed safe until they weren't
- **One place the platforms are just genuinely different: resource governance**
- **The toolkit: reading plans, thinking about indexes, troubleshooting without blaming the database**

Chatty Applications

One user action, N round trips. The database never sees the request that's actually slow.



One page load. Many round trips. Each one pays network RTT + parse + plan.

EXPLAIN sees each fast statement—not the slow user experience created by all of them together.

Chatty Applications — batch reads too

Batching is not only for INSERTs: one logical page load can still be one network round trip.

1 ONE RESULT

Shape the page model in SQL

JOIN / LATERAL / aggregates

```
SELECT jsonb_build_object(  
  'profile', ...,  
  'orders', jsonb_agg(...));
```

Best when the data belongs to one view model.

2 MULTIPLE RESULTS · ONE ROUND TRIP

Batch independent SELECTs

Driver API — example: `NpgsqlBatch`

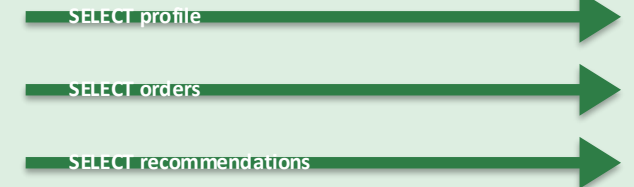
```
SELECT profile ...  
SELECT orders ...  
SELECT recommendations ...
```

Read each result with `NextResult()`.

3 LOW-LATENCY STREAM

Pipeline queries

`libpq pipeline mode` · PostgreSQL 14+



Send without waiting; read results in order.

WRITES STILL BATCH TOO multi-row INSERT · UNNEST arrays · COPY

Different default: PostgreSQL functions/procedures do not directly emit arbitrary result sets like a SQL Server stored procedure. `refcursor` can emulate this, but it is transaction-bound and requires `FETCH` handling—prefer a driver batch or one structured result.

Pooling moves cost — it does not remove it

A warm connection is faster for the client. Every warm PostgreSQL connection is still a backend process on the server.

CLIENT-SIDE POOL · IN EACH APP INSTANCE

Optimize request latency

Reuse an authenticated, TLS-ready connection instead of creating one for every request.

- **Benefit:** lowest checkout latency; no extra network hop.
- **Cost:** every pool slot remains one PostgreSQL backend — even while idle.
- **Scale effect:** app instances × pool size = total server connections.

8 app instances × 10 pooled connections = 80 PostgreSQL backends

SERVER-SIDE POOLER · SHARED BY ALL APPS

Protect database capacity

PgBouncer or PgCat multiplexes many client sessions onto a bounded set of real PostgreSQL backends.

- **Benefit:** caps backend memory, scheduling overhead, and concurrency.
- **Cost:** one extra hop; excess work may queue instead of opening more backends.
- **Design effect:** pooling mode limits which session semantics the app can use.

80 client sessions → pooler → 20 PostgreSQL backends

Use both: modest pools close to the app for latency, feeding a bounded shared pool for server survival.

Pooling mode is an application contract

More multiplexing means fewer backend processes — and fewer session-level assumptions your application can safely make.

SESSION

Backend held for the client session

LOW MULTIPLEXING · MAXIMUM COMPATIBILITY

WHEN TO USE

Use when session state is part of the design.

APPLICATION DESIGN

Temp tables, LISTEN/NOTIFY, session SET, prepared statements, and session advisory locks behave normally.

Trade-off: backend count still follows concurrent client sessions.

TRANSACTION

Backend held only for one transaction

HIGH MULTIPLEXING · USUAL WEB/API CHOICE

WHEN TO USE

Use for stateless request/transaction workloads.

APPLICATION DESIGN

Keep state inside the transaction; prefer SET LOCAL; assume the next transaction may run on another backend.

Trade-off: session-scoped features need redesign or explicit pooler support.

STATEMENT

Backend released after every statement

MAX MULTIPLEXING · RARELY PRACTICAL

WHEN TO USE

Use only for independent autocommit statements.

