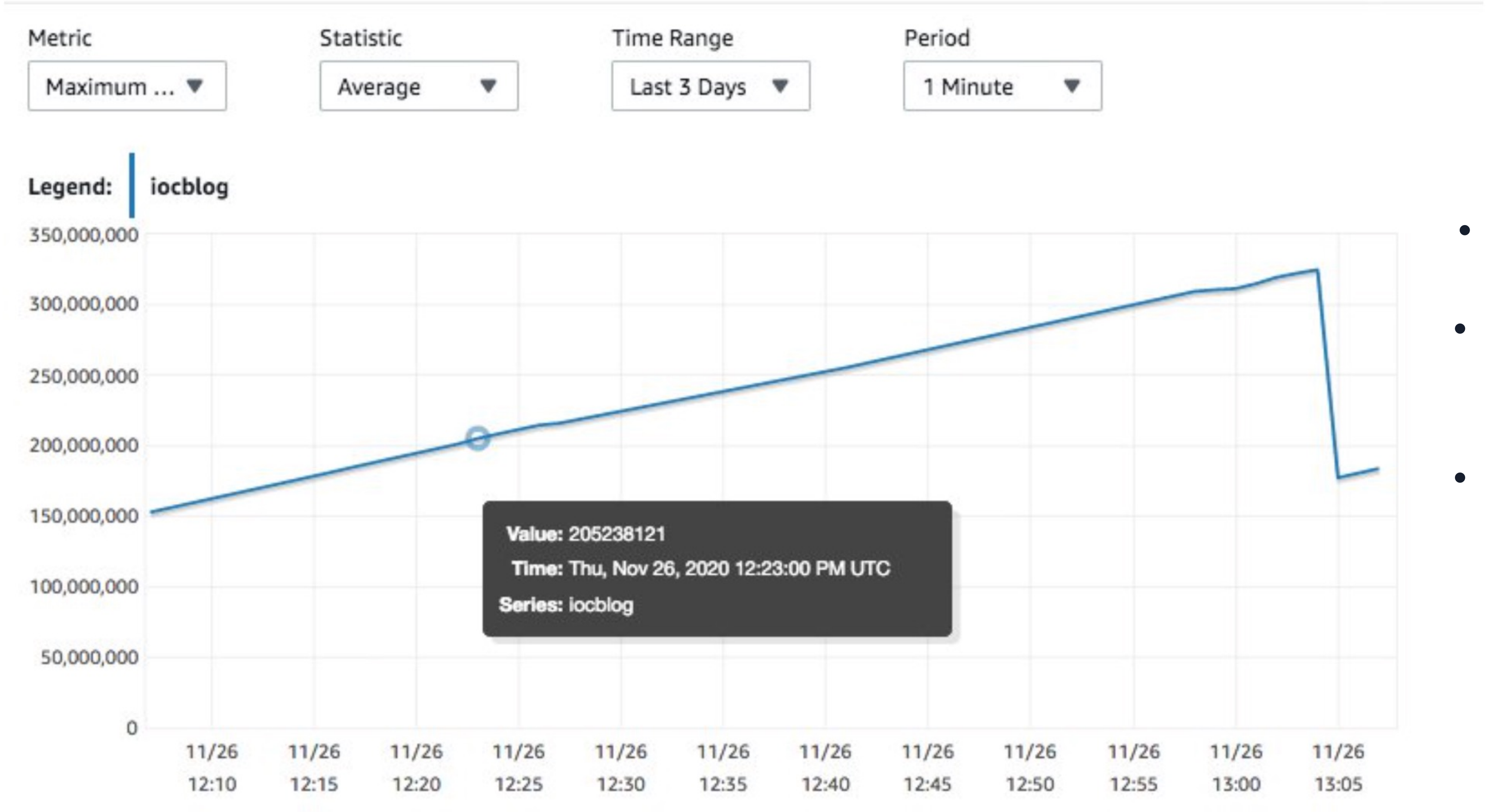
MultiXacts in PostgreSQL

usage, side effects, and monitoring

Divya Sharma

Sr. PostgreSQL Database Engineer

MaximumUsedTransactionIDs CloudWatch metric



Value: 205238121
Time: Thu, Nov 26, 2020 12:23:00 PM UTC
Series: iocblog

- Depicts utilization of normal xids
- autovacuum_freeze_max_age (defaults to 200 mil)
- Transaction ID Wraparound

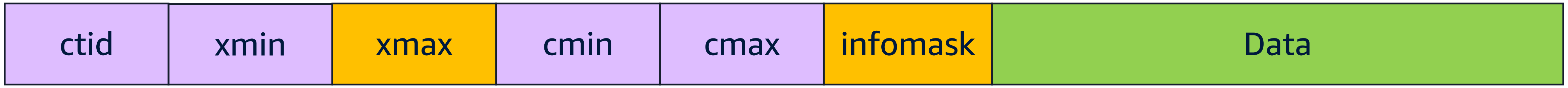
Assigned to individual transactions performing write operations. Each transaction has **one XID** used in tuple headers **(xmin/xmax)**

What if there are multiple transactions locking the same row concurrently?

