aiven

Was ist MVCC und was sind die Auswirkungen davon?

Patrick Stählin, Swiss PG Day 2023

Versprechen (ACID)

- Änderungen sind **a**tomar
- Daten sind konsistent (**c**onsistency)
- Änderungen sind **i**soliert
- Daten sind **d**auerhaft abgelegt

Multi

Von einer Tupel (“Row”) kann es mehrere Instanzen geben, die zur gleichen Zeit gültig sein können.

ID	Name
1	Maier
1	Meier
1	Meyer

Version

Alle Tupel sind mit einer Tupel-ID versehen (`ctid`). Jedes Tupel hat eine Versionsnummer, ab der es gültig (`xmin`) ist und eine Versionsnummer, ab der es nicht mehr gültig ist (`xmax`).

xmin	xmax	ID	Name
1	3	1	Maier
3	7	1	Meier
7	0	1	Meyer

Concurrency Control

Das Verfahren erlaubt mehrfachen Zugriff und regelt diesen. Dazu wird jeder Transaktion eine Nummer vergeben (`txid_current`).

Es gibt verschiedene Regeln die befolgt werden müssen, um die Sichtbarkeit von Tupel abzuklären. Diese sind abhängig vom jeweiligen Isolation-Level.

`txid_current == 7`

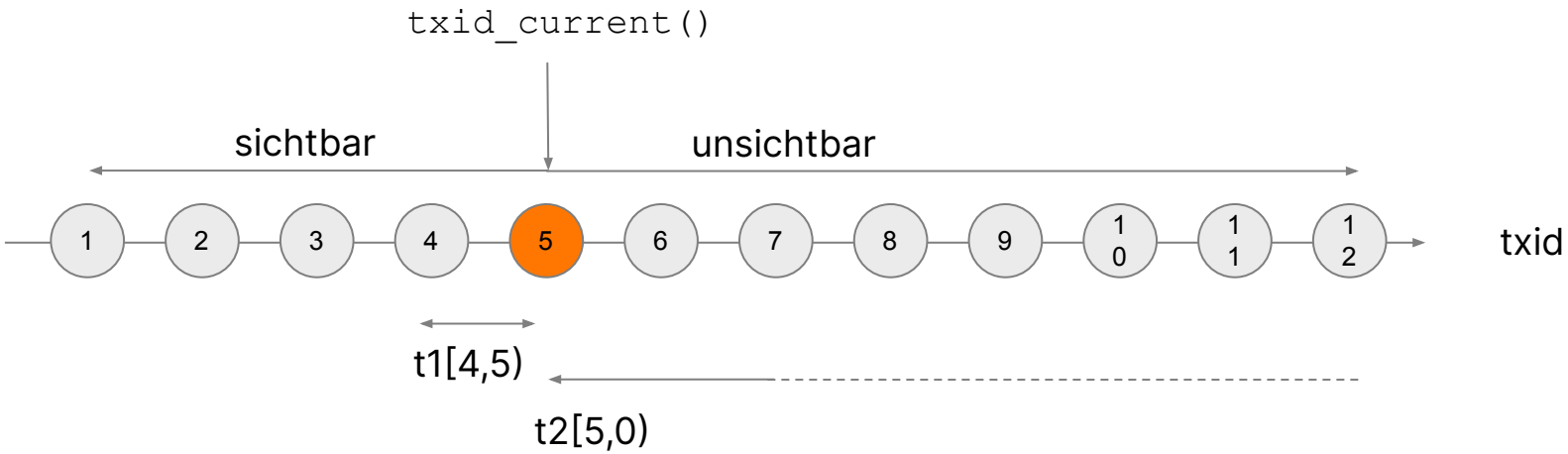
xmin	xmax	ID	Name
1	3	1	Müller
3	7	1	Meier
7	0	1	Meyer