APPLICATION DESIGN

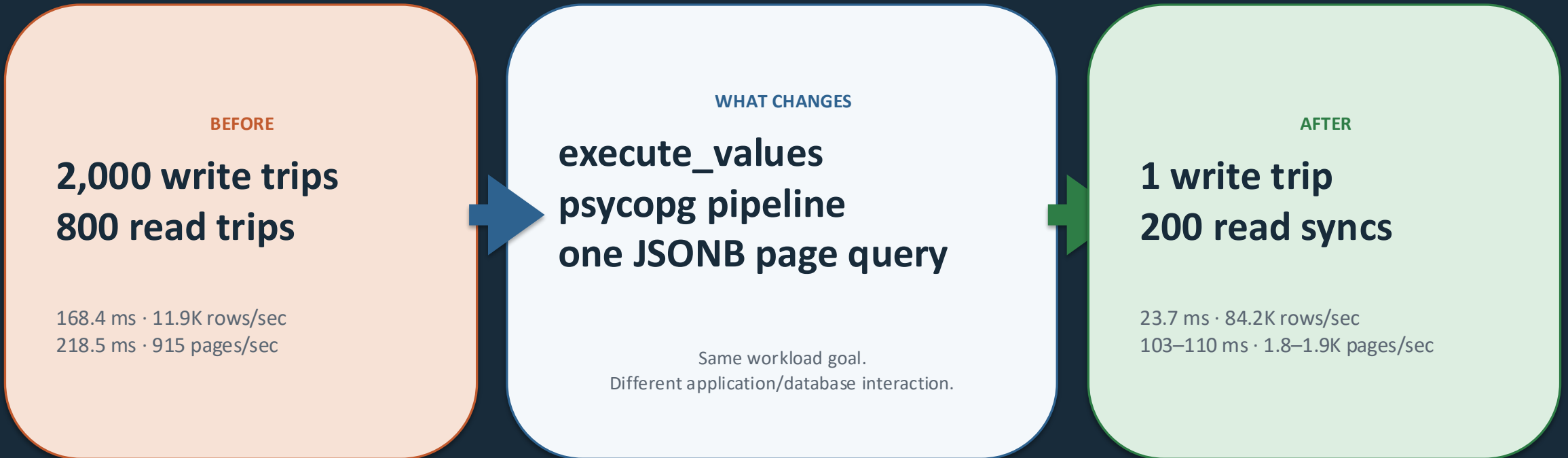
Multi-statement transactions and session state are unavailable; each statement must stand alone.

Trade-off: strongest restriction on application behavior.

Decision rule: need session semantics? **SESSION** · stateless request/transaction? **TRANSACTION** · isolated autocommit only? **STATEMENT** (rare)

Live demo: batch writes and reads

Same logical work. Fewer network synchronization points. Higher throughput.



EXPECTED RESULT 7.1x write throughput · about 2x read throughput

```
bash 01_chatty_applications/run.sh
```

WATCH: round trips · wall time · rows/pages per second

PATTERN 2 OF 5 · FLAGSHIP DEMO

Accidental N+1 Queries

Remember that 63-query page?

“Here's what was actually happening — and here it is at a scale where the problem becomes undeniable.”

The schema

customers

```
id  BIGSERIAL PK
name TEXT
region TEXT
```

orders

```
id  BIGSERIAL PK
customer_id BIGINT REFERENCES
customers
created_at  TIMESTAMP
total  NUMERIC

INDEX (customer_id, created_at
DESC)
```

Seed: 5,000 customers, 500,000 orders (~100 orders/customer, spread over the last 180 days).

Use case: “Fetch each customer with their most recent order in the last 30 days.”

The naive version

Pseudo-code (Django-style ORM)

```
customers = Customer.objects.all()
for c in customers:
    latest =
    c.orders.filter(created_at__gte=now()-30d)
        .order_by('-created_at').first()
```

1 query for the customer list
+ 1 query PER customer for their latest order
= 5,001 total round trips for 5,000 customers



Sub-millisecond in isolation. EXPLAIN on one of these looks perfect — the cost only shows up when you count executions.

In isolation, it looks perfect — that's the trap

The per-customer query

```
SELECT * FROM orders
WHERE customer_id = 42 AND created_at >= NOW() - INTERVAL '30 days'
ORDER BY created_at DESC LIMIT 1;
```

EXPLAIN (ANALYZE, BUFFERS)

```
Limit  (cost=0.43..4.68 rows=1 width=30) (actual time=0.020..0.021 rows=1 loops=1)
->  Index Scan using idx_orders_customer_date on orders
      (actual time=0.020..0.020 rows=1 loops=1)
```

Execution Time: 0.035 ms

0.035 ms. Fast. Correct index used. Nothing here tells you this runs 5,000 times per page load.

The measured cost of 5,001 round trips

5,001 round trips

~234.5 ms
total wall time

vs.