Introduction to MultiXacts in PostgreSQL

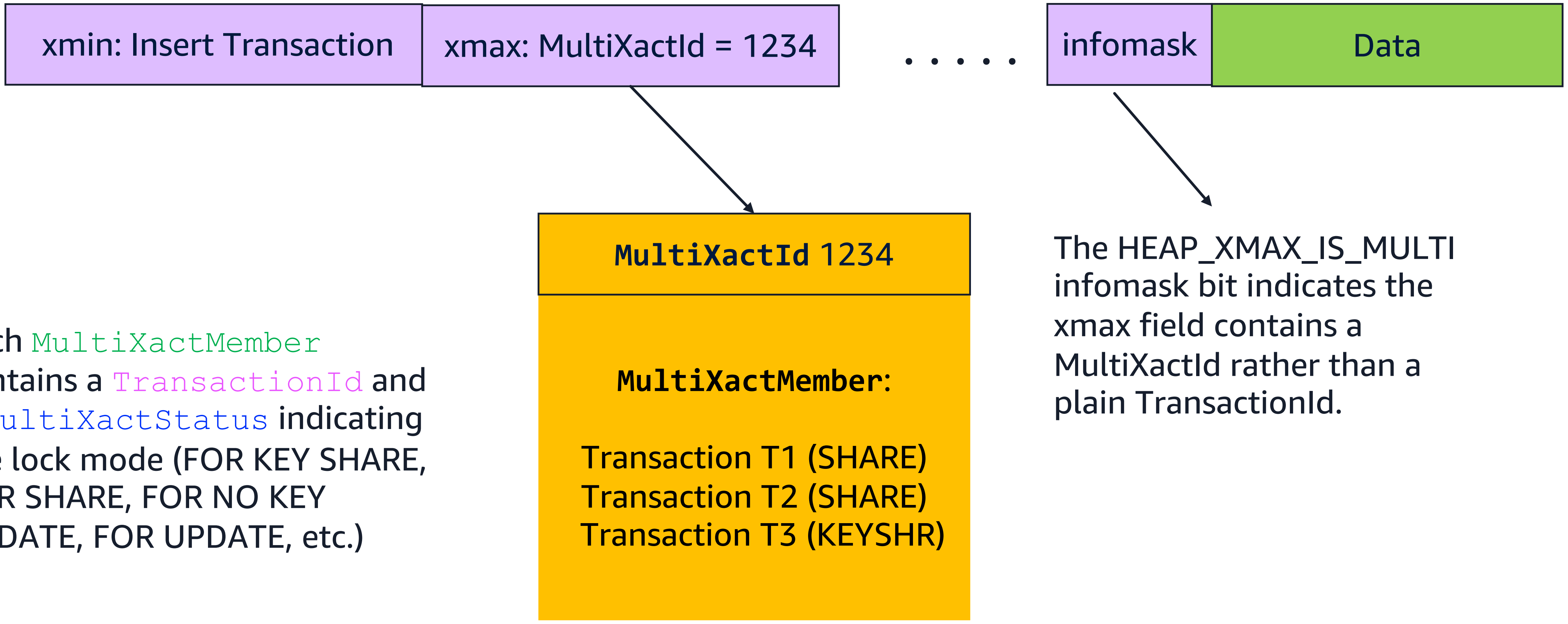
Introduction to MultiXacts in PostgreSQL

POSTGRESQL EXTENSIONS



When multiple transactions attempt to lock the same row simultaneously, PostgreSQL turns to a specialized structure called MultiXact IDs