Vor- und Nachteile

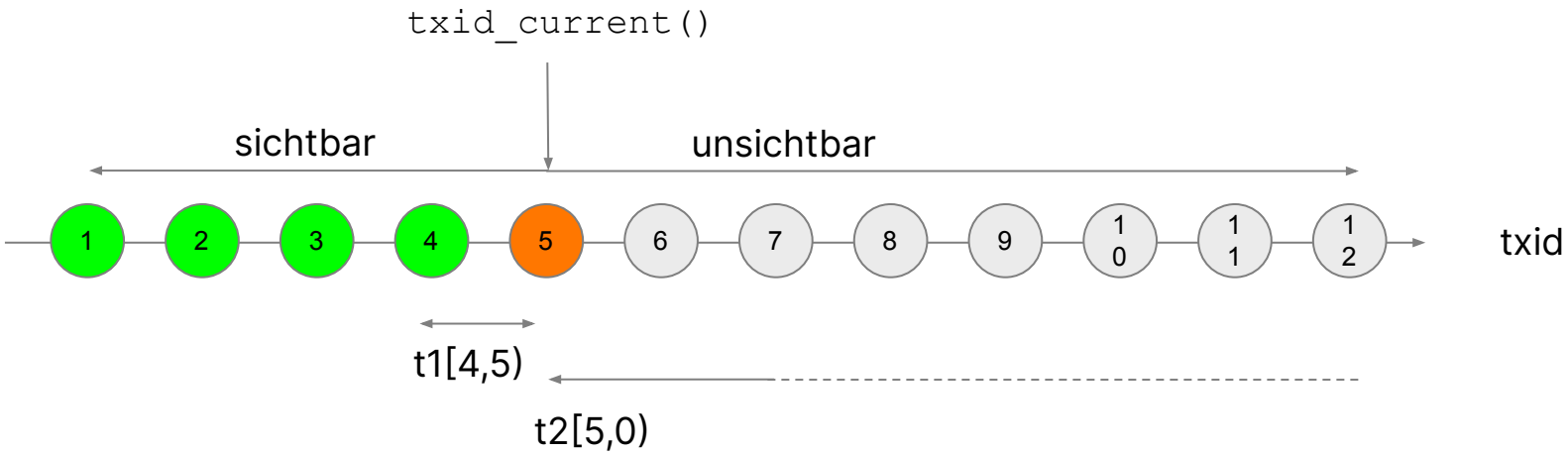
- Kommt ohne Locks aus
- Veränderungen sind einfach
- Optimistisches Verfahren

- Daten- und I/O-intensiv
- Es muss aufgeräumt werden
- Lesezugriff ist komplizierter

Sichtbarkeit von Tupel



Sichtbarkeit von Tupel



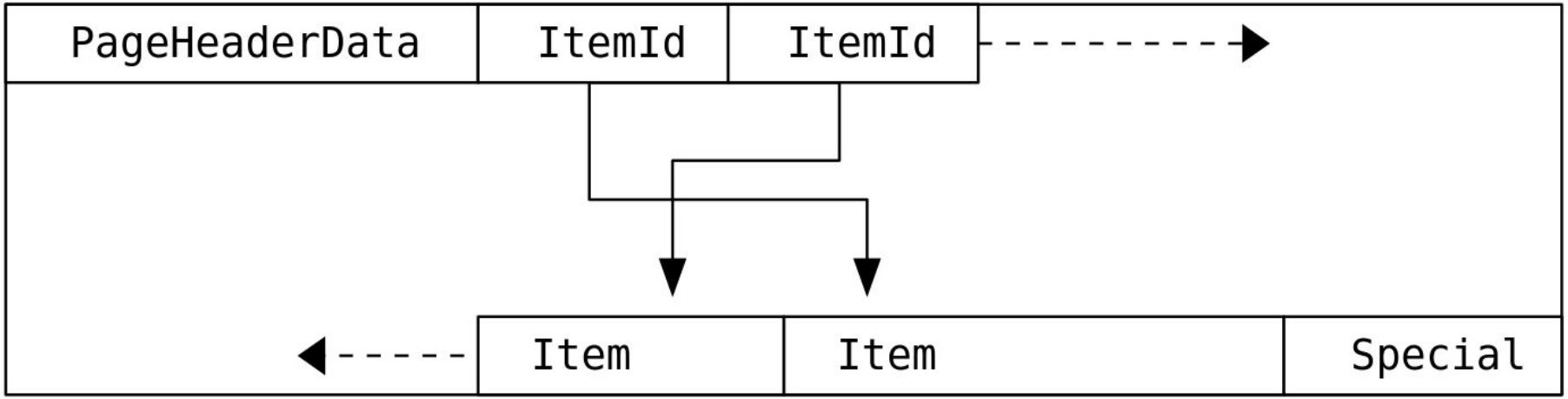
MVCC

- Keine User-Daten werden verändert (write-only)
- `DELETE` updated nur `xmax`
- Update ist ein `DELETE` gefolgt von `INSERT`
- Auch “leere” Updates erzeugen neue Tupel

Exkurs: Wie werden die Daten abgelegt

- Tabellen sind einzelne Dateien
- Grosse Tupel werden aber ausgelagert (TOASTed)
- Indizes sind ebenfalls einzelne Dateien
- Die Dateien bestehen dann aus einzelnen 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

Man kann gewisse Felder auch abfragen:

```
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=>
```

Mit der Extension `pageinspect` kommt man an alle Felder ran.

