Zeilensperren in PostgreSQL: eine anatomische Betrachtung

Laurenz Albe

PROUD CONTRIBUTOR TO
PostgreSQL
the world's most advanced open source database

CYBERTEC
POSTGRESQL SERVICES & SUPPORT

Laurenz Albe

Talking Head

Email

laurenz.albe@cybertec-postgresql.com

Phone

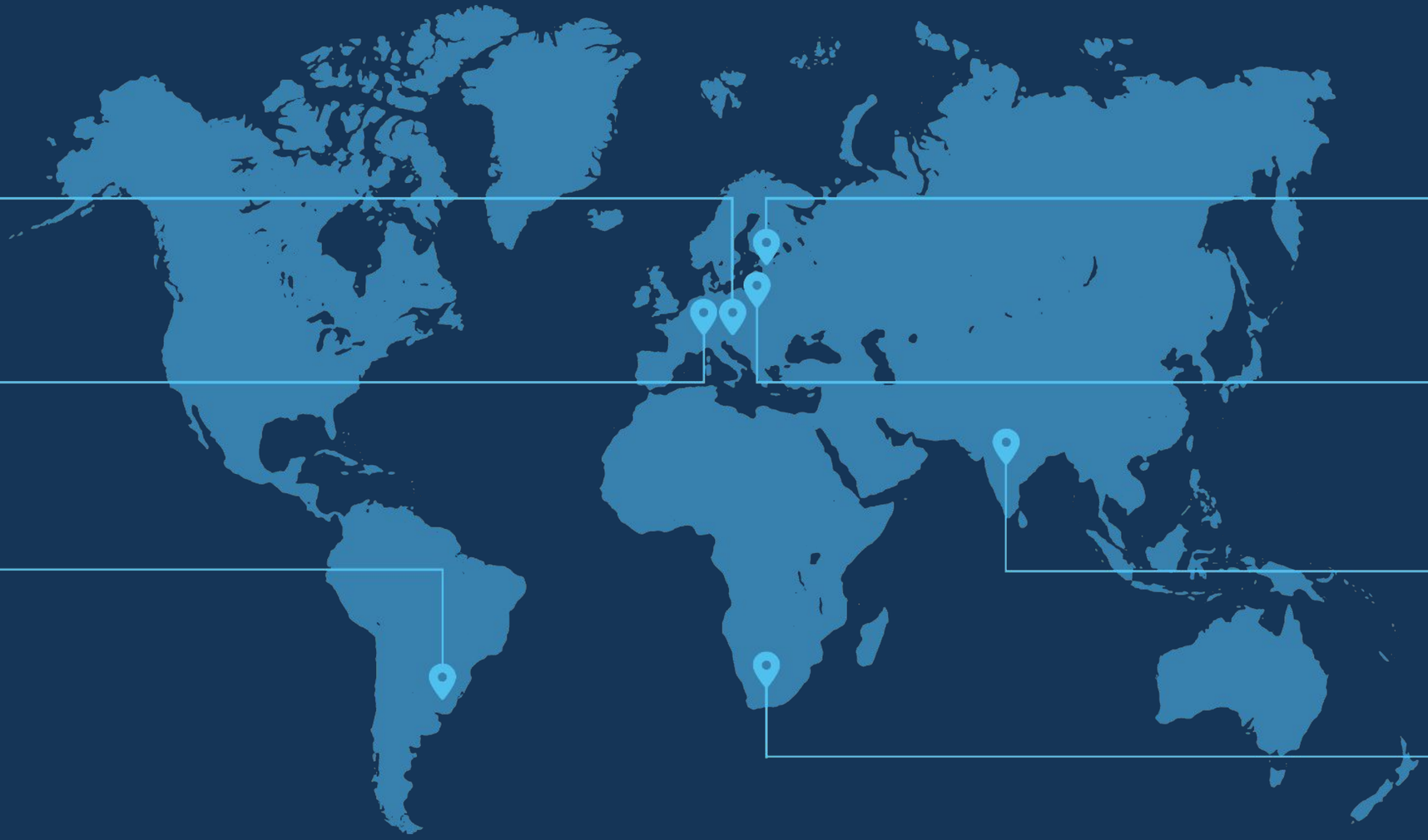
+43 670 805 6265

www.cybertec-postgresql.com

@cybertec-postgresql

www.youtube.com/@cybertecpostgresql

PROUD CONTRIBUTOR TO PostgreSQL
the world's most advanced open source database



AUSTRIA (HQ)