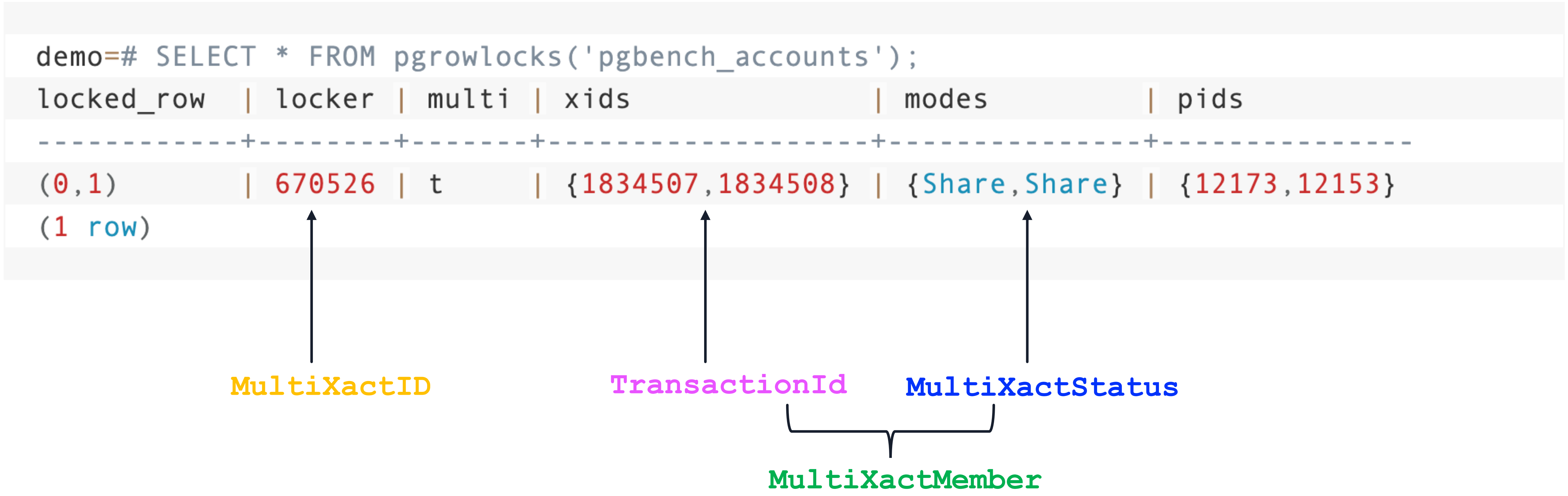
Introduction to MultiXacts in PostgreSQL



The HEAP_XMAX_IS_MULTI infomask bit indicates the xmax field contains a MultiXactId rather than a plain TransactionId.

Each `MultiXactMember` contains a `TransactionId` and a `MultiXactStatus` indicating the lock mode (FOR KEY SHARE, FOR SHARE, FOR NO KEY UPDATE, FOR UPDATE, etc.)

pgrowlocks extension to view MultiXact information



```
demo=# SELECT * FROM pgrowlocks('pgbench_accounts');
locked_row  | locker | multi | xids              | modes         | pids
------------+--------+-------+-------------------+---------------+---------------
(0,1)       | 670526 | t     | {1834507,1834508} | {Share,Share} | {12173,12153}
(1 row)
```

Operations using MultiXacts

Operations using MultiXacts

Foreign Keys and MultiXacts

Concurrent inserts referencing a parent table cause shared locks on parent rows.

Under concurrent usage those locks are converted to MultiXacts

SELECT…FOR SHARE and MultiXacts

Prevents updates and deletions while allowing shared locks.

Triggers MultiXacts when multiple sessions lock the same row.

Sub-transactions and MultiXacts

Sub-transactions from explicit savepoints and PL/pgSQL EXCEPTION clauses.

