- Intro- 5min
- Go - 5min
  - Cancel.timeout, cancel context
  - ORM flex
- Tx - 15min
  - idle transaction - xid
  - Long transaction
  - Subtransaction - txid
- Patterns - 10min
  - UPDATE, fillfactor
  - Temp tables
  - DML CTE, avoiding network hops, idempotent DMLs ON CONFLICT
  - Avoid session variables - pgbouncer
  -

Yoann La Cancellera

Lead database performance engineer - support

Avoiding pitfalls with Postgres and Go



- **Yoann La Cancellera**

  Support - database performance engineer.

- Percona is a 100% open-source editor.
  Historically MySQL, now mongo, redis and postgres as well. We do pg_tde

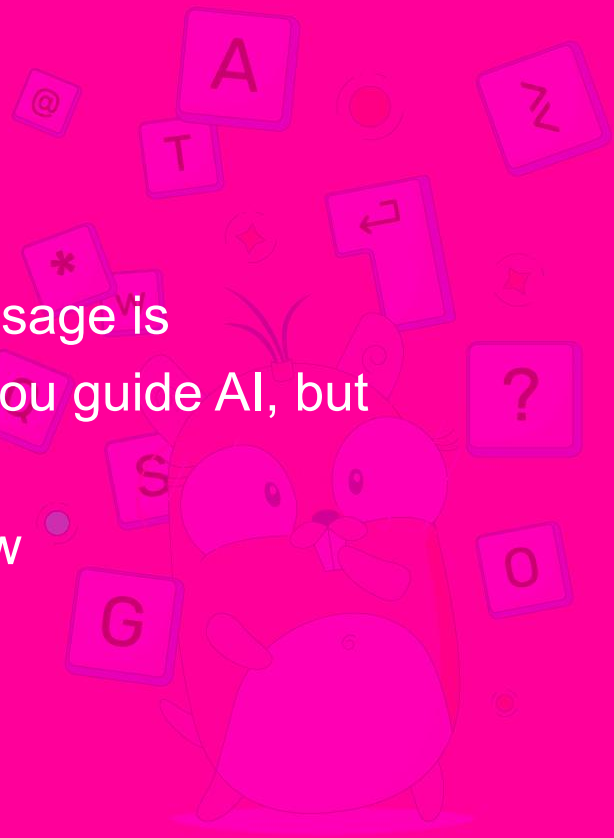
- Helping DBAs and devs on terabytes databases often in urgency and always without context
  - Bugs, slow queries, schema patterns, HA, architecture
  - Any type of env and constraints



INTRODUCTION

Goal:

- Not telling you what the perfect Go usage is
- AI-neutral: the takeaways can help you guide AI, but not necessarily
- Giving you my database point of view



Golang pitfalls

Easy stuff first!

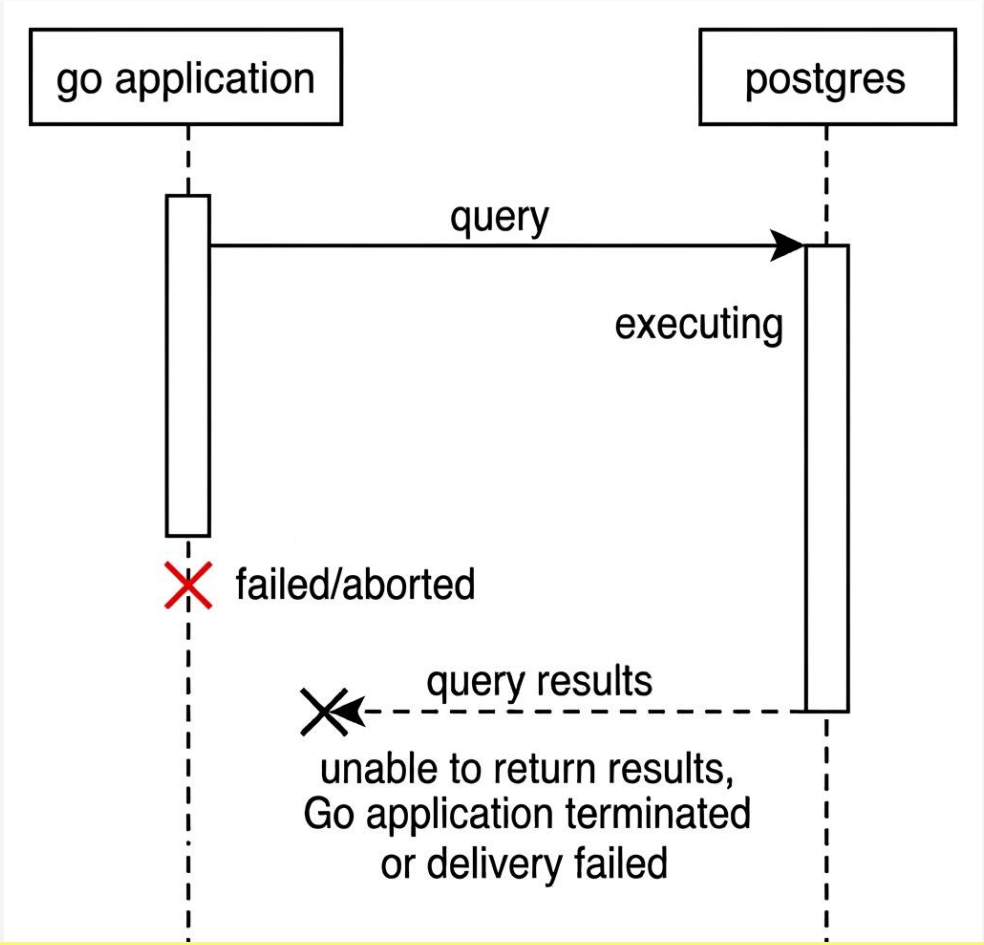
- Not using context
- Being rigid on ORMs