CYBERTEC POSTGRESQL
INTERNATIONAL (HQ)

SWITZERLAND

CYBERTEC POSTGRESQL
SWITZERLAND

URUGUAY

CYBERTEC POSTGRESQL
SOUTH AMERICA

ESTONIA

CYBERTEC POSTGRESQL
NORDIC

POLAND

CYBERTEC POSTGRESQL
POLAND

INDIA

CYBERTEC POSTGRESQL
INDIA PRIVATE LIMITED

SOUTH AFRICA

CYBERTEC POSTGRESQL
SOUTH AFRICA

Database Products & Tools

CYBERTEC MIGRATOR

CYPEX

CYBERTEC PGEE

scalefield

PG WATCH

PG SQUEEZE

scalefield secure

CYBERTEC LakeHouse

YAIM

WAL BOUNCER

DATA MASKING

pg_ TIMETABLE

PATRONI ENVIRONMENT SETUP

POSTGRESQL CONFIGURATOR

PG SHOW PLANS

PL/pgSQL_sec CYBERTEC

ISO 14001 CERTIFIED

ISO 27001 CERTIFIED

Grundlagen

Warum dieser Vortrag?

Jeder weiß, dass es Zeilensperren (row locks) gibt, aber nur wenige wissen

- wie Zeilensperren mit Fremdschlüsseln zusammenspielen
- wie Zeilensperren implentiert sind
- wie genau PostgreSQL Zeilen sperrt
- wie man Zeilensperren untersuchen kann
- was zum Teufel ein “MultiXact” wirklich ist
- warum MultiXacts problematisch sein können

Die Inspiration für den Titel habe ich von Gülçin Yildirim Jelíneks Vortrag über Tabellensperren übernommen!

Wozu brauchen wir Zeilensperren?

- PostgreSQL MVCC macht Zeilensperren oft unnötig
  - Leser und Schreiber sperren einander nicht; der Leser sieht die Vorversion
- Aber nur jeweils eine Transaktion kann eine Zeile verändern

  ➡ `UPDATE` und `DELETE` setzen exklusive Zeilensperren
- Wie alle Sperren werden Zeilensperren bis zum Ende der Transaktion gehalten

mit Zeilensperren arbeiten

Wie setzt man eine Zeilensperre?

- Meist setzt man Zeilensperren nicht explizit
- Zeilensperren werden automatisch von `UPDATE`, `DELETE` und (manchmal) `INSERT`-Statements gesetzt
- Explizite Zeilensperren setzt man mit

  `SELECT … FOR <Grad der Sperre>`

  Explizit Zeilen sperren („pessimistisches Sperren”) ist auf der Standard-Isolationsstufe `READ COMMITTED` erforderlich, um Datenkonsistenz sicherzustellen: wenn man beim Lesen die Zeile sperrt, kann sie niemand anderer verändern. So vermeidet man „Lost Updates”.

Grade von Zeilensperren

PostgreSQL kennt vier Grade von Zeilensperren:

	KEY SHARE	SHARE	NO KEY UPDATE	UPDATE
KEY SHARE				X
SHARE			X	X
NO KEY UPDATE		X	X	X
UPDATE	X	X	X	X

Zweck der verschiedenen Grade von Zeilensperren

- `KEY SHARE`: niemand anderer darf die Zeile löschen oder eine Spalte verändern, die zu einem Primärschlüssel oder einem eindeutigen Schlüssel gehört ⇒ für die Verwaltung von Fremdschlüsseln verwendet
- `SHARE`: niemand anderer darf die Zeile verändern ⇒ nicht verwendet; ein Überrest von früher
- `NO KEY UPDATE`: ich möchte die Zeile verändern, ohne Spalten von Primärschlüsseln oder eindeutigen Schlüsseln zu ändern ⇒ von UPDATE-Statements verwendet
- `UPDATE`: ich möchte die Zeile löschen oder eine Spalte eines Primärschlüssels oder eindeutigen Schlüssels ändern ⇒ von `DELETE`- und schlüsselverändernden `UPDATE`-Statements verwendet