(preview — fix comes next)

~0.047 ms
per statement overhead

This is on localhost — no network between app and database. In production, each of those round trips crosses a real network. Even a fast same-region hop is often 0.5–2 ms of pure latency. At just 1 ms of round-trip latency, 5,000 sequential queries costs 5+ seconds — before the database does any meaningful work at all.

The fix: set-based rewrite with LATERAL

```
                SELECT c.id, c.name, o.id AS order_id, o.created_at, o.total
FROM customers c
LEFT JOIN LATERAL (
    SELECT o.id, o.created_at, o.total
    FROM orders o
    WHERE o.customer_id = c.id
        AND o.created_at >= NOW() - INTERVAL '30 days'
    ORDER BY o.created_at DESC
    LIMIT 1
) o ON true;
```

*One query. The subquery runs once per customer inside a single plan — the database does the looping, not the network.
(SQL Server: this is what CROSS APPLY / OUTER APPLY does.)*

One query, one plan

```
EXPLAIN (ANALYZE, BUFFERS)
```

```
Nested Loop Left Join (actual time=0.025..17.879 rows=5000 loops=1)
```

```
  Buffers: shared hit=20037
```

```
    -> Seq Scan on customers c (actual time=0.005..0.213 rows=5000 loops=1)
```

```
    -> Limit (actual time=0.003..0.003 rows=1 loops=5000)
```

```
      -> Index Scan using idx_orders_customer_date on orders o
```

```
        (actual time=0.003..0.003 rows=1 loops=5000)
```

```
Execution Time: 18.011 ms
```

1 round trip. ~19 ms total (18.0 ms execution + ~1 ms client overhead).

Before / after

5,001 round trips

~234.5 ms

1 round trip (LATERAL)

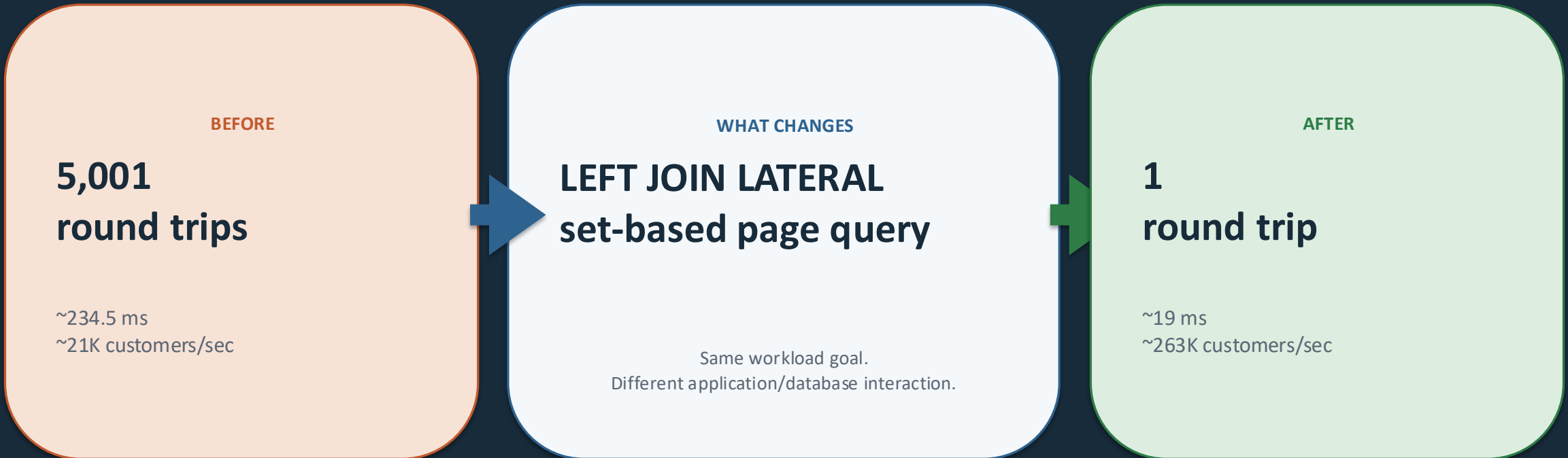
~19 ms

~12x faster — on localhost, with zero network latency. In production, the gap is dramatically larger.

“That's the 63-query page, solved the same way: one query per user action, not one query per row.”

Live demo: 5,001 queries become one

The individual query is fast. The repeated network pattern is not.