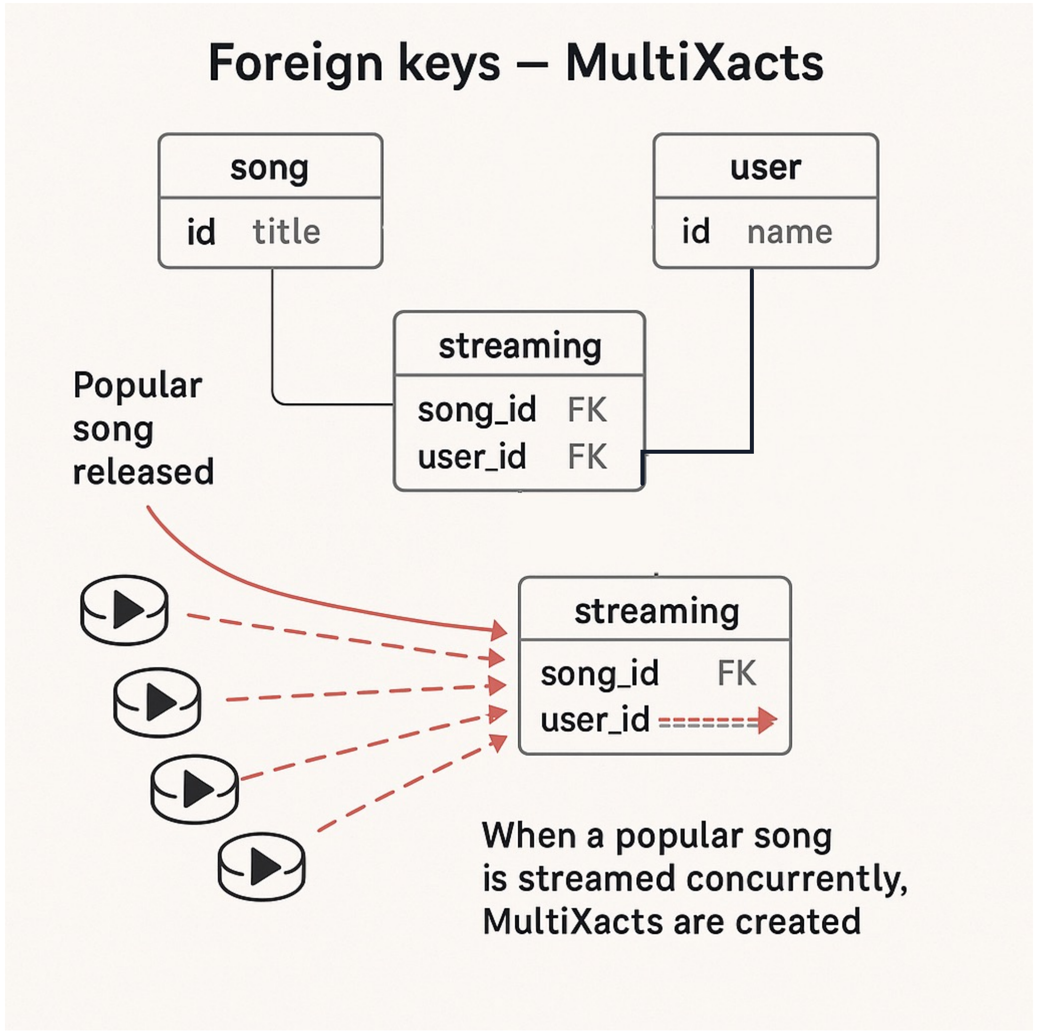
Each sub-transaction has a separate XID, leading to MultiXact usage.

Drivers and Abstraction Layers

Automatic savepoints and transaction wrappers can generate numerous MultiXact wait events.

Example : PostgreSQL JDBC autosave option

Operations using MultiXacts – Foreign Keys



You get a KEY SHARE lock **on the remote table** if you insert or delete a row in the table with the foreign key constraint.

That lock will prevent modifications on keys **in the remote table**.