Zeilensperren und Fremdschlüssel

Ein Problem mit referentieller Integrität

```
CREATE TABLE p (id bigint PRIMARY KEY, val text);
INSERT INTO p VALUES (1);
CREATE TABLE c (id bigint REFERENCES p);
```

Datenbanksitzung 1:

```
BEGIN;
DELETE FROM p WHERE id = 1;
```

Datenbanksitzung 2:

```
INSERT INTO c VALUES (1);
```

Sitzung 2 muss blockieren, bis Sitzung 1 die Transaktion mit COMMIT oder ROLLBACK beendet — aber die Statements ändern verschiedene Tabellen!

Lösung für das Problem

- Ein `INSERT` in die referenzierende Tabelle sperrt die referenzierte Zeile mit einer `KEY SHARE`-Sperre
- Die `KEY SHARE`-Sperre verträgt sich nicht mit der `UPDATE`-Sperre des `DELETE`
  ⇒ die Integrität ist sichergestellt
- Wohlgemerkt ist das nicht die einzige Art, das Problem zu lösen: Oracle sperrt beim `DELETE` die referenzierenden Zeilen.

Beim pessimistischen Sperren den richtigen Grad wählen

- Die meisten Leute sperren so, wenn sie ein UPDATE planen:
  `SELECT ... FROM p WHERE ... FOR UPDATE;`
- Aber das sperrt ein `INSERT`, das mit einem Fremdschlüssel auf diese Zeile verweist
- **Der korrekte Grad der Sperre wäre `FOR NO KEY UPDATE`**
- `FOR UPDATE` ist nur angebracht, wenn man plant, die Zeile zu löschen oder eine Schlüsselspalte zu verändern
- Historische Anmerkung: bevor `NO KEY UPDATE` und KEY SHARE erfunden wurden, hat PostgreSQL `UPDATE`- und `SHARE`-Sperren verwendet — daher der Name!

Implentierung von Zeilensperren

Wie speichert PostgreSQL normalerweise Sperren?

- In einer Tabelle im Shared Memory („Lock Table”)
- Der Speicher wird beim Server-Start reserviert
- Die Anzahl der Einträge ist in v18:

```
(max_connections + autovacuum_worker_slots +
max_worker_processes + max_wal_senders + 2 +
max_prepared_transactions) * max_locks_per_transaction
```

- Damit ist standardmäßig Platz für 8704 Sperren
- Die Lock Table kann mit der View `pg_locks` eingesehen werden

Das Problem mit Zeilensperren in der Lock Table

- Normalerweise ist die Zahl der Tabellen überschaubar, nicht aber die Anzahl der Zeilen
- Wenn Zeilensperren in der Lock Table gespeichert würden, könnte sie leicht überlaufen
- Um damit umzugehen, müsste man zu Seiten- oder Tabellensperren eskalieren
- Das ist kompliziert, schlecht für die Nebenläufigkeit und kann zu überraschenden „Deadlocks” führen
- Daher **speichert PostgreSQL Zeilensperren nicht dauerhaft in der Lock Table**

Wie implementiert PostgreSQL Zeilensperren?

- Zeilensperren werden auf der Tabellenzeile selbst gespeichert
  - Vorteil: beliebig viele Sperren ohne Überlauf der Lock Table
  - Nachteil: Zeilensperren ändern die Tabelle und werden auf die Disk geschrieben

```
 4      4      4     6          2            2       1        0-n
xmin | xmax | cid | ctid | infomask2 | infomask | hoff | null bitmap
```

- `xmax` speichert, wer die Sperre hält
  - `xmax` bestimmt normalerweise die Sichtbarkeit der Zeile
    (aber man kann ohnehin nur sichtbare Zeilen sperren)
  - Problem, wenn mehrere eine Zeile sperren (mehr darüber später)
- Die Bits in infomask speichern den Sperrengrad und andere Details

