aiven

What is MVCC and why should I care about it?

Patrick Stählin, PGConf.de 2025

Promise of PG (ACID)

- Changes are **a**tomic
- Data is **c**onsistent
- Changes are **i**solated
- Data is stored **d**urable

Multi

There can be multiple copies of a tuple (“row”) that can be valid at the same time.

ID (PK)	Name
1	M**a**ier
1	M**e**ier
1	M**ey**er

Version

All tuples have an id (`ctid`). Additionally each tuple has a version number from which it starts to be visible (`xmin`) and from which it will no longer be (`xmax`).

xmin	xmax	ID	Name
1	3	1	M**a**ier
3	7	1	M**e**ier
7	0	1	M**ey**er

Concurrency Control

The protocol allows concurrent access and has a couple of built in rules. To identify which tuples are visible, each transaction gets a transaction number (`txid_current`).

There are a couple of rules that are followed to determine the visibility of a tuple. Those rules are dependent on the isolation-level currently set.

`txid_current == 7`

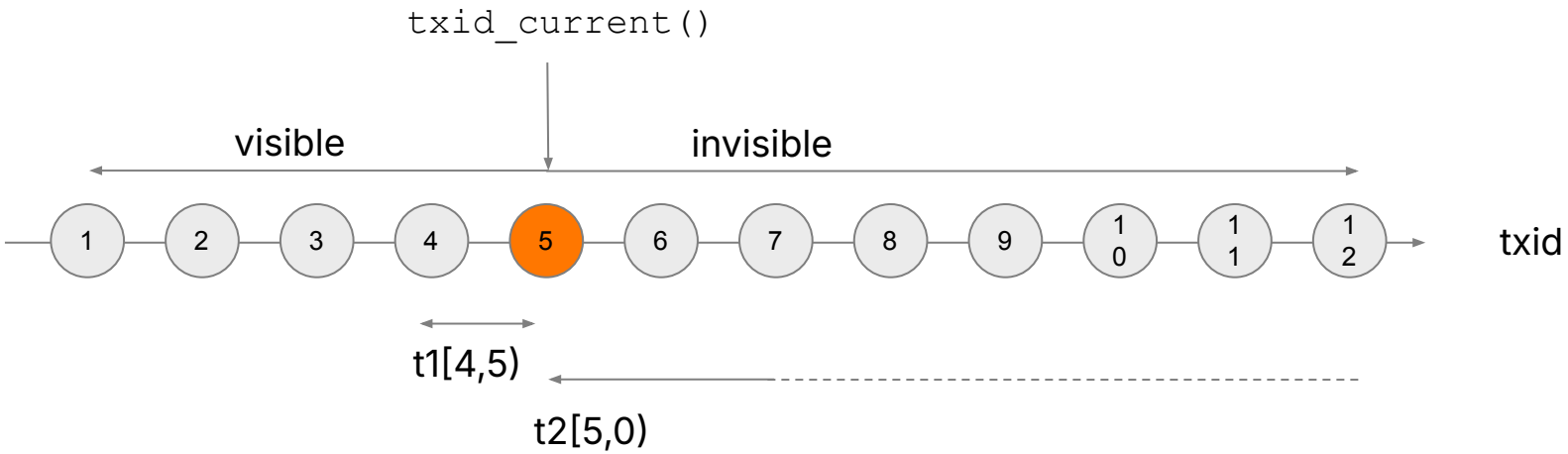
xmin	xmax	ID	Name
1	3	1	M**a**ier
3	7	1	M**e**ier
7	0	1	M**ey**er