MultiXacts Storage, Wraparound and Vacumming

MultiXacts Storage

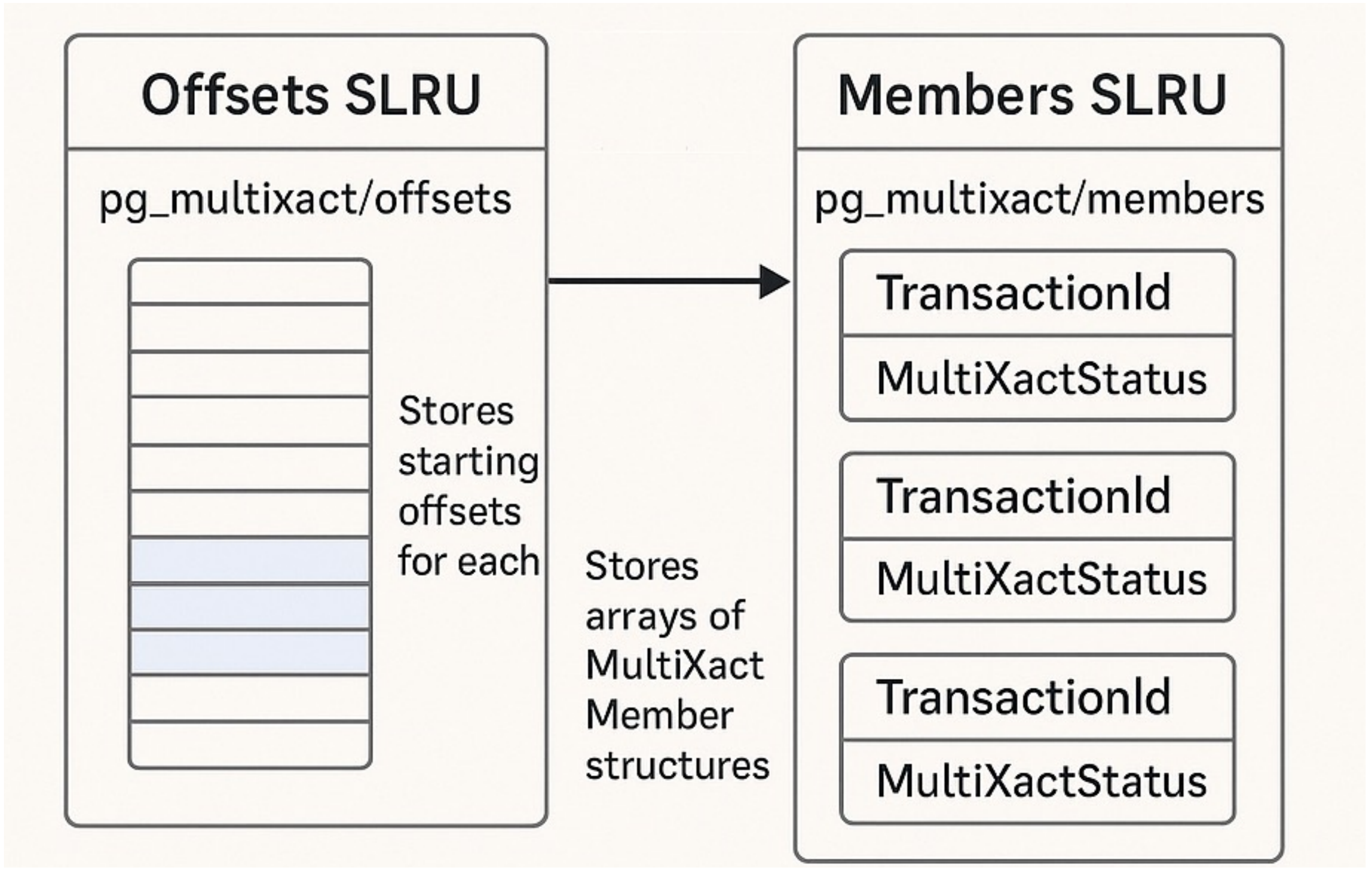
- pg_multixact is the subdirectory under the data directory PGDATA
- pg_multixact/offsets : Stores the starting offset for each MultiXactId member
- pg_multixact/members : Stores arrays of MultiXactMember structures (TransactionId + status flags)
- Needs to be stored on disk but is also cached in shared memory for performance reasons – SLRUs buffers in memory used

`MultiXactId` 1234

`MultiXactMember`:

Transaction T1 (SHARE)
Transaction T2 (SHARE)
Transaction T3 (KEYSHR)

MultiXacts Storage



MultiXactStatus indicates the lock mode (FOR KEY SHARE, FOR SHARE, FOR NO KEY UPDATE, FOR UPDATE, etc.)

MultiXacts Wraparound

- Like transaction IDs, multixacts use 32-bit counters and require **wraparound management**

- No new MXIDs generated once there are fewer than three million left until wraparound (2.1 Billion) **OR** if space occupied by pg_multixacts is ~20GB.

- **Wraparound can happen for normal transaction IDs and MultiXacts independently.**

```
ERROR: multixact "members" limit exceeded
DETAIL: This command would create a multixact with X members, but the remaining
space is only enough for 0 members.
```

Autovacuum for MultiXacts wraparound prevention

To prevent [MultiXact wraparound](), the autovacuum daemon invokes an autovacuum worker based on one of the following conditions (whichever happens first):

- If the MultiXact IDs consumed reach the threshold `autovacuum_multixact_freeze_max_age` (defaults to 400 million)
- If the amount of disk storage occupied by the MultiXact directories crosses an internal threshold (10 GB)

The system will launch autovacuum processes to prevent wraparound even when autovacuum is otherwise disabled

Manual Vacuum for MultiXacts

- A standard **VACUUM** command will only release MultiXact IDs older than vacuum_multixact_freeze_min_age and remove their corresponding files from the pg_multixact directories.