Wie Zeilensperren gesetzt werden

- Wenn es keinen Konflikt mit einer existierenden Sperre gibt, einfach xmax und infomask setzen und fertig
- Andernfalls eine „tuple”-Sperre in der Lock Table setzen (blockiert andere Transaktionen und bildet eine Warteschlange)
- Auf alle Transaktionen warten, die die Zeile gesperrt halten
- Aktuelle Version der freigewordenen Zeile holen, von vorne beginnen
- xmax und infomask ändern
- „tuple”-Sperre in der Lock Table freigeben

Wie Zeilensperren in `pg_locks` aussehen

- Zeilensperren, die niemanden blockieren, sind nicht in `pg_locks`
- Wenn eine Transaktion auf die Zeilensperre wartet:

```
 locktype      | relation | transactionid | pid   | mode          | granted
---------------+----------+---------------+-------+---------------+---------
 relation      | 70965    |               | 19539 | RowShareLock  | t
 tuple         | 70965    |               | 19539 | ExclusiveLock | t
 transactionid |          |     20476     | 19539 | ShareLock     | f
```

Wie Zeilensperren in `pg_locks` aussehen

- Wenn zwei Transaktionen auf die Zeilensperre warten:

```
 locktype      | relation | transactionid | pid   | mode          | granted
---------------+----------+---------------+-------+---------------+---------
 relation      |   70965  |               | 19539 | RowShareLock  |   t
 tuple         |   70965  |               | 19539 | ExclusiveLock |   t
 transactionid |          |     20476     | 19539 | ShareLock     |   f
 relation      |   70965  |               | 19636 | RowShareLock  |   t
 tuple         |   70965  |               | 19636 | ExclusiveLock |   f
```

Zeilensperren untersuchen

```
CREATE EXTENSION pgrowlocks;

SELECT * FROM pgrowlocks('tab');
 locked_row | locker | multi |     xids      |           modes            |   pids
------------+--------+-------+---------------+----------------------------+-----------
 (0,2)      |  20474 | f     | {20474}       | {"No Key Update"}          | {14785}
 (0,4)      |      1 | t     | {20474,20475} | {"For Key Share",Update}   | {14785,0}
```

- Durchsucht die ganze Tabelle ⇒ kann lange dauern
- Garantiert keinen konsistenten „Snapshot”
- Zeigt nur die Tupel, die man sehen kann — wenn man gerade eine Zeile verändert, sieht man die Sperre auf der Vorversion nicht

He! Was ist dieser `locker` mit der Nummer 1 in der zweiten Zeile?

MultiXacts

Was ist ein MultiXact?

- Ein Objekt, das mehrere Transaktionen („xact”) repräsentiert
- Einer der ärgerlichsten Namen in der PostgreSQL-Welt
- Notwendig, weil nur 4 Bytes in ein `xmax` passen, aber mehrere Transaktionen gleichzeitig eine Zeile sperren können
- Beschreibt mehrere Transaktionen und ihren Sperrengrad
- Muss persistiert werden und einen Absturz überleben

Wann werden MultiXacts erzeugt?

- Normalerweise bei der Verwendung von Fremdschlüsseln
  - Gleichzeitige Transaktionen fügen Zeilen ein, die dieselbe Zeile einer anderen Tabelle referenzieren (beide sperren die Zeile mit `KEY SHARE`)
  - Die von einem `INSERT` referenzierte Zeile wird gleichzeitig verändert (`KEY SHARE` und `NO KEY UPDATE`-Sperre)
- Auch möglich in einer einzelnen Transaktion mit Sub-Transaktionen
  - wenn die Sub-Transaktion eine Sperre der Transaktion eskaliert (Sub-Transaktionen haben auch Transaktionsnummern)