Not using context

- Golang will be fine. (right?)
- But **slow queries will stay running on postgres**
- The app should define what's "too slow"

Postgres workaround:
client_connection_check_interval



Not using context

- Golang will be fine. (right?)
- But slow queries will stay running on postgres
- **The app should define what's "too slow"**

Postgres workaround:
client_connection_check_interval

```
rows, err = db.Query("select slow")

rows, err = db.QueryContext(
    context.TODO(),"select"
)
rows, err = db.QueryContext(
    context.Background(), "select"
)

queryCtx, cancel := context.WithTimeout(
    ctx, 5*time.Second
)
defer cancel()
rows, err = db.QueryContext(queryCtx, "select")
```

Being rigid on ORMs

- On anything generating SQLs
- **They are bound to generate bad queries sometime**
- Does not mean they don’t have value

```
type User struct {
 ID      uint
 Orders []Order
}

type Order struct {
 ID      uint
 UserID uint
}

func main() {
…
  var users []User
  err = db.Preload("Orders").Find(&users).Error
}


SQL: SELECT * FROM "users"

SQL: SELECT * FROM "orders" WHERE "orders"."user_id" IN
($1,$2,$3,$4,$5)
```

Being rigid on ORMs

- On anything generating SQLs
- **They are bound to generate bad queries sometime**
- Does not mean they don’t have value

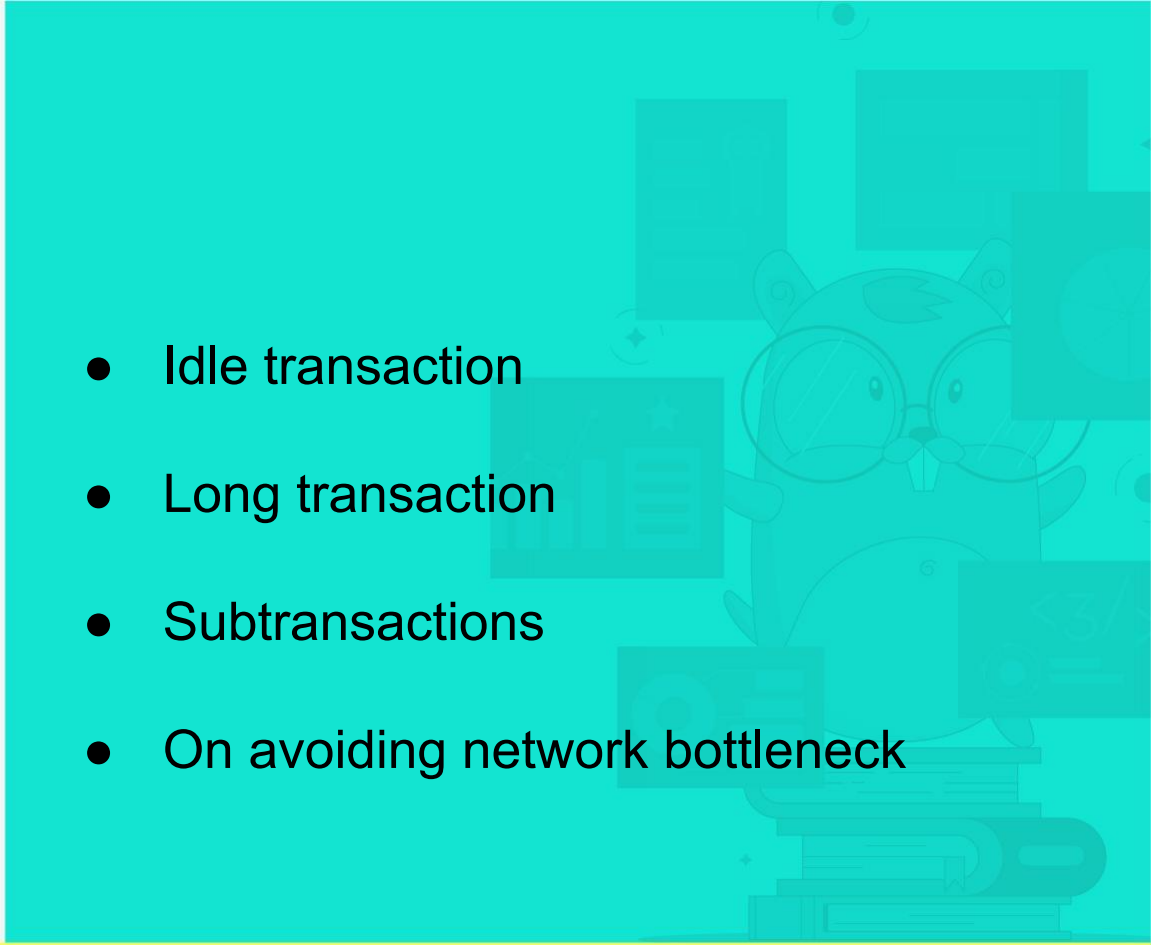
- https://openai.com/index/scaling-postgresql/
  2026. While solving math problems I guess

- *For example, we once identified an extremely costly query that joined 12 tables, where spikes in this query were responsible for past high-severity SEVs*
  …
  Many of these problematic queries are generated by Object-Relational Mapping frameworks (ORMs), so it’s important to carefully review the SQL they produce and ensure it behaves as expected.