- Running the **VACUUM FREEZE** command releases all MultiXact IDs that are completed and is used to prevent a MultiXact wraparound.

Monitoring MultiXacts

Monitoring MultiXacts

- Tracking MultiXact ID Age
- Monitoring size for pg_multixact directory
- Monitoring and analyzing SLRU buffer activity
- Identifying wait events related to MultiXacts

Tracking MultiXact ID Age

Find the databases that are holding the highest MultiXact age using the following query :

```
SELECT datname, mxid_age(datminmxid)FROM pg_database ORDER BY 2 desc;
```

then connect to that database to find the tables with the oldest MultiXact age:

```
SELECT   relname, mxid_age(relminmxid) FROM pg_class

WHERE    relminmxid::text::bigint > 1

ORDER BY 2 DESC LIMIT    20;
```

If your workload isn't generating MultiXact IDs, then the mxid_age(relminmxid) value for your tables will be zero.

Monitoring size for pg_multixact directory

- The members storage can grow up to about **20GB before reaching wraparound**, with aggressive vacuum starting when it exceeds about 10GB
- For **Amazon Aurora PostgreSQL-Compatible Edition**, you can use the **aurora_stat_utils extension** to examine the storage occupied by subdirectories under pg_multixacts.
- For **RDS PostgreSQL**, you can use the **rds_tools extension** - pg_ls_multixactdir() function to monitor the directory disk space usage.

Monitoring size for pg_multixact directory

- For **RDS PostgreSQL**, you can use the **rds_tools extension** - pg_ls_multixactdir() function to monitor the directory disk space usage.

```
postgres=> create extension rds_tools;
CREATE EXTENSION
```

```
postgres=> SELECT pg_size_pretty(coalesce(sum(size), 0)) AS members_size
FROM rds_tools.pg_ls_multixactdir ()
WHERE name LIKE 'pg_multixact/members%';
members_size
--------------
 12 kB
(1 row)
```

Monitoring and analyzing SLRU buffer activity

pg_stat_slru statistics view (version 13+) that shows information about SLRU caches, including blocks hit and blocks read counters :

```
demo=# SELECT * FROM pg_stat_slru WHERE name in ('MultiXactMember','MultiXactOffset');
      name       | blks_zeroed | blks_hit | blks_read | blks_written | blks_exists | flushes | truncates |          stats_
-----------------+-------------+----------+-----------+--------------+-------------+---------+-----------+------------------
 MultiXactMember |           0 |        9 |         1 |            1 |           0 |       4 |         0 | 2024-09-13 14:2
 MultiXactOffset |           0 |       12 |         2 |            1 |           4 |       4 |         0 | 2024-09-13 14:2
(2 rows)
```

This output won't tie to which sessions are using MultiXacts, you can only confirm the fact that MultiXacts are being used and monitor the rate of usage.

SLRU search – operating with atomic ops

1. Scan linearly the array of buffers to see if one contains the page we want
2. If we find it, we're done
3. If not, the scan has chosen a "victim" buffer to evict (least recently used)
    1. Evict it, leaving buffer free
    2. Load our page onto our buffer
    3. Increment "recently used" counter using atomic ops
4. Now the page in buffer can be processed

Ref : What is an SLRU anyway? - PGConf DE 2025

Optimizing search with SLRU banks (v17+)

1. Scan linearly the array of buffers of the bank that contains the page to see if one contains the page we want
2. If we find it, we're done
3. If not, the scan has chosen a "victim" buffer to evict (least recently used)
    1. Evict it, leaving buffer free
    2. Load our page onto our buffer
    3. Increment "recently used" counter using atomic ops
4. Now the page in buffer can be processed

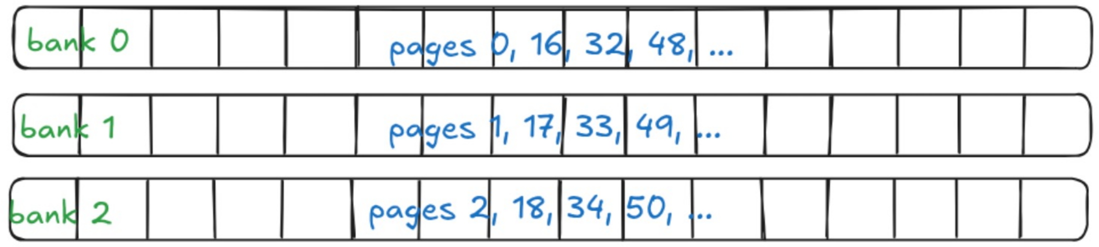
Ref : [What is an SLRU anyway?](#) - PGConf DE 2025