EXPECTED RESULT about 12x lower elapsed time and higher throughput

```
bash 02_n_plus_1/run.sh
```

WATCH: round trips · total wall time · customers per second

Parameterization Surprises

A parameterized query – the kind almost every ORM issues by default

```
PREPARE get_orders(bigint) AS SELECT * FROM orders WHERE customer_id = $1;
```

Data shape: one huge account, thousands of small ones

SKEWED VALUE

customer_id = 1

400,100 rows

44.46% of all rows

Examples of typical parameter values (4 of 4,999)

customer_id = 2

100 rows | 0.011%

customer_id = 817

100 rows | 0.011%

customer_id = 2,604

100 rows | 0.011%

customer_id = 5,000

100 rows | 0.011%

ANALYZE puts customer_id=1 in the MCV list — but a generic plan is built before the bound value is known, so it can't consult that MCV frequency at all.

Generic plan vs. custom plan — same value, same statement

GENERIC plan (forced)

```
Index Scan using idx_orders_customer_date
(cost=0.42..563.68 rows=183 width=56)
(actual ... rows=400100 loops=1)
Planning Time: 14.015 ms
Execution Time: 179.326 ms
```

CUSTOM plan (forced)

```
Seq Scan on orders
(cost=0.00..21488.00 rows=400500 width=56)
Filter: (customer_id = '1'::bigint)
Planning Time: 0.063 ms
Execution Time: 48.975 ms
```

179.3 ms (rows=183 estimated, 400,100 actual — wrong scan) vs. 49.0 ms (correctly estimated) — ~3.7x, from plan choice alone. Same data, same statement, same parameter value.

In practice: PostgreSQL tries custom plans for the first 5 executions, then switches to generic if its average cost looks competitive — forced here via `plan_cache_mode` for a repeatable demo.

“Postgres isn't wrong here — the surprise is the trade-off you signed up for by reusing a plan.”

Same symptom class, different mechanism

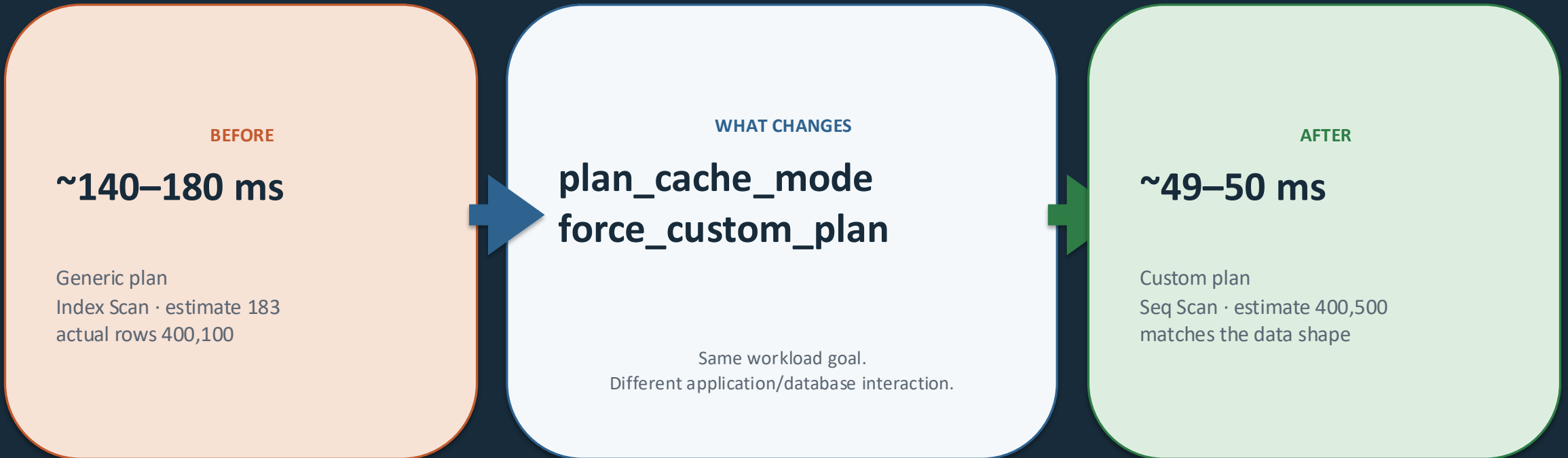
“This smells like parameter sniffing. The mechanism doesn't map 1:1 — PostgreSQL's lever is `plan_cache_mode`, not `OPTION (RECOMPILE)` or a plan guide.”

- Check `pg_prepared_statements` to see what your driver is actually holding open per session
- JDBC: `prepareThreshold=0` disables server-side prepare; `psycpg3`: `prepare_threshold`; `asyncpg`: `statement_cache_size=0` — every driver exposes this differently
- If a query's plan depends heavily on parameter value, force it explicitly: `SET plan_cache_mode = force_custom_plan` for that session/statement

Know whether your driver/ORM is issuing long-lived server-side prepared statements — that's the switch that decides which of these two plans you get.