- Solution:
  Just plan for possible “raw” SQL.
  Not everywhere, just on hot slow paths.

Transaction pitfalls

Hardest stuff second ?



- Idle transaction
- Long transaction
- Subtransactions
- On avoiding network bottleneck

Idle transactions

Keeping transactions open for any reason can be a limiter

- Calling external resources during tx
- Waiting for validation
- Costly compute not done upfront

```
tx, err := db.Begin()
defer tx.Rollback()

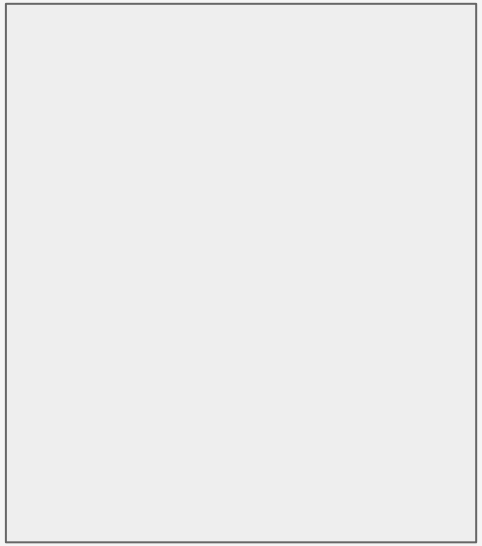
var price float64
tx.QueryRow("SELECT price FROM products WHERE id =
1").Scan(&price)


resp, err := http.Get(api.URL + "/eur-to-usd")
…
fmt.Sscan(string(body), &rate)


tx.Exec("INSERT INTO purchases (product_id, amount_usd) VALUES
(1, $1)", price*rate)


tx.Commit()
```

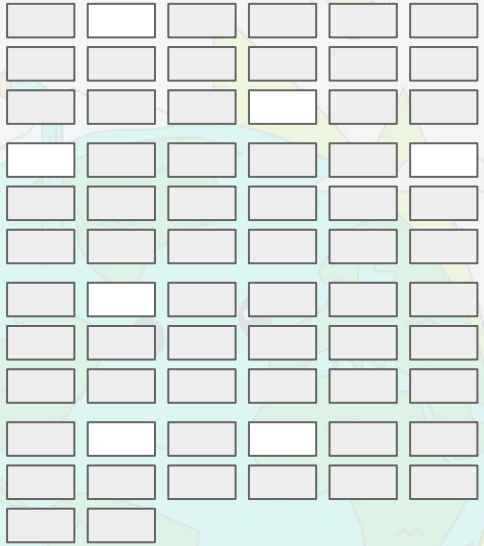
Idle transactions, why is it bad: how **tables** are structured



HEAP Table



Segments (1 GB)

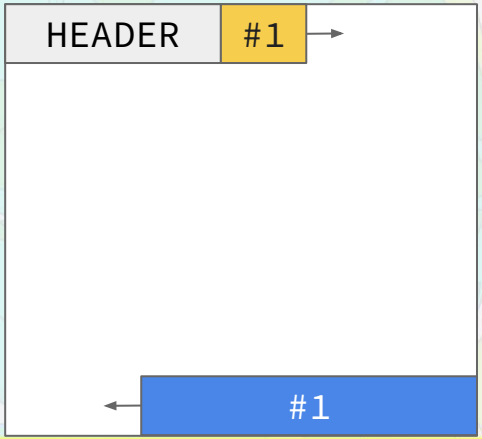


Pages (8 KB)

- Idle transactions, why is it bad: how **pages** are structured

```
INSERT INTO scores (id, score) VALUES (1, 200);
```

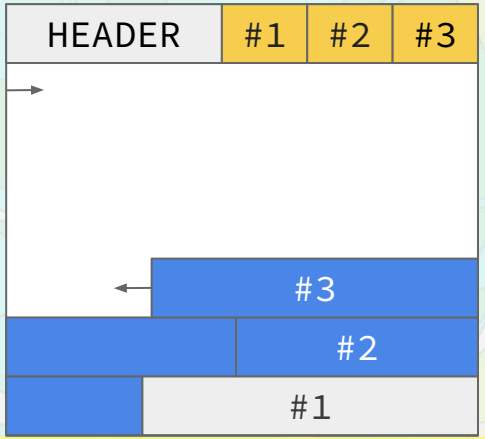
id	score	ctid	**xmin**	**xmax**
1	200	(0,1)	1	



Idle transactions, why is it bad: how **pages** are structured

```
BEGIN;
INSERT INTO scores (id, score) VALUES (2, 350), (3, 100);
COMMIT;
```

id	score	ctid	**xmin**	**xmax**
1	200	(0,1)	1	
2	350	(0,2)	2	
3	100	(0,3)	2	

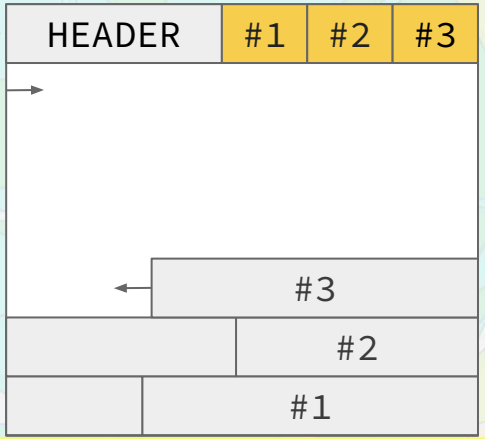


Idle transactions, why is it bad: how **pages** are structured

```
BEGIN ISOLATION LEVEL SERIALIZABLE;
SELECT *, ctid, xmin, xmax FROM scores;

-- NOT commit, keep it open and running. txid=3
```