Optimizing search with SLRU banks (v17+)

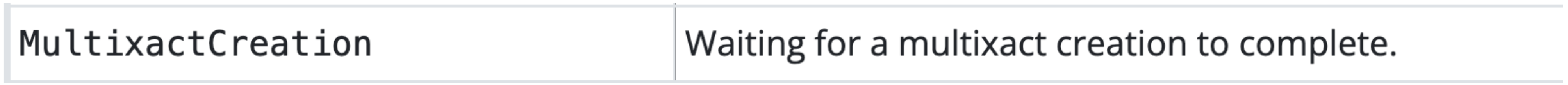
- Partitions SLRU buffer slots into multiple "banks" to reduce lock contention.

- By using separate bank locks instead of a single global lock, multiple backends can:

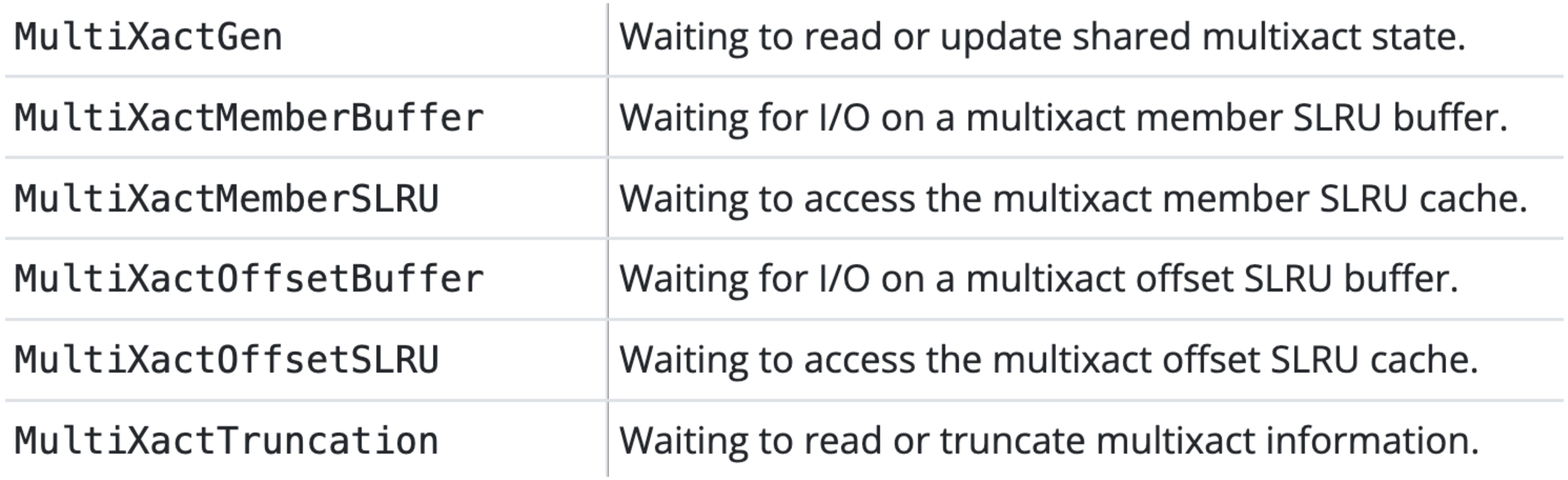
  - Access different SLRU pages simultaneously if they're in different banks
  - Reduce lock contention during high-concurrency workloads



Identifying wait events related to MultiXacts



`MultixactCreation`	Waiting for a multixact creation to complete.