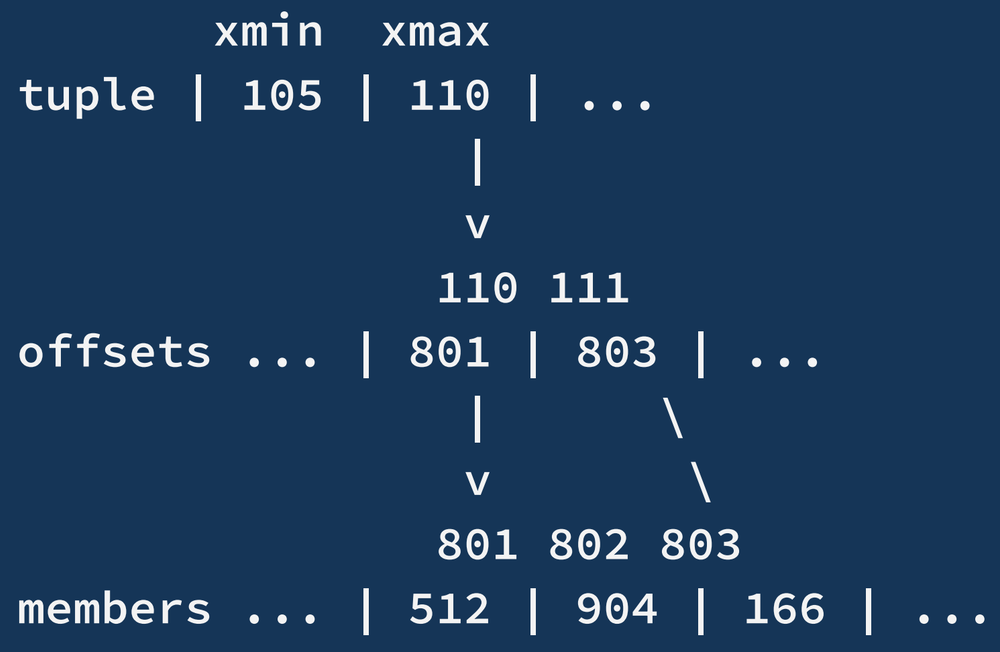
Beispiel eines MultiXact mit Subtransaktionen

```
BEGIN;
DO
$$BEGIN
    PERFORM id FROM rowlock WHERE id = 2 FOR NO KEY UPDATE;
    BEGIN
        DELETE FROM rowlock WHERE id = 2;
    EXCEPTION
        WHEN integrity_constraint_violation THEN
            NULL;
    END;
END;$$;
```

```
locked_row | locker | multi | xids          | modes                          | pids
-----------+--------+-------+---------------+--------------------------------+-----------
(0,2)      |   2    |   t   | {20479,20480} | {"For No Key Update",Update} | {15077,0}
```

Implementation von MultiXact

- In `pg_multixact` im PostgreSQL-Datendirectory persistiert
- Zwei Datenstrukturen: „`members`” und „`offsets`”



MultiXact-Problem 1: Performance

- PostgreSQL muss Mitglieder von MultiXacts nachschauen um zu wissen, wer eine Zeile gesperrt oder geändert hat
- MultiXact haben für „members” und „offsets” SLRU-Caches (Simple Least Recently Used)
- Wenn es viele MultiXacts gibt oder diese viele Mitglieder haben, können die Caches zu klein sein
- „wait events” für MultiXact offsets/members überwachen
  ⇒ pgWatch oder pg_wait_sampling einsetzen
- Caches können seit v13 in `pg_stat_slru` überwacht werden
- Seit v17 kann die Cache-Größe mit `multixact_offset_buffers` und `multixact_member_buffers` eingestellt werden

MultiXact-Problem 2: Überlauf von MultiXact-Nummern

- Müssen Commit überleben (ein Mitglied könnte von einem `UPDATE` stammen)
- Wie Transaktionsnummern aus einem 4-Byte-Zähler erzeugt, der irgendwann überläuft
- Nicht mehr benötigt, wenn Tupel „eingefroren” sind; in `pg_class.relminmxid` und `pg_database.datminmxid` verfolgt
- Können „Anti-Wraparound” Autovacuum-Läufe auslösen
- Nur ein Problem, wenn MultiXacts schneller als Transaktionen erzeugt werden

MultiXact-Problem 2: Überlauf von MultiXact-Nummern

- Überwachen mit

```
SELECT datname, mxid_age(datminmxid),
       current_setting('autovacuum_multixact_freeze_max_age') AS threshold
FROM pg_database;
```

Fragen

Why CYBERTEC?

Pure-play PostgreSQL Company

Specialised, International and Global Team

25 years as a Key Contributor to the PostgreSQL Community

Product & Service Around PostgreSQL

Fundamentally Independent Operated