id	score	ctid	**xmin**	**xmax**
1	200	(0,1)	1	
2	350	(0,2)	2	
3	100	(0,3)	2	

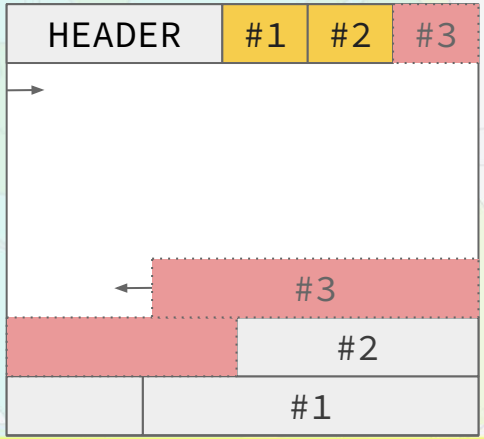


Idle transactions, why is it bad: how **pages** are structured

```
DELETE FROM scores WHERE id = 3;

-- the previous tx in select will still access it,
because its txid=3 is between xmin=2 and xmax=4
```

id	score	ctid	xmin	xmax
1	200	(0,1)	1	
2	350	(0,2)	2	
3	100	(0,3)	2	4

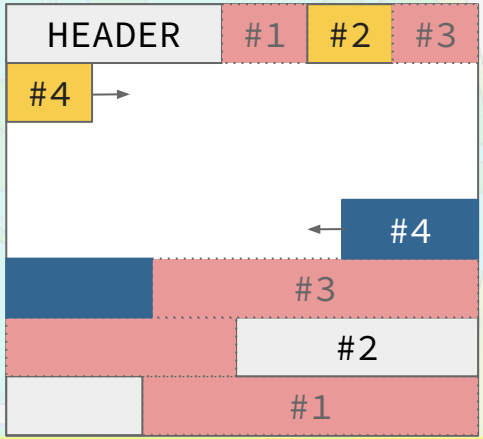


Idle transactions, why is it bad: how **pages** are structured

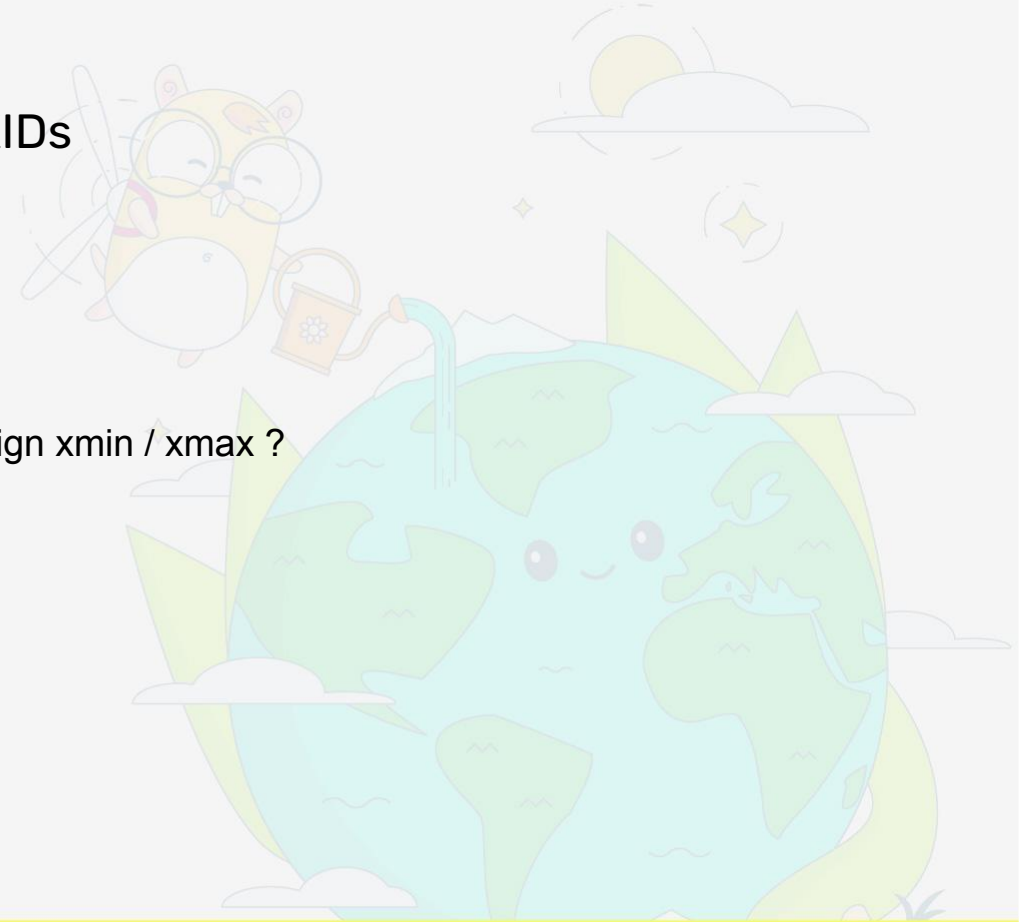
```
UPDATE scores SET score = 10 WHERE id = 1;

-- txid=3 select will still keep the old score,
because it's between xmin=1 and xmax=5
```

id	score	ctid	xmin	xmax
1	200	(0,1)	1	5
2	350	(0,2)	2	
3	100	(0,3)	2	4
1	10	(0,4)	5	



Idle transactions, why is it bad: TXIDs



Sounds easy.

But how many TXID available do we have to assign xmin / xmax ?