Pros and cons of MVCC

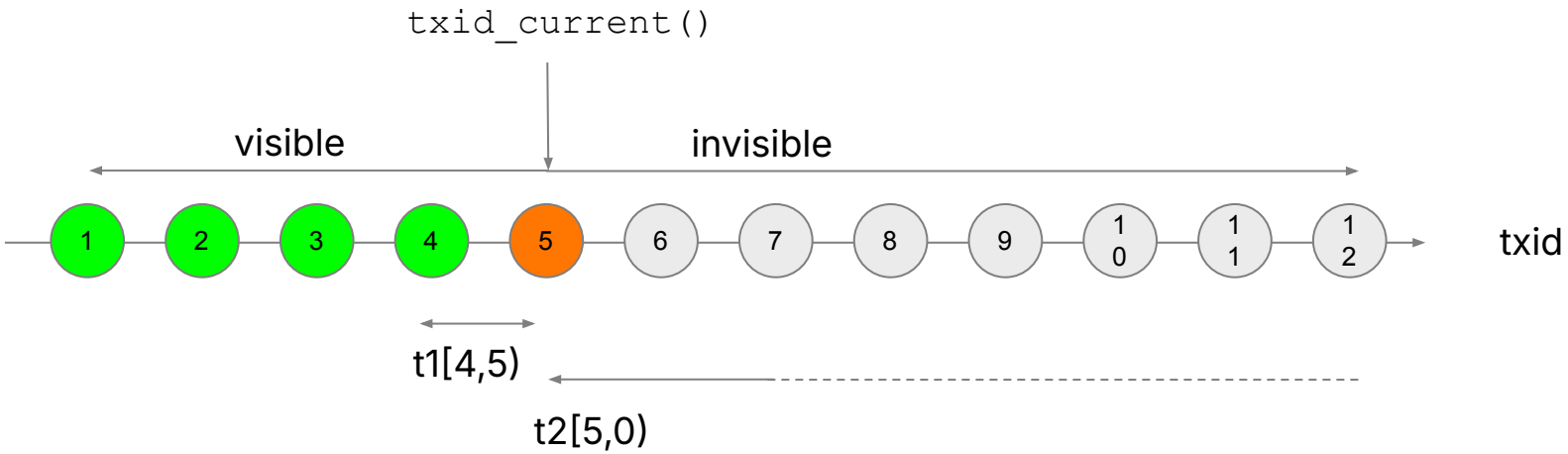
- Doesn’t need locking
- Changes are easy
- It’s an optimistic algorithm

- Data- and I/O-intense
- You need to clean up
- Read access gets complicated

Visibility of tuples



Visibility of tuples



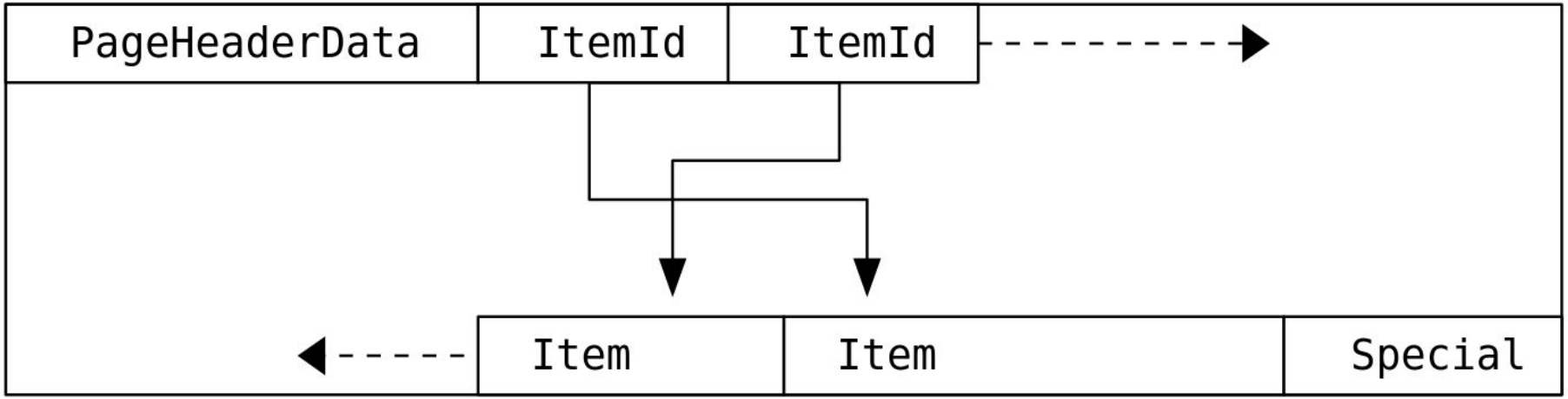
MVCC

- User data is immutable (write-only)
- `DELETE` only updates `xmax`
- UPDATE ist a `DELETE` followed by an `INSERT`
- Empty updates also create new tuples