Live demo: generic plan vs. custom plan

Same statement. Same skewed value. Only the planning context changes.



EXPECTED RESULT about 3x lower execution time for the skewed value

```
bash 03_parameterization/run.sh
```

WATCH: estimated vs. actual rows · scan type · execution time

Pagination That Breaks Down at Scale

OFFSET/LIMIT pagination works fine at page 1 — and gets progressively, then catastrophically, slower as users page deeper.

Page 1 (OFFSET 0)



0.096 ms

Deep page (OFFSET 400,000)



157.234 ms

gray = scanned & discarded to satisfy OFFSET · colored = the 20 rows actually returned

Cost grows linearly with OFFSET — the planner has no pushdown that skips counted-but-discarded rows, no matter how well the sort column is indexed.

Measured evidence — same index, only the offset changes

Query	Buffers read	Execution time
LIMIT 20 OFFSET 0	shared hit=20 read=3	0.096 ms
LIMIT 20 OFFSET 400000	shared hit=399907 read=1532	157.234 ms

“Same query. Same index. Same 20 rows returned. Paging to a deep page costs ~1,600x more buffer reads than page one — the database is faithfully doing exactly what OFFSET tells it to do.”

SQL Server contrast: OFFSET ... FETCH NEXT has the exact same cliff — this isn't even a PostgreSQL-specific gotcha.

The fix: keyset (seek) pagination

```
SELECT id, customer_id, created_at, total FROM orders
WHERE (created_at, id) < ($1, $2)  -- last row's values from the previous page
ORDER BY created_at DESC, id DESC LIMIT 20;
```

Index seeks directly — no scan-and-discard



row-wise comparison $(created_at, id) < (...)$ is directly index-able on a matching composite b-tree — no OR-of-ANDs rewrite needed

At the same depth: 1.369 ms vs. 157.234 ms — ~115x faster, and flat no matter how deep the user pages.

“If your pagination UI only ever needs Next/Previous, keyset pagination should be the default, not the optimization you reach for after the incident.”

Live demo: stop counting past 400,000 rows

Same depth. Same 20 rows. A seek replaces scan-and-discard.

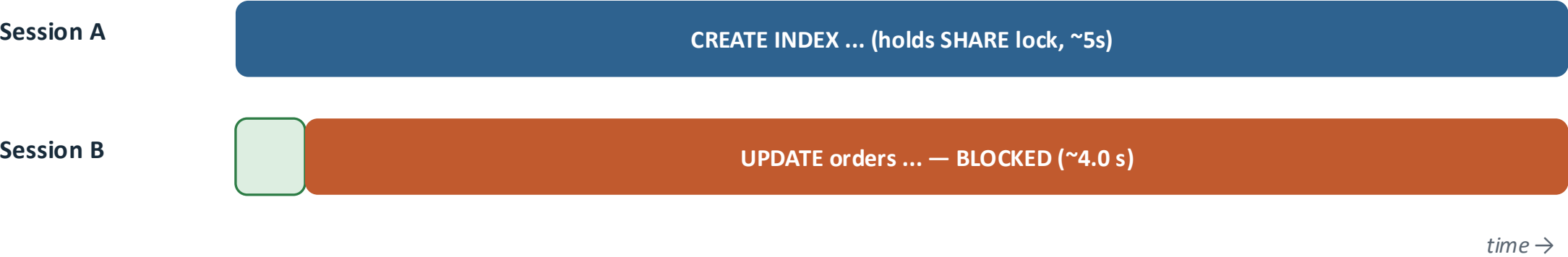


```
bash 04_pagination/run.sh
```

WATCH: execution time · buffers · flat cost at depth

CREATE INDEX looks read-only. It isn't.

A plain CREATE INDEX takes a SHARE lock — blocking writes for the entire build.



`pg_stat_activity` snapshot while blocked — `pg_blocking_pids()` shows exactly who

pid	wait_event_type	wait_event	query	blocked_by
70316	Timeout	PgSleep	BEGIN; LOCK TABLE ... pg_sleep(5)	{}
70320	Lock	relation	UPDATE orders SET total=... WHERE id=1	{70316}

For long builds in production: `pg_stat_progress_create_index` shows live phase + block progress.