Idle transactions, why is it bad: TXIDs

Sounds easy.

But how many TXID available do we have to assign xmin / xmax ?

A full uint32!

=> Just 4 billions. (actually small)



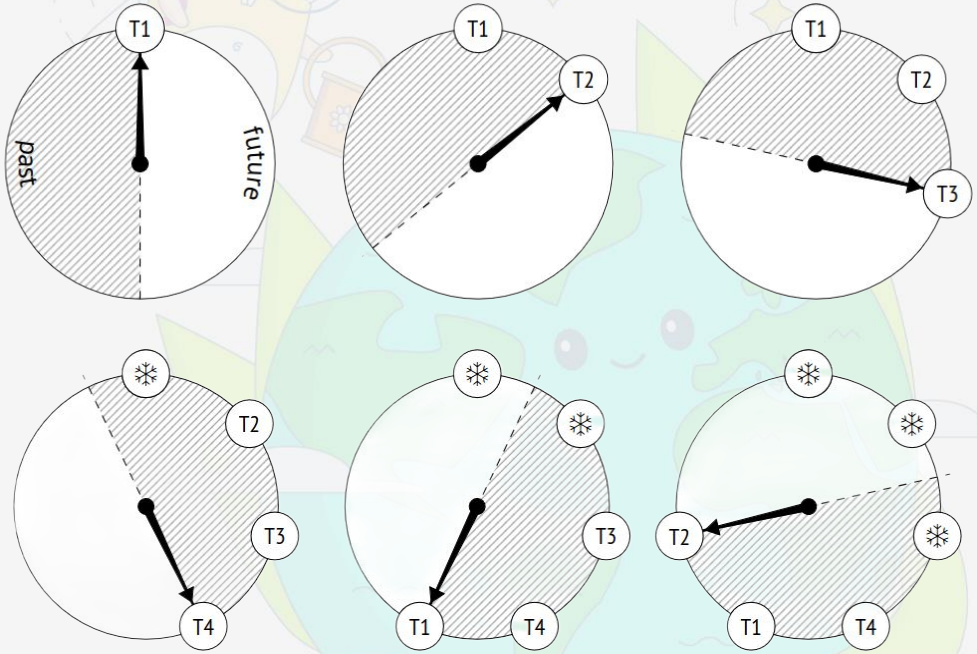
Idle transactions, why is it bad: TXIDs

But it actually loops!

To avoid lacking txid, it will **reuse** old ones.

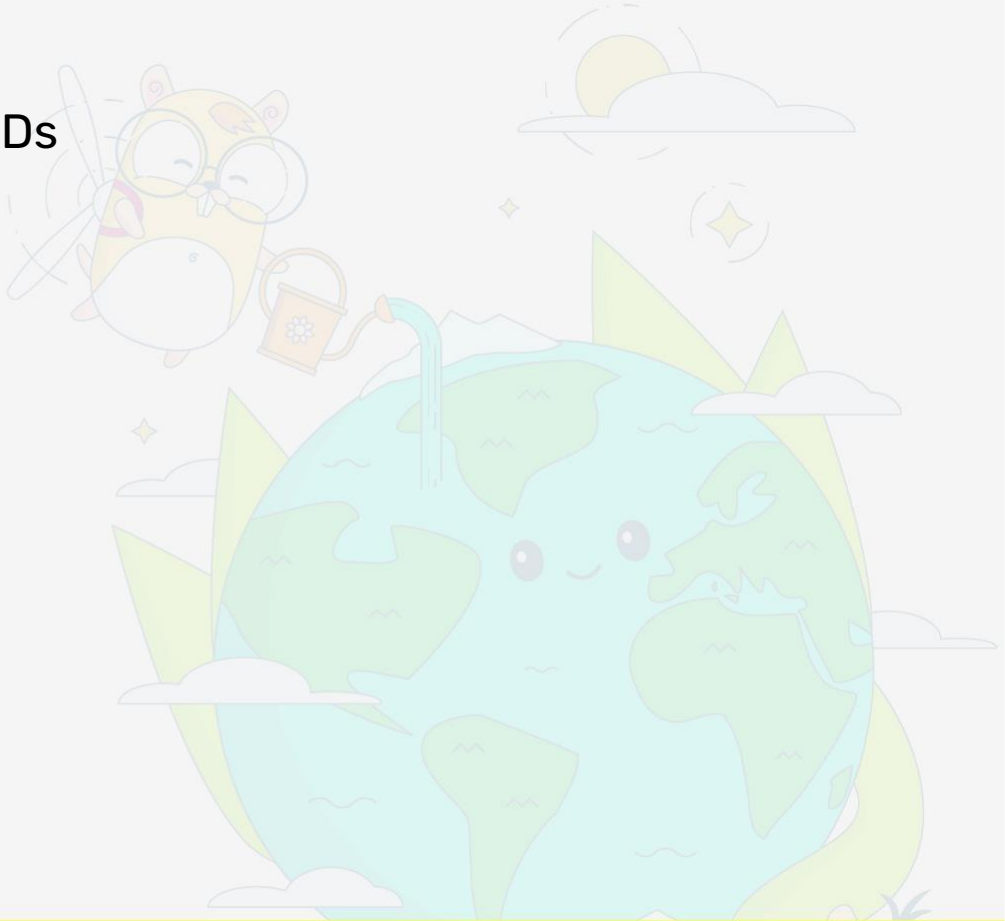
Past rows xmin would become future rows xmin.

To avoid that: it will **“freeze” rows** with a “frozen” bit meaning **we can ignore its xmin**



PostgreSQL 14 internals - Egor Rogov

Idle transactions, why is it bad: TXIDs



Cool beans.