Digression: How do we store things

- Tables are single files (up to 1 GB)
- Big tuples will not reside completely in those files but in extra ones (TOASTed)
- Indices also get one file each
- Files contain many pages (8 kB)

Heap Page Layout



https://www.postgresql.org/docs/current/storage-page-layout.html

Page Item Layout

Field	Type	Length	Description
t_xmin	TransactionId	4 bytes	insert XID stamp
t_xmax	TransactionId	4 bytes	delete XID stamp
t_cid	CommandId	4 bytes	insert and/or delete CID stamp (overlays with t_xvac)
t_xvac	TransactionId	4 bytes	XID for VACUUM operation moving a row version
t_ctid	ItemPointerData	6 bytes	current TID of this or newer row version
t_infomask2	uint16	2 bytes	number of attributes, plus various flag bits
t_infomask	uint16	2 bytes	various flag bits
t_hoff	uint8	1 byte	offset to user data

https://www.postgresql.org/docs/current/storage-page-layout.html

Page Item Layout

Most of those fields are queryable:

```
defaultdb=> select xmin, xmax, ctid, * from jobs;
xmin  | xmax | ctid  | job_id | done
------+------+-------+--------+------
28173 |    0 | (0,1) |      1 | f
28185 |    0 | (0,2) |      2 | f
(2 rows)

defaultdb=>
```

If you have access to `pageinspect`, you can read all the fields.

https://www.postgresql.org/docs/current/pageinspect.html

Write Ahead Log (WAL)

- Crash-Recovery
- Replication
- Point-in-Time-Recovery

https://www.postgresql.org/docs/current/wal-intro.html

Example

```
txid_current = 12

BEGIN; INSERT INTO t VALUES ('A'); COMMIT;
```

Tuple ID	xmin	xmax	cid	ctid	data

Example

```
txid_current = 12

BEGIN; INSERT INTO t VALUES ('A'); COMMIT;
```

Tuple ID	xmin	xmax	cid	ctid	data
1	**12**	**0**	**0**	**(0,1)**	**A**

Example

```
txid_current = 13

BEGIN; UPDATE t SET data = 'B' WHERE data = 'A'; COMMIT;
```

Tuple ID	xmin	xmax	cid	ctid	data
1	12	0	0	(0,1)	A

Example

```
txid_current = 13

BEGIN; UPDATE t SET data = 'B' WHERE data = 'A'; COMMIT;
```

Tuple ID	xmin	xmax	cid	ctid	data
1	12	**13**	0	**(0,2)**	A
2	**13**	**0**	**0**	**(0,2)**	**B**

Example

```
txid_current = 14

BEGIN; DELETE FROM t WHERE data = 'B'; COMMIT;
```

Tuple ID	xmin	xmax	cid	ctid	data
1	12	13	0	(0,2)	A
2	13	0	0	(0,2)	B

Example

```
txid_current = 14

BEGIN; DELETE FROM t WHERE data = 'B'; COMMIT;
```

Tuple ID	xmin	xmax	cid	ctid	data
1	12	13	0	(0,2)	A
2	13	**14**	0	(0,2)	B

Use-case 1

- Job-Processor runs periodically
- Updates all jobs from the past and marks them as done

Use-case 1

Schema:

```
CREATE TABLE jobs (
    job_id SERIAL PRIMARY KEY,
    done bool DEFAULT ‘f’,
    scheduled_for TIMESTAMP DEFAULT NOW()
);
```

Job:

```
at_now = SELECT now();
SELECT * FROM WHERE done = ‘f’ AND scheduled_for < ?:at_now;
[...]
UPDATE jobs SET done = ‘t’ where scheduled_for < ?:at_now;
```

Use-case 1

```
=> UPDATE jobs SET done = 't' where scheduled_for < ?:at_now;
10000000 rows affected
=>
```

Use-case 1, too many UPDATEs

```
=> UPDATE jobs SET done = 't' where scheduled_for < ?:at_now;
10000000 rows affected
=>
```

- (Almost) each tuple gets a new version
- Those changes are written to WAL files
- Backup grows as there is more table bloat
- Backup grows as you need to keep the WAL files for PITR (Point-in-time Recovery)
- Replication lag may go through the roof
- Size on disk >> size of data