The fix: CREATE INDEX CONCURRENTLY

CREATE INDEX (plain)

**Concurrent UPDATE:
blocked ~4.0 s**

CREATE INDEX CONCURRENTLY

**Concurrent UPDATE:
completes in 0.036 s**

SHARE UPDATE EXCLUSIVE lock permits concurrent reads AND writes — no blocking at all.

Trade-off, stated honestly: CONCURRENTLY runs two full table scans and waits out concurrent transactions to build a consistent index without blocking — slower overall, can't run inside a transaction block, and can fail partway through, leaving an INVALID index to drop and retry. “Non-blocking” is not “free.”

The subtler trap: ADD COLUMN ... DEFAULT ...

Since PG 11, “adding a column is basically instant if you don't populate real data” — mostly true, but the boundary isn't “does the default look like a function call.”

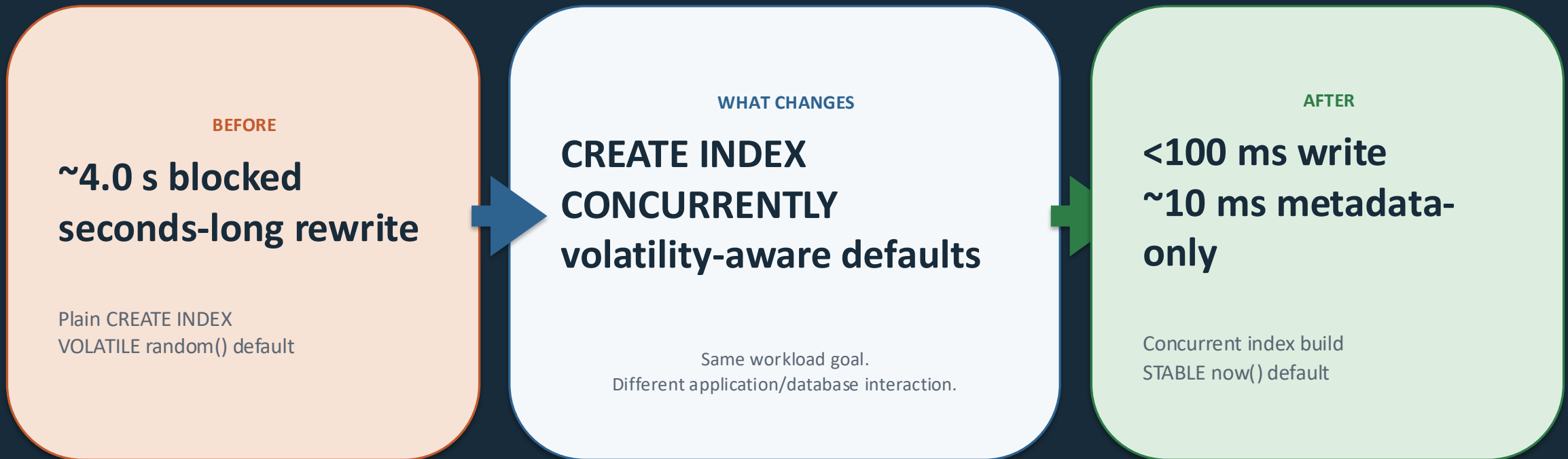
Statement	Time	Why
DEFAULT 'completed'	3.5 ms	constant – metadata-only
DEFAULT now()	1.3 ms	STABLE – computed once, metadata-only
DEFAULT random()	1,691.9 ms	VOLATILE – full table rewrite

“now() and random() are both function calls — and behave completely differently. The deciding factor is volatility (STABLE vs. VOLATILE), a classification most app developers never had to think about.”

“Before you ship a migration, ask: does this need to lock, rewrite, or scan the whole table?”

Live demo: safe-looking DDL, visible consequences

Two before/after pairs: write blocking and full-table rewrites.



EXPECTED RESULT write latency stays low · metadata-only change avoids rewrite

```
bash 05_schema_changes/run.sh
```

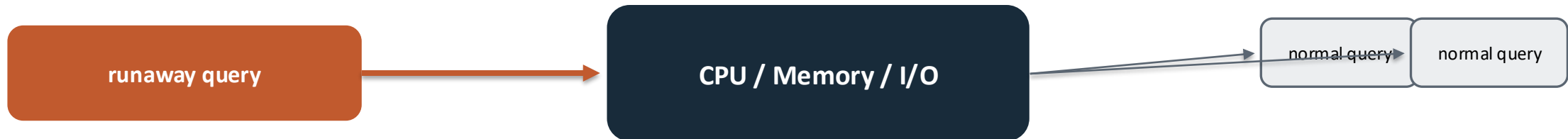
WATCH: blocked-write duration · ALTER TABLE timing

No Built-in Resource Governance

Everything so far was “same problem, different mechanism.” This one is genuinely different.