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

Beispiel

```
txid_current = 12

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

Tuple ID	xmin	xmax	cid	ctid	data

Beispiel

```
txid_current = 12

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

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

Beispiel

```
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

Beispiel

```
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**

Beispiel

```
txid_current = 14

BEGIN; DELETE FROM t data = ‘B’; COMMIT;
```

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

Beispiel

```
txid_current = 14

BEGIN; DELETE FROM t data = ‘B’; COMMIT;
```

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

Problemfall 1

- Job-Processor der periodisch läuft
- Updated alle Jobs in der Vergangenheit

Problemfall 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;
```

Problemfall 1

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

Problemfall 1, zu viele Updates

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

- Jedes Tupel wird neu geschrieben
- Die Changes gehen in WAL-Files
- Das Backup wird grösser, da es mehr Table-Bloat gibt
- Das Backup wird grösser, da die WAL-Files gebackupt werden müssen für PITR (Point-in-time Recovery)
- Replication-Lag schiesst in die Höhe
- Disk-Auslastung ist >> Daten

Sichtbarkeit von Tupel 2

- Laufende Transaktionen haben sequenzielle `txid`s
- Transaktionen können abgebrochen werden
- ⇒ Simpler xmin, xmax Filter funktioniert nicht

Je nach Isolation-Level gelten andere Regeln!

Sichtbarkeit von Tupel 2

Commit-Log (xact)

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

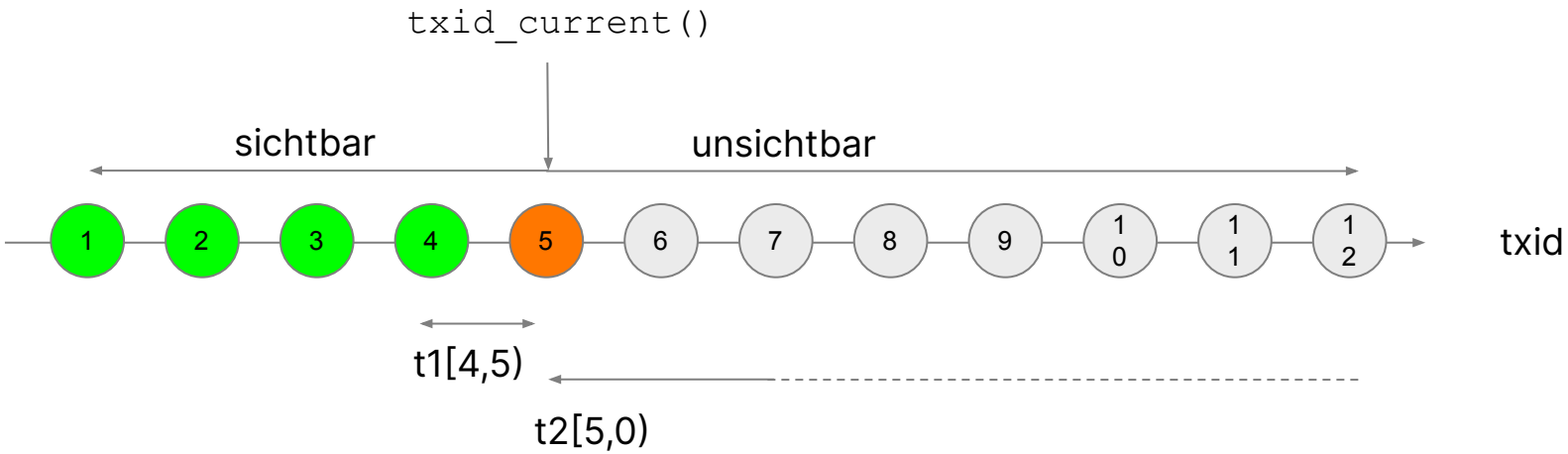
Sichtbarkeit von Tupel 2

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

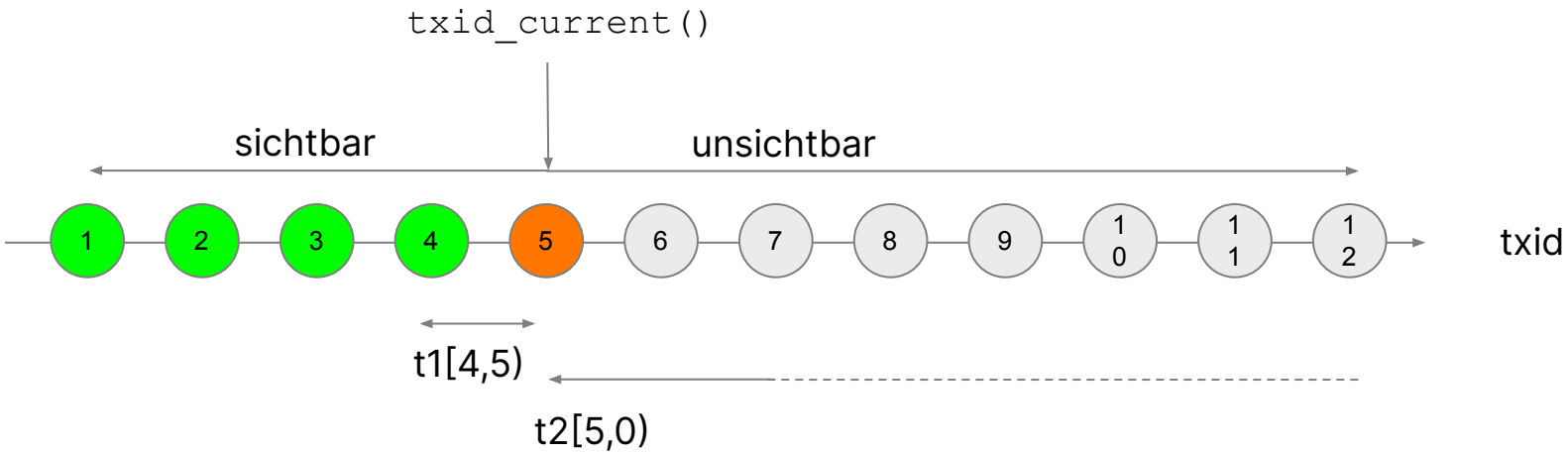
2 ⇒ xmin, erste txid die noch aktiv ist
6 ⇒ xmax, erste txid die noch nicht aktiv ist
2,3 ⇒ pendente Transaktionen

Der Snapshot wird je nach Isolation-Level beim Ausführen des ersten Statements berechnet.

Sichtbarkeit von Tupel 2

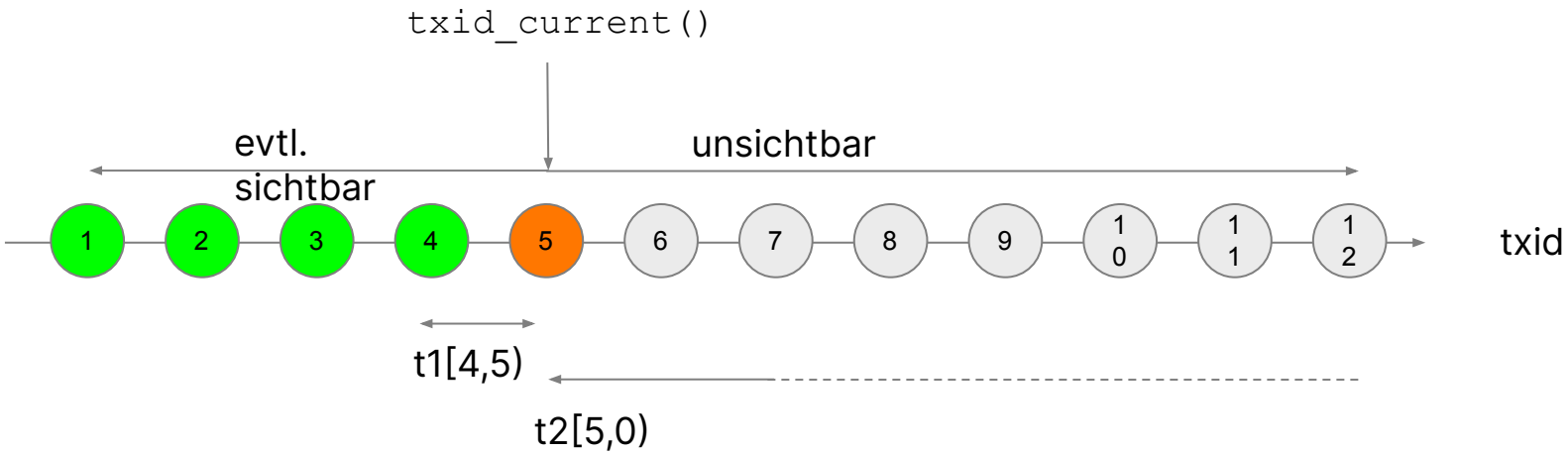


Sichtbarkeit von Tupel 2



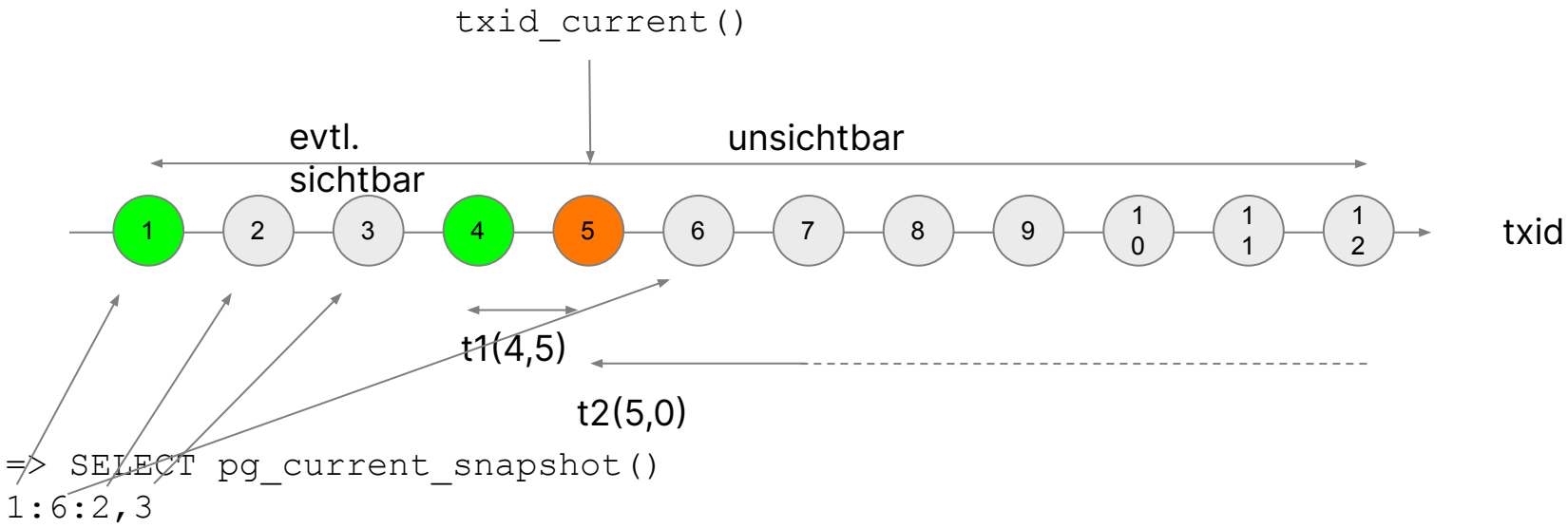