Visibility of tuples II

- Running transactions have sequential `txid`s
- Transactions may be aborted
- ⇒ Simple xmin, xmax filter doesn’t work

Also, depending on the isolation-level we need additional rules

Visibility of tuples II

Commit-Log (xact log)

txid	1	2	3	4	5
state	COMMITTED	IN_PROGRESS	ABORTED	COMMITTED	SUB_COMMITED

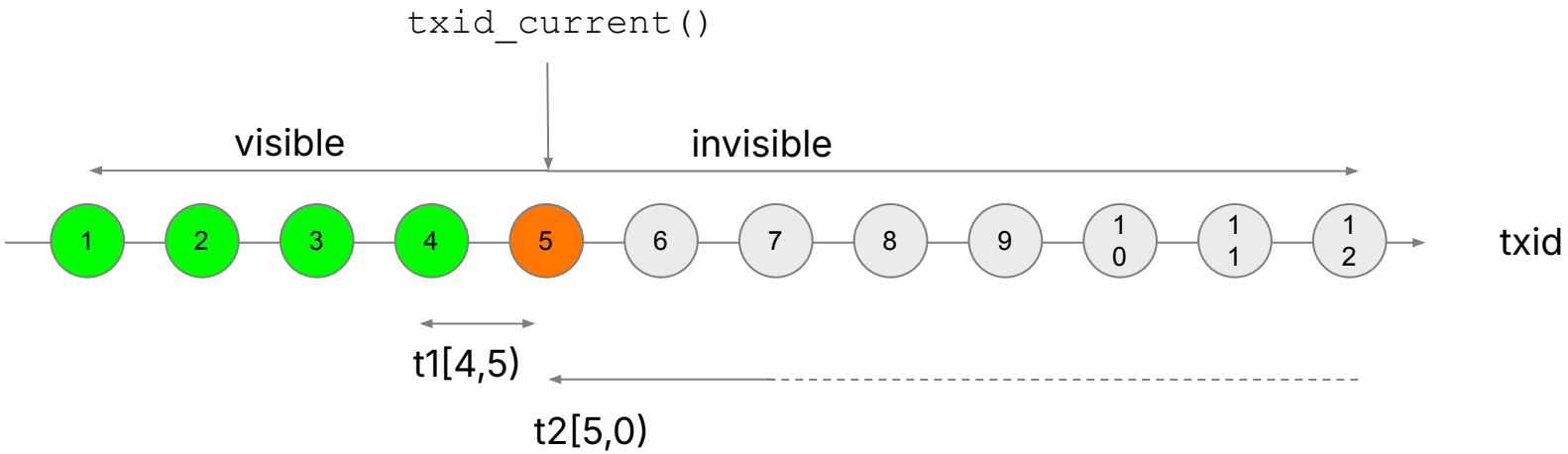
Visibility of tuples II

```
=> SELECT pg_current_snapshot()
2:6:2,3
```

