HIGH-CONCURRENCY DISTRIBUTED SNAPSHOTS

ANTS AASMA

pgconf.eu 2024

Hello

About me

Ants Aasma

Senior Database Consultant

13 years of helping people make PostgreSQL run fast

About this talk

- Overview of database concurrency.
- How we solve this today in PostgreSQL.
- Proposal for how to do it in the future.

What are snapshots

Lets start with ACID

We all love transactions!

- Atomicity - all or nothing!
- Consistency - there are rules!
- Isolation - none of this concurrency weirdness!
- Durability - stuff doesn't just disappear!

MVCC core tenets

- When a query runs it sees database state as unchanging.

MVCC core tenets

- When a query runs it sees database state as unchanging.
- Meanwhile we want to perform updates without waiting for this query.

MVCC core tenets

- When a query runs it sees database state as unchanging.
- Meanwhile we want to perform updates without waiting for this query.
- After the updates complete we want to run queries that can see these writes.

MVCC core tenets

- When a query runs it sees database state as unchanging.
- Meanwhile we want to perform updates without waiting for this query.
- After the updates complete we want to run queries that can see these writes.
- Original query can still see original state.

MVCC core tenets

- When a query runs it sees database state as unchanging.
- Meanwhile we want to perform updates without waiting for this query.
- After the updates complete we want to run queries that can see these writes.
- Original query can still see original state.
- Therefore we need to have different versions of rows visible to different queries.

Snapshots to the rescue

- Tag every row version with transactions that added it and removed it.
- When starting a read, create a snapshot datastructure.
  - `XidInMVCCSnapshot(xid, snapshot) -> bool`
- Represents a point in time in the past.
- Divides world into past and future.
- Ideally, snapshots should agree on the order of things.
  - If we have snapshot that thinks txA is past, txB is future, then it should be impossible to get a snapshot that thinks txB is past, txA is future.

Complicating factors

- Writing transactions can run for a long time.

Complicating factors

- Writing transactions can run for a long time.
- They may also run for a short time.

Complicating factors

- Writing transactions can run for a long time.
- They may also run for a short time.
- Transaction completion order does not match start order (xid order).

Complicating factors

- Writing transactions can run for a long time.
- They may also run for a short time.
- Transaction completion order does not match start order (xid order).
- While writing row versions we don’t know the completion order.

Complicating factors

- Writing transactions can run for a long time.
- They may also run for a short time.
- Transaction completion order does not match start order (xid order).
- While writing row versions we don’t know the completion order.
- Not feasible to go back and rewrite everything.

SQL isolation levels

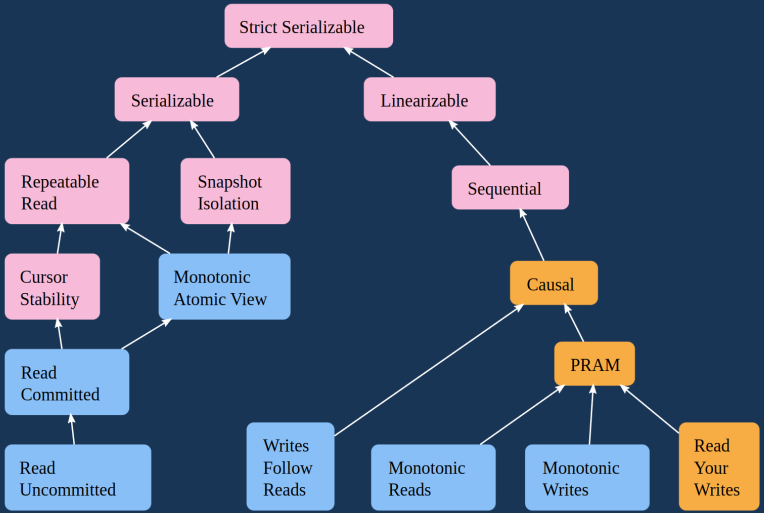
```
BEGIN ISOLATION LEVEL READ UNCOMMITTED
                      READ COMMITTED
                      REPEATABLE READ
                      SERIALIZABLE
```


SQL isolation levels

```
BEGIN ISOLATION LEVEL READ UNCOMMITTED
                      READ COMMITTED
                      REPEATABLE READ
                      SERIALIZABLE
```

- Not well defined.
- Does not capture the space of possibilities.

Zoo of consistency levels



More consistent is not always more better

Higher consistency levels have fundamental tradeoffs that are impossible to engineer out.

More consistent is not always more better

Higher consistency levels have fundamental tradeoffs that are impossible to engineer out.

- For availability

More consistent is not always more better

Higher consistency levels have fundamental tradeoffs that are impossible to engineer out.

- For availability
- For latency

More consistent is not always more better

Higher consistency levels have fundamental tradeoffs that are impossible to engineer out.

- For availability
- For latency

Users must be able to pick a level suitable for their problem.

Where we are today

How snapshots work in PostgreSQL

- Every writing transaction publishes their xid in shared memory (`ProcArray`).

How snapshots work in PostgreSQL

- Every writing transaction publishes their xid in shared memory (`ProcArray`).
- When acquiring a snapshot (`GetSnapshotData()`) use it to fill in the following:

How snapