

The anatomy of PostgreSQL row locks

Laurenz Albe

www.cybertec-postgresql.com



Senior Consultant Laurenz Albe

MAIL laurenz.albe@cybertec.at
PHONE +43 670 605 6265
WEB www.cybertec-postgresql.com



www.cybertec.at





CYBERTEC
MIGRATOR

CYPEX

CYBERTEC
PGEE

scalefield

**DATA
MASKING**

**WAL
BOUNCER**

pg_
TIMETABLE

PL/pgSQL_**sec**
CYBERTEC

POSTGRES
TDE

PG**WATCH**

ora_migrator

zheap
SERVING POSTGRES STORAGE

PG
**SHOW
PLANS**

POSTGRES
CONFIGURATOR

**PG
SQUEEZE**

YAIM

PATRONI
ENVIRONMENT SETUP

CYBERTEC
POSTGRES SERVICES & SUPPORT



The basics



Why this talk?

- everybody knows that there are row locks, but few people know
 - what lock levels exist and why
 - how row locks interact with foreign keys
 - how row locks are implemented
 - how PostgreSQL takes row locks
 - how to examine row locks
 - what the heck a multixact really is
 - why multixacts can be annoying

Title inspiration stolen from Gülçin Yıldırım Jelínek's talk about table locks!



Why do we need row locks?

- PostgreSQL MVCC often avoids the need for row locks
 - read and write don't conflict; the reader sees the old version
- but only one transaction can modify a row at a given time
 - ⇒ UPDATE and DELETE take exclusive row locks
- like all locks, row locks are held until transaction end



Working with row locks



How do you take a row lock?

- most of the time, you don't take row locks explicitly
- taken implicitly by UPDATE, DELETE and (sometimes) INSERT statements
- you can take them explicitly with
`SELECT ... FOR <lock mode>`
- taking explicit row locks that way is important on the default READ COMMITTED isolation level to guarantee data consistency through *pessimistic locking*: you lock the row while reading it, so no concurrent transaction can modify it to avoid *lost updates*



Lock modes for row locks

PostgreSQL knows four row lock levels:

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



Purpose of the different row lock levels

- KEY SHARE: you want to keep others from deleting the row or modifying a primary or unique key column
⇒ used for foreign key management
- SHARE: you want to keep others from modifying the row
⇒ not used, a vestige from bad old times
- NO KEY UPDATE: you intend to update the row, but won't modify a primary or unique key columns
⇒ used by UPDATE statements
- UPDATE: you intend to delete the row or modify a key column
⇒ used by DELETE and key modifying UPDATE statements



Row locks and foreign keys



A problem with referential integrity

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

Session 1:

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

Session 2:

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

Session 2 must block until session 1 commits or rolls back — but the statements modify different tables!



Solution for the referential integrity problem

- INSERT on the child table puts a KEY SHARE lock on the referenced row in the parent table
 - the KEY SHARE lock conflicts with the UPDATE lock that DELETE takes
- ⇒ integrity is guaranteed

Note that this is not the only way to solve the problem: Oracle locks the referencing rows when you delete from the parent table



Choosing the correct lock level for pessimistic locking

- most people lock rows as follows if they intend to UPDATE

```
SELECT ... FROM p WHERE ... FOR UPDATE;
```

- but that blocks an INSERT referencing that row via a foreign key
- **the correct lock level is FOR NO KEY UPDATE**
- FOR UPDATE is only correct if you intend to DELETE or to UPDATE a key column

Historical note: before NO KEY UPDATE and KEY SHARE were invented, PostgreSQL used UPDATE and SHARE locks — hence the names!



Implementation



How does PostgreSQL usually store locks?

- shared memory lock table
- allocated at server start
- number of slots in v18:

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

- by default, room for 8704 locks
- can be inspected with the `pg_locks` view



The problem with row locks in the lock table

- while there are (usually) not so many tables, you can have many rows
- if you store row locks in the lock table, the table could easily fill up
- to deal with that problem, you'd have to escalate row locks to page or table locks
- that is difficult, bad for concurrency and can lead to surprising deadlocks
- consequently, **PostgreSQL doesn't permanently store row locks in the lock table**



How does PostgreSQL implement row locks?

- **row locks are stored in the database row itself**
 - advantage: can have arbitrarily many without lock escalation
 - disadvantage: locking a row makes the page dirty and lead to writes

Tuple header:

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

- the locker is stored in xmax
 - xmax is normally used for tuple visibility
(but you only ever lock rows that are visible anyway)
 - problem with more than one locker (later!)
- flags in infomask store the lock mode and other details



How row locks are taken

- if there is no conflicting lock on the row, update `xmax` and `infomask` and be done
- otherwise, take a tuple lock in the shared memory lock table (prevents concurrent modifications and forms a proper queue)
- wait for all conflicting transactions to end
- fetch the latest row version and start over
- update `xmax` and `infomask`
- release the tuple lock in the lock table



Row locks in pg_locks

- a single row lock with no waiters is not visible in pg_locks
- if one transaction waits for a row lock:

locktype	relation	transactionid	pid	mode	granted
relation	70965		19539	RowShareLock	t
tuple	70965		19539	ExclusiveLock	t
transactionid		20476	19539	ShareLock	f

- if two transactions wait for a row lock:

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



Inspecting row locks

```
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}

- scans the whole table $\Rightarrow$ slow on large tables
- no consistent snapshot guaranteed
- only shows the tuples visible to you — so if you just updated a row, you won't see the lock on the old row version

Hey! What is that locker number 1 in the second row?



MultiXacts



What is a MultiXact?

- object that represents multiple transactions
- one of the most annoying names in the PostgreSQL universe
- necessary because `xmax` only fits four bytes, but several concurrent transactions can lock the same row
- describes multiple transactions and their lock level
- needs to be persistent and crash safe



When are MultiXacts created?

- typically in connection with foreign keys
 - if concurrent transactions insert rows that reference the same row in another table (each takes a KEY SHARE lock on the referenced row)
 - if the row referenced by an insert is concurrently updated (KEY SHARE and NO KEY UPDATE lock)
- also possible in a single transaction with subtransactions
 - if the subtransaction escalates a lock taken by the transaction (subtransactions have transaction numbers as well)



Example of a MultiXact with a subtransaction

```
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}



MultiXact implementation

- persisted in `pg_multixact` in the data directory
- two data structures: members and offsets

```
      xmin  xmax
tuple | 105 | 110 | ...
      |
      v
      110   111
offsets ... | 801 | 803 | ...
      |           \
      v           \
      801   802   803
members ... | 512 | 904 | 166 | ...
```



MultiXact problem 1: performance

- PostgreSQL needs to look up MultiXact members to determine who locked or updated a row
- MultiXact have “SLRU” (Simple Least Recently Used) caches for offsets and members
- if there are many MultiXacts or they have many members, the caches can be too small
- monitor MultiXact offset/member wait events in `pg_stat_activity`
⇒ use `pgWatch` or `pg_wait_sampling`
- can be viewed in `pg_stat_slru` from v13 on
- cache size can be tuned with `multixact_offset_buffers` and `multixact_member_buffers` from v17 on



MultiXact problem 2: MultiXact ID wraparound

- must be kept around after commit (member could be from an UPDATE)
- similar to transaction IDs; generated from a 4-byte unsigned integer counter that wraps around at some point
- become obsolete as tuples are frozen, tracked in `pg_class.relminmxid` and `pg_database.datminmxid`
- can trigger anti-wraparound autovacuum runs
- normally only a problem with some workloads that generate MultiXacts faster than transactions
- monitor with

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



Questions