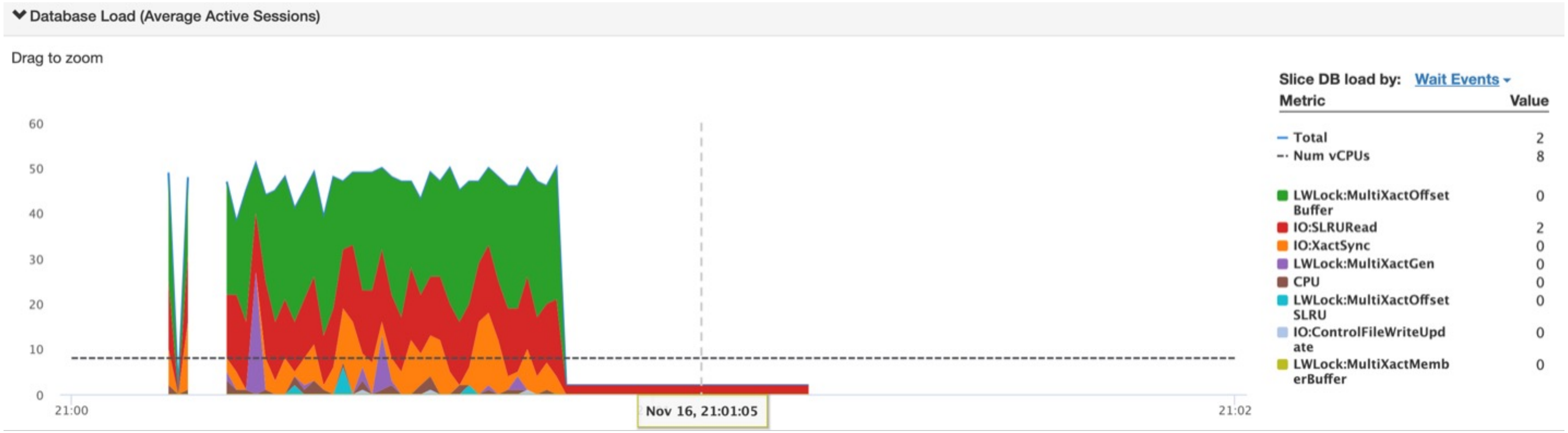


`MultiXactGen`	Waiting to read or update shared multixact state.
`MultiXactMemberBuffer`	Waiting for I/O on a multixact member SLRU buffer.
`MultiXactMemberSLRU`	Waiting to access the multixact member SLRU cache.
`MultiXactOffsetBuffer`	Waiting for I/O on a multixact offset SLRU buffer.
`MultiXactOffsetSLRU`	Waiting to access the multixact offset SLRU cache.
`MultiXactTruncation`	Waiting to read or truncate multixact information.

Identifying wait events related to MultiXacts

Wait events (checked through pg_stat_activity or Performance Insights) can indicate if database workload activity is using the MultiXact mechanism heavily.

Visible sometimes when aggressive vacuum is run to clean up for MultiXacts.



Slice DB load by: Wait Events

Metric	Value
— Total	2
-- Num vCPUs	8
🟩 LWLock:MultiXactOffsetBuffer	0
🟥 IO:SLRURead	2
🟧 IO:XactSync	0
🟪 LWLock:MultiXactGen	0
🟫 CPU	0
🟦 LWLock:MultiXactOffsetSLRU	0
🟦 IO:ControlFileWriteUpdate	0
🟨 LWLock:MultiXactMemberBuffer	0

Best practices for managing MultiXacts

Best practices for managing MultiXacts

Targeted VACUUM FREEZE

Perform targeted VACUUM FREEZE on affected tables or partitions to immediately reduce MultiXact bloat.

Adjust Autovacuum Settings

Adjust autovacuum settings per table, lowering autovacuum_multixact_freeze_max_age to trigger more frequent cleanup.

Best practices for managing MultiXacts

Optimise Memory Parameters

Increase memory parameters to cache MultiXact, reducing disk I/O.

Up to Aurora PostgreSQL 16: multixact_offsets_cache_size, multixact_members_cache_size

PostgreSQL 17+: multixact_offset_buffers, multixact_member_buffers

Reduce Long-Running Transactions

Reduce long-running transactions, which delay vacuum cleanup, by using connection poolers.

Set idle_in_transaction_session_timeout.

Optimise application logic.

Summary

Key Takeaways

- MultiXacts represent multiple transactions locking the same row concurrently
- Operations using MultiXacts : SELECT .. FOR KEY SHARE, subtransactions, savepoints etc.
- Also a 32 bit counter; Wrap limit: 2.1 Billion. Separate wraparound for member storage (~20GB space).
- Running the **`VACUUM FREEZE`** command releases all MultiXact IDs that are completed and is used to prevent a MultiXact wraparound.
- Reduce long running idle transactions (idle_in_transaction_session_timeout)
- v17+ has optimized search with SLRU banks, so upgrade!

Useful Links

https://deepwiki.com/postgres/postgres

https://aws.amazon.com/blogs/database/multixacts-in-postgresql-usage-side-effects-and-monitoring/

Thank you!

Divya Sharma

Sr. PostgreSQL Database Engineer, AWS

https://www.linkedin.com/in/divyasharma95/