How does it freeze rows ?

Idle transactions, why is it bad: TXIDs

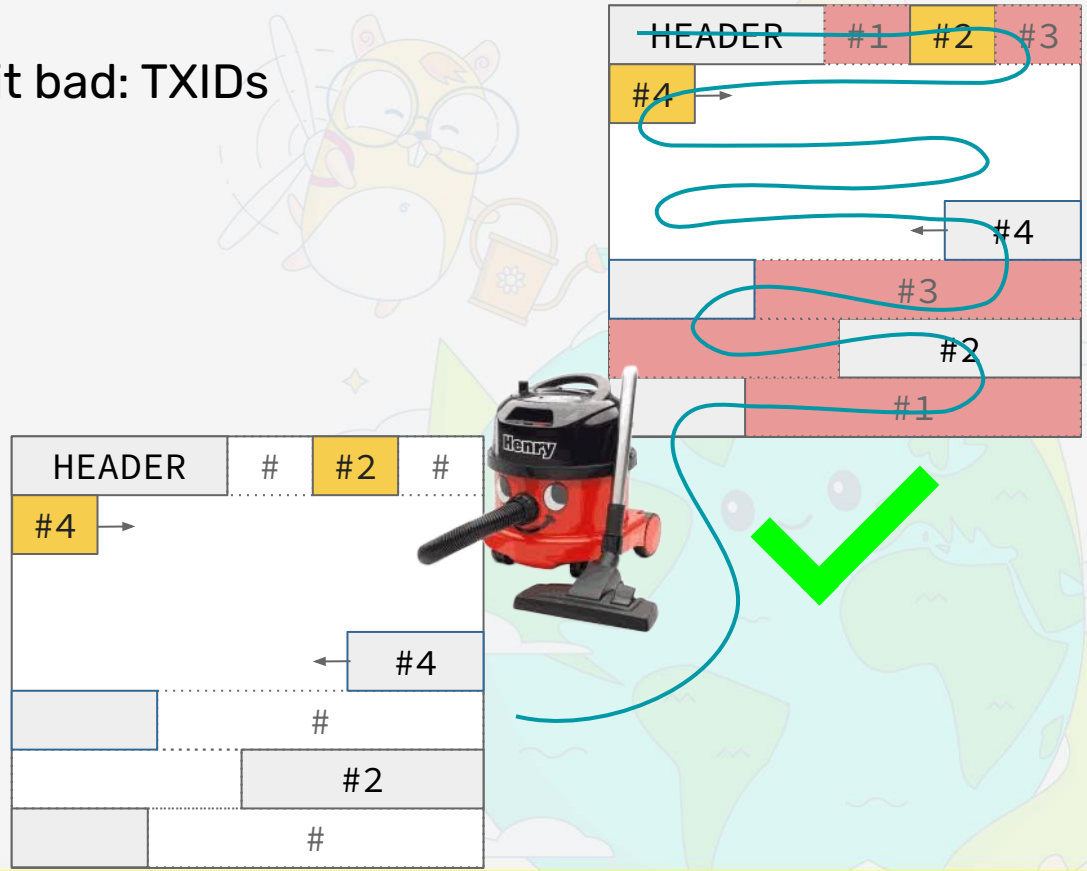
Cool beans.

How does it freeze rows ?

The infamous VACUUM

Dead rows will be removed.