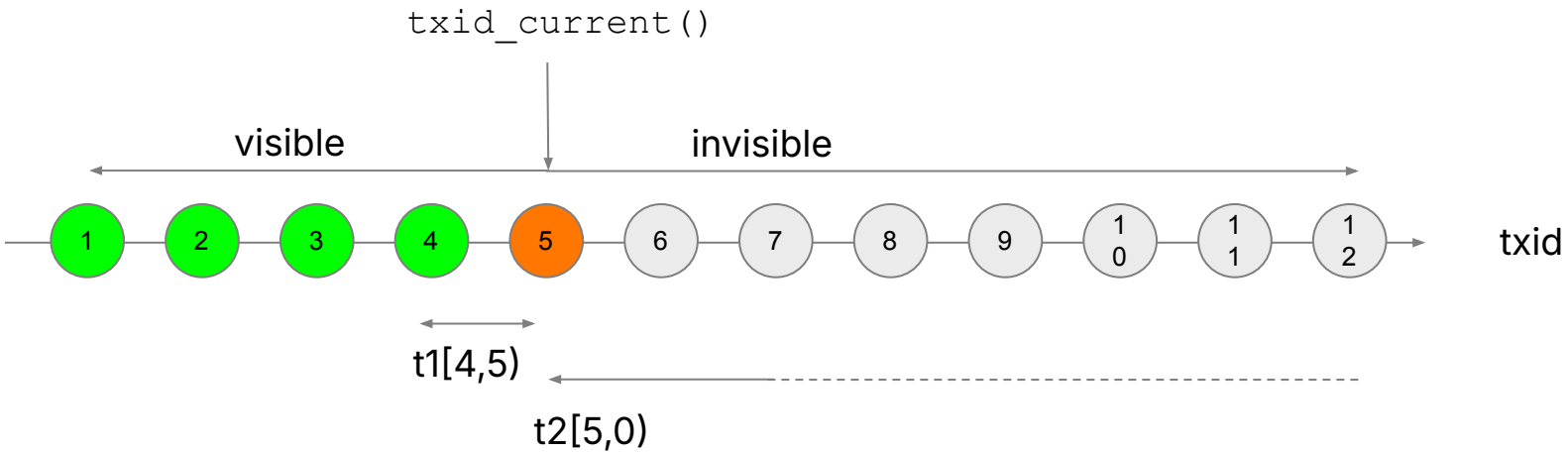
2 ⇒ xmin, first txid that is still active
6 ⇒ xmax, first txid that is not yet active
2,3 ⇒ pending transactions

The snapshot gets calculated when you execute the first statement in your transaction, depending on the isolation level.