One missing join predicate, one accident

```
SELECT count(*) FROM orders o1 JOIN orders o2 ON o1.total < o2.total;  
-- Nested Loop (cost=0.42..3,093,927,969.67 rows=112,500,000,000 width=0)
```



SQL Server has Resource Governor — classifier functions route sessions into workload groups with CPU/memory/IOPS caps. Vanilla PostgreSQL has no admission control layer: `work_mem` is granted per sort/hash node, not capped per session, so concurrent expensive operations stack additively with no server-side ceiling.

The fix moves into the app

```
SET statement_timeout = '2000ms';  
SELECT count(*) FROM orders o1 JOIN orders o2 ON o1.total < o2.total;  
-- ERROR: canceling statement due to statement timeout (session stays healthy)
```

- statement_timeout / lock_timeout / idle_in_transaction_session_timeout — blunt but effective backstops
- Connection pool sizing + workload separation (ties back to Pattern 1)
- Circuit breakers / backoff in the application when the database signals distress
- Workload isolation via read replicas for reporting/analytics vs. OLTP

“On SQL Server, misbehaving workloads were sometimes a database configuration problem. On PostgreSQL, they're an application architecture problem from day one — so you might as well design for it from the start.”

Live demo: put a ceiling on the runaway query

The goal is containment: bounded failure instead of unbounded contention.



EXPECTED RESULT worst-case execution time becomes predictable

```
bash 06_resource_governance/run.sh
```

WATCH: cancel time · error handling · successful follow-up query

Reading execution plans — the skill underneath every demo today

EXPLAIN (ANALYZE, BUFFERS) — estimated vs. actual rows, scan types, join methods, loop counts. The muscle transfers completely from SQL Server, different syntax and operator names.

- Pattern 2 (N+1): loops=5000 inside one plan vs. 5,001 separate round trips
- Pattern 3 (parameterization): estimated rows=183 vs. actual rows=400,100
- Pattern 4 (pagination): shared hit=20 vs. shared hit=399,907
- Pattern 6 (resource governance): cost=...3,093,927,969.67 before you ever hit execute

Find the row where estimated and actual diverge, or where loops multiplies a small number into a big one. That's the one skill this whole talk has been teaching, one demo at a time.

Thinking about indexes — resolving the opening anecdote

“Remember the 8 TB database with roughly 80% unused indexes? Here's the smaller, tamer version of exactly how I'd have found that in PostgreSQL.”

index_name	idx_scan	index_size
orders_pkey	3	19 MB
idx_orders_created_at	4	30 MB
idx_orders_total	196	21 MB
idx_orders_customer_date	10,008	31 MB

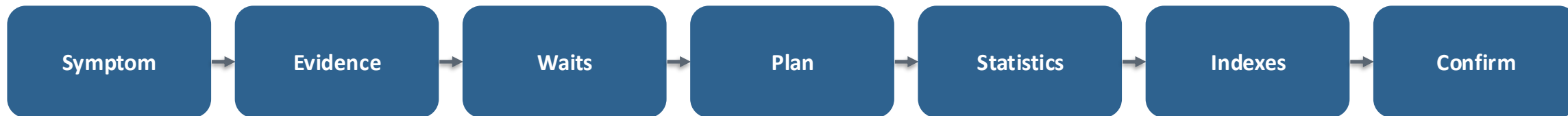
900,000 inserts.
All 4 indexes updated on every one.

table alone: 80 MB
all indexes: 102 MB

indexes cost more than the table they index.

“Index for the workload you actually have, not the schema you happen to have. Check idx_scan before you add the fifth index — or before you remove the one nobody asked about.”

Investigate, don't guess



“What NOT to do” (still true here):

- Don't rebuild all indexes blindly.
- Don't bump `work_mem` / `shared_buffers` without evidence of what's actually constrained.
- Don't throw hardware at a bad query shape (Pattern 2 doesn't get better with a bigger box — it gets 5,000x better with one fewer round trip).
- Don't blame the database before checking EXPLAIN.

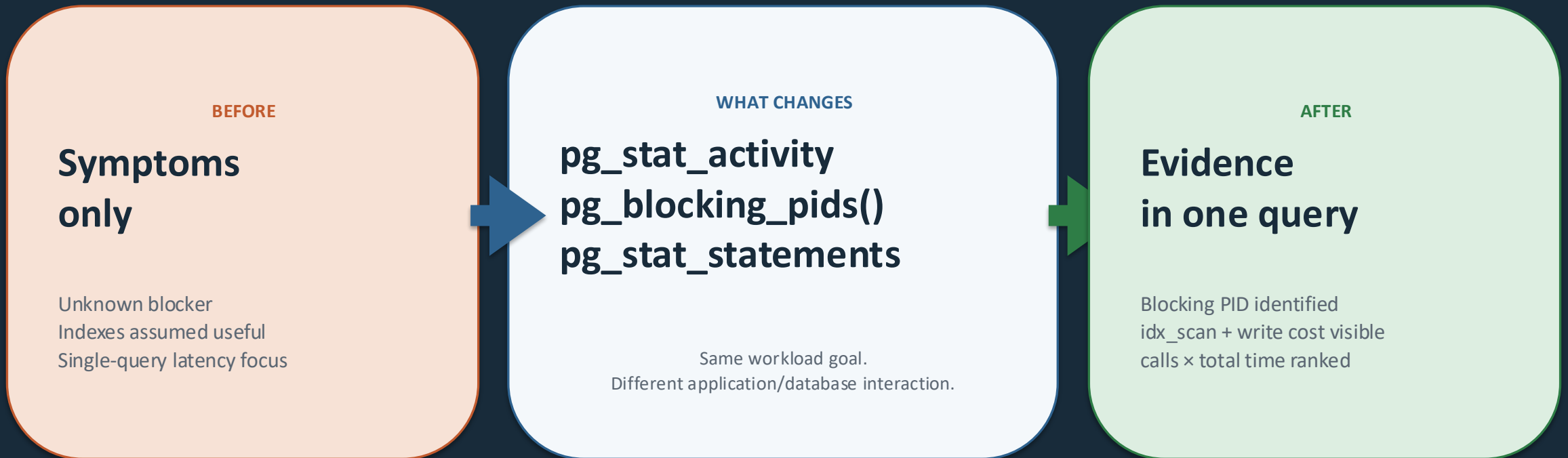
“Waits” in practice: `wait_event_type` in `{Lock, LWLock, BufferPin, IO, IPC}` — name the category before you guess the cause.

SQL Server ↔ PostgreSQL — the Rosetta Stone

What you need	SQL Server	PostgreSQL
Historical query performance	Query Store	pg_stat_statements
Real-time activity	DMVs, sp_whoisactive	pg_stat_activity, pg_locks
Wait statistics	sys.dm_os_wait_stats	pg_stat_activity.wait_event
Deep diagnostic tracing	Extended Events	auto_explain
Blocking / lock contention	sp_who2, blocking-leader queries	pg_locks, pg_blocking_pids()
Diagnostic script libraries	Glenn Berry's scripts	pgHero, community scripts
Plan visualization	SSMS graphical plan	explain.dalibo.com, pgAdmin

Live demo: replace guesses with evidence

This is a diagnostic improvement: shorten time to root cause before changing anything.



EXPECTED RESULT faster diagnosis · fewer blind fixes · safer tuning

```
bash 07_toolkit/run.sh
```

WATCH: time to evidence · blocker identity · cumulative workload cost

Five patterns, one thread

- Chatty applications
- Accidental N+1 queries
- Parameterization surprises
- Pagination that breaks down at scale
- Schema changes that seemed safe until they weren't

“Every single one of them existed on SQL Server too. What changed when I switched wasn't the problem. It was the defaults, the tools, and — in the case of resource governance — one genuine architectural difference that moved a fix out of the database and into my application.”

THE TAKEAWAY

“This was never a talk about which database is better. It's a talk about what happens to your instincts when you move from one set of defaults to another — and how to rebuild the ones that don't survive the trip.”

The Top 10 (one slide, take a photo)

1. Use connection pooling everywhere.
2. Design for transient failures and retries.
3. Think in sets, not loops.
4. Index for the workload you have, not the schema you drew.
5. Keep transactions short.
6. Measure before you tune — EXPLAIN first, opinions second.
7. Learn to read plans and waits; they outlive any one database engine.
8. Establish observability from day one, not after the incident.
9. Design for horizontal scale before you need it.
10. Assume the resource ceiling is your job to enforce, not the engine's.

THANKS FOR LISTENING!

ANY QUESTIONS?



Help me and the conference
get better!

Give feedback for this
session and the conference.



PostgreSQL Conference
Europe 2026

Valencia

Oct 20th - 23rd