Rows will be **frozen** based on the **oldest TXID** currently open



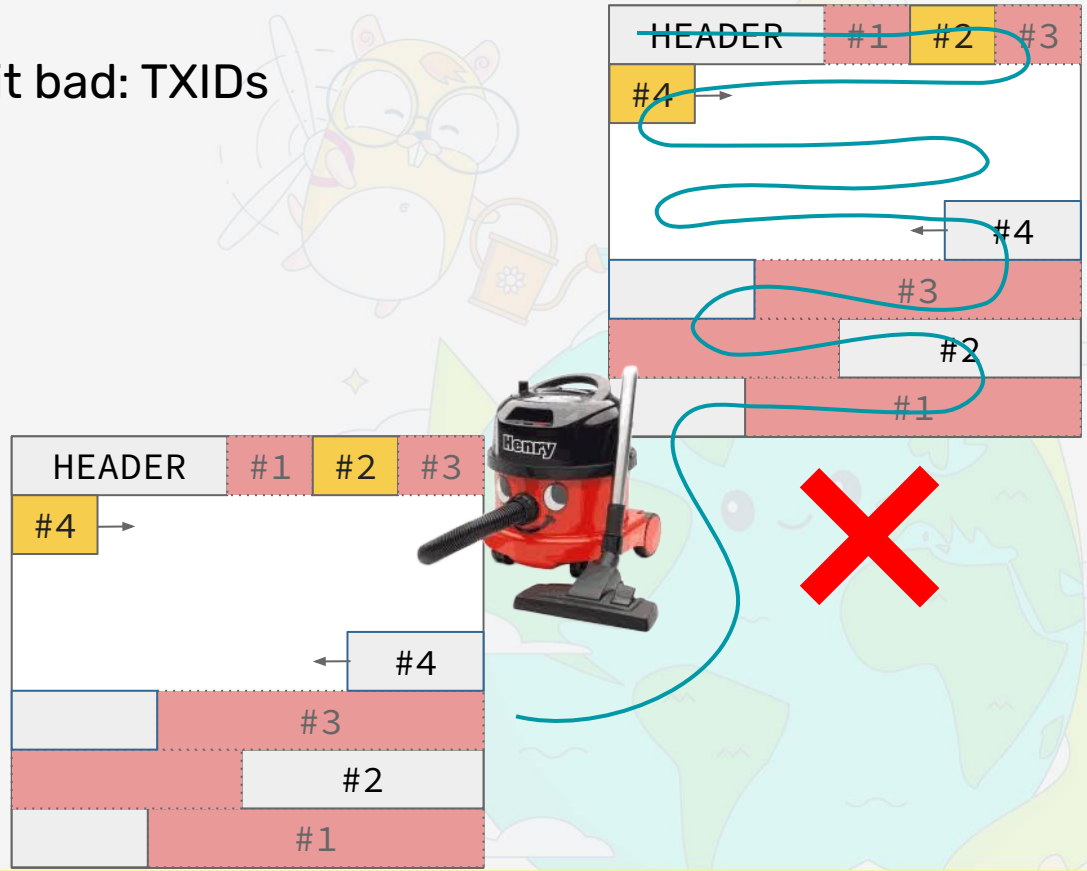
Idle transactions, why is it bad: TXIDs

Unless we kept some TX open!

Rows versions accumulate

Pages become bloated

And the vacuum might not be able to freeze more rows!



Idle transactions

So, keeping transactions open:

- **Bloating tables**
  (=> slowing down every accesses, wasting storage)
- **Might lead to txid exhaustion !**
  (=> pg readonly, big vacuum.)
- Pg workaround:
  idle_in_transaction_session_timeout

```
tx, err := db.Begin()
defer tx.Rollback()

var price float64
tx.QueryRow("SELECT price FROM products WHERE id =
1").Scan(&price)


resp, err := http.Get(api.URL + "/eur-to-usd")
…
fmt.Sscan(string(body), &rate)


tx.Exec("INSERT INTO purchases (product_id, amount_usd) VALUES
(1, $1)", price*rate)


tx.Commit()
```

Long transactions

Long-running (but not idle!) transactions are a bit worse, but could be necessary

- Same impacts
- No workarounds from postgres

```
tx, err := db.Begin()
defer tx.Rollback()

var price float64
tx.QueryRow("SELECT price FROM products WHERE id =
1").Scan(&price)


tx.Exec("INSERT INTO purchases (product_id, amount_usd)
VALUES (1, $1)", price*rate)


// Tons of slow DMLs or SELECTs
[...]

tx.Commit()
```

Sub transactions

Why not split a long transaction with subtransactions ?

=> even worse

=> **can happen implicitely**

```
// pgx
tx, _ := pool.Begin(ctx)    // BEGIN
sp, _ := tx.Begin(ctx)      // SAVEPOINT sp_1

// pgx interface just expects something that can do Begin()
// a conn, a tx, both implements Begin()
func BeginFunc(
     ctx context.Context,
     db interface {
         Begin(ctx context.Context) (Tx, error)
     },
…

// bun
tx, _ := db.BeginTx(ctx, nil)    // BEGIN
sp, _ := tx.BeginTx(ctx, nil)    // SAVEPOINT SP_<random hex>

// GORM
db.Transaction(func(tx *gorm.DB) error {     // BEGIN
      return tx.Transaction(func(tx *gorm.DB) error {
              // SAVEPOINT sp<random>
              return nil
      })
}) // In GORM, calling tx.Begin() on a tx doesn't nest.
// GORM's source returns ErrInvalidTransaction for that case.
```

Sub transactions

Why not split a long transaction with subtransactions ?

=> even worse

https://postgres.ai/blog/20210831-postgresql-subtransactions-considered-harmful

```
Why worse:
  -   Create even more TXID. (per sub-tx)
      So, faster exhaustion
```

Sub transactions

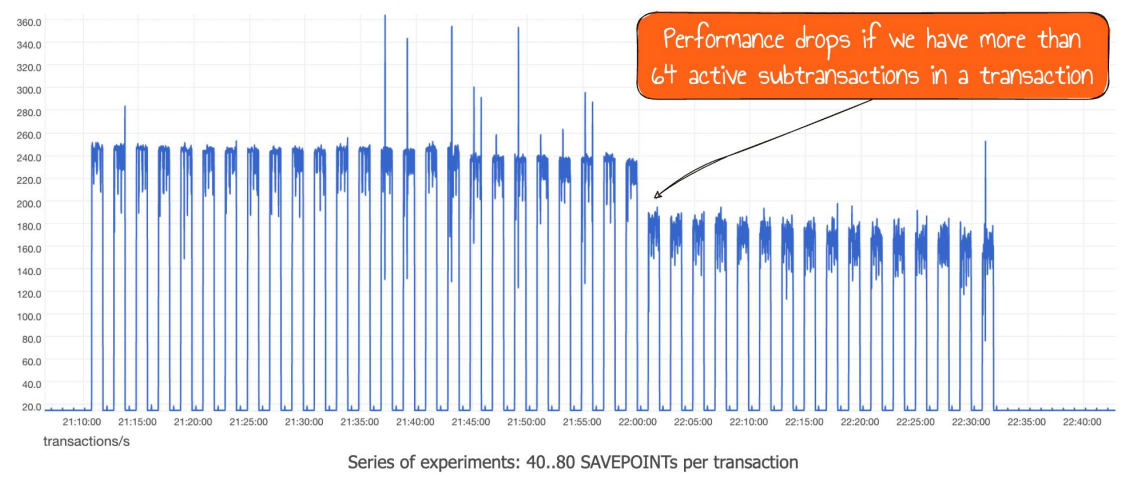
Why not split a long transaction with subtransactions ?