```
=> SELECT pg_current_snapshot()
1:6:2,3
```

Sichtbarkeit von Tupel 2




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

Sichtbarkeit von Tupel 2



Problemfall 2

- Job-Processor der periodisch läuft
- Neues Design, updated Jobs jetzt einzeln

Problemfall 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;
```

Problemfall 2

- Updated jetzt nur noch einzelne Rows
- Jobs werden immer noch immer zwei mal geschrieben
- Lange Transaktion welche alle Jobs lockt
- Globales xmin bleibt stehen

Problemfall 2, lange Transaktionen

- Locken Tabelle und verhindern Vacuum
- Verhindern das Äufräumen von Table-Bloat
- Verhindert ggf. auch das verändern von Jobs

Aufräumen (VACUUM)

- Dead-tuple removal (auch bei Indices)
- xact cleanup (txid ↔ state)
- Freeze txids (`t_infomask |= XMIN_FROZEN`)
  Verhindert das verschwinden von Tupel die drohen verloren zu gehen wegen, dem txid-Wraparound
- Update FSM (Free-Space-Map), VM (Visibility-Map) und Statistiken
- Braucht `ShareUpdateExclusiveLock`
- Leere Pages werden nur von VACUUM FULL gelöscht!

Dead-tuple removal

- Nicht mehr erreichbare Tupel werden aus den Pages entfernt (xmax < globaler xmin)
- Heap-Pages werden defragmentiert
- Verweise aus Indices auf tote Tupel werden entfernt
- Free-Space-Map (FSM) und Visibility-Map (VM) werden updated

Zusammenfassung

- Jedes Update schreibt ein neues Tupel
- VACUUM ist wichtig
- Haltet Transaktionen kurz

Vielen Dank!

Patrick Stählin

Aiven PostgreSQL® Team

Senior Developer, Aiven

patrick.staehlin@aiven.io

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

@thepacki

Referenzen

- 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