Visibility of tuples II

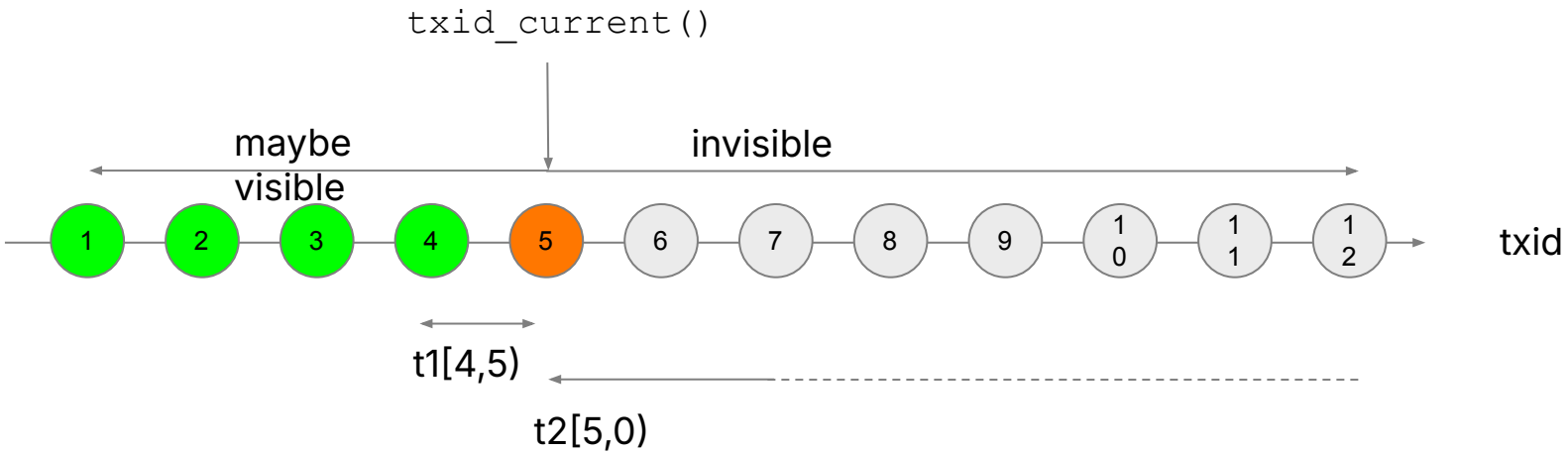


Visibility of tuples II



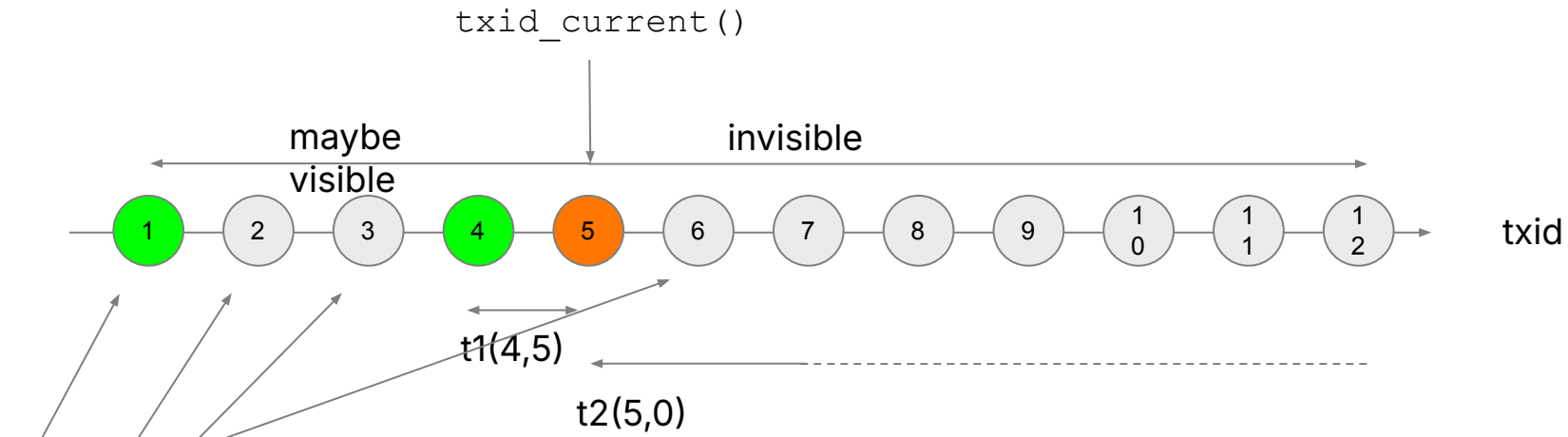

```
=> SELECT pg_current_snapshot()
1:6:2,3
```

Visibility of tuples II




```
=> SELECT pg_current_snapshot()
1:6:2,3
```

Visibility of tuples II



Use-case 2

- Job-Processor still runs periodically
- New design, we’re now updating jobs one-by-one

Use-case 2

Schema:

```
CREATE TABLE jobs ( [...] );
```

Job:

```
at_now = SELECT now();
SELECT id FROM WHERE done = ‘f’ AND scheduled_for < at_now
FOR UPDATE;

BEGIN;
[for each job]
UPDATE jobs SET done = ‘t’ WHERE job_id = ?:job_id;
COMMIT;
```

Use-case 2

- Updates single rows
- Jobs will still get written twice
- Long running transaction on jobs, maybe blocking other things
- Global xmin stays pinned due to the `…FOR UPDATE` clause

Use-case 2, long transactions

- Locking of tables
- Makes cleanup of bloat impossible
- May also lock writes/updates of jobs

Cleaning up (VACUUM)

- Dead-tuple removal (also on Indices)
- xact cleanup (txid ↔ state mapping)
- Freeze txids (`t_infomask |= XMIN_FROZEN`)
  Freezes tuples that may be removed due to txid-wraparound
- Updates FSM (Free-Space-Map), VM (Visibility-Map) und statistics
- Needs a `ShareUpdateExclusiveLock`
- Empty pages will only removed by VACUUM FULL

Dead-tuple removal

- Non-reachable tuples will get removed from pages (`xmax` < global `xmin`)
- Heap-Pages get defragmented
- References from indices to dead-tuples are being removed
- Free-Space-Map (FSM) und Visibility-Map (VM) get updated

Conclusion

- Every `UPDATE` will write a new tuple
- `VACUUM` is very important
- Keep your transactions (and connections) short lived
  - `transaction_timeout` (PG 17 and up)
  - `idle_in_transaction_session_timeout`

Thank You!

Patrick Stählin

Aiven PostgreSQL® Team

Senior Developer, Aiven

patrick.staehlin@aiven.io

https://www.linkedin.com/in/patrickstaehlin

Bsky @packi.ch

References

- https://www.interdb.jp/pg/pgsql05.html
- https://habr.com/en/companies/postgrespro/articles/477648/
- https://www.postgresql.org/docs/current/storage-page-layout.html