=> even worse

https://postgres.ai/blog/20210831-postgresql-subtransactions-considered-harmful

```
Why worse:
  -  Create even more TXID. (per sub-tx)
     So, faster exhaustion

  -  Performance cliff after having more
     than 64 sub-tx in a tx
```



Series of experiments: 40..80 SAVEPOINTs per transaction

Sub transactions

Why not split a long transaction with subtransactions ?

=> even worse

https://postgres.ai/blog/20210831-postgresql-subtransactions-considered-harmful

Why worse:

- Create even more TXID. (per sub-tx) So, faster exhaustion
- Performance cliff after having more than 64 sub-tx in a tx
- Contention on the multixactID store (row lock caches)

Sub transactions

Why not split a long transaction with subtransactions ?

=> even worse

https://postgres.ai/blog/20210831-postgresql-subtransactions-considered-harmful

Why worse:

- Create even more TXID. (per sub-tx) So, faster exhaustion
- Performance cliff after having more than 64 sub-tx in a tx
- Contention on the multixactID store (row lock caches)
- Can even kill the performance of hot standby queries

Sub transactions

Why not split a long transaction with subtransactions ?

=> even worse

https://postgres.ai/blog/20210831-postgresql-subtransactions-considered-harmful



Sub transactions

Why not split a long transaction with subtransactions ?

=> even worse

https://postgres.ai/blog/20210831-postgresql-subtransactions-considered-harmful

```
TLDR

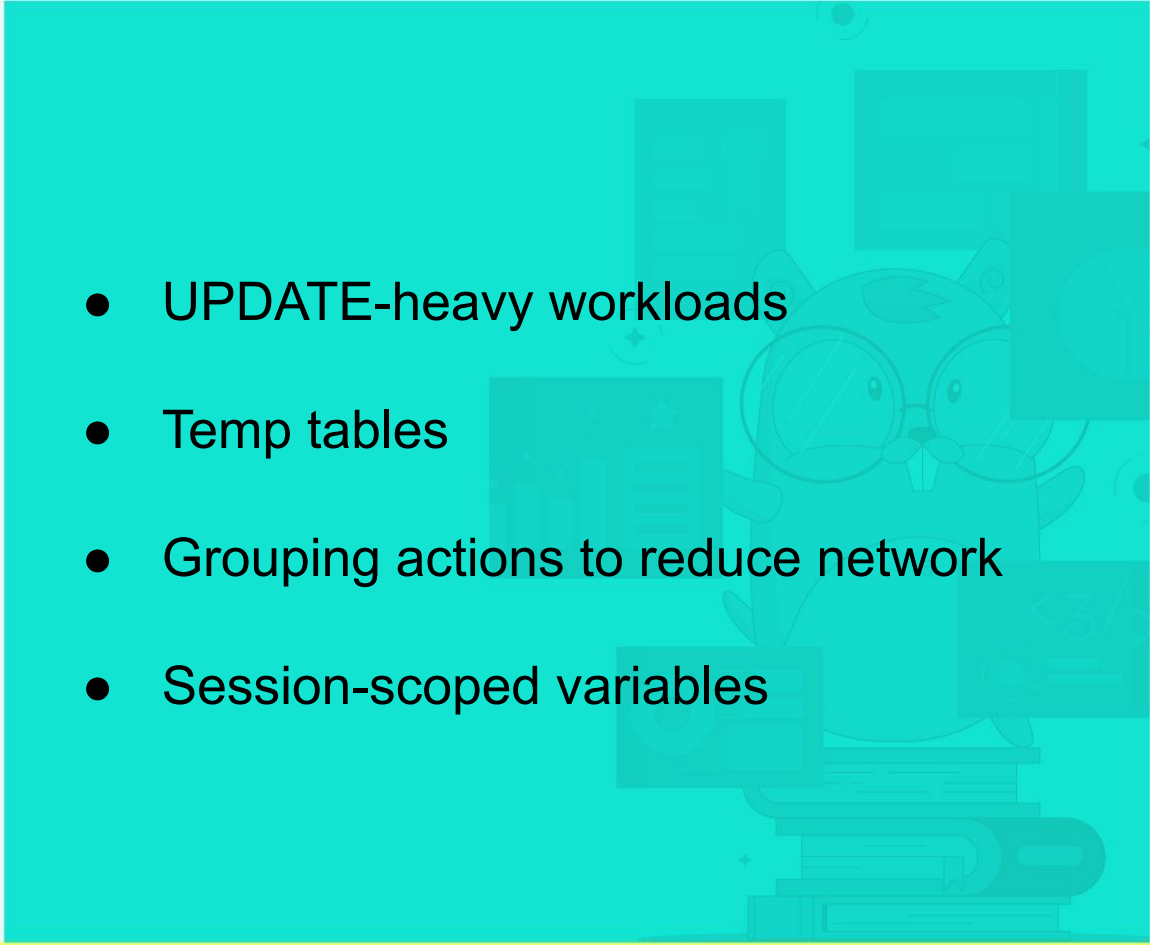
Sub transaction bad.

Make sure your ORM don't do them.

Broke gitlab. Broke AWS.
```

Usage patterns

- UPDATE-heavy workloads
- Temp tables
- Grouping actions to reduce network
- Session-scoped variables



Usage patterns

UPDATE-heavy workload

```
Todo:
Update being 2 op in a trenchcoat

Fillfactor/hot updates
```

Temp tables

Catalog xid exhaustion

Cte are just better

Grouping actions to reduce network

Cte

On conflicts clauses

Usage patterns

Session-scope variables

Pgbouncer: polluting next tx

Aws rds: pinning connections

Avoiding pitfalls with Postgres and Go

QA

yoann.lacancellera@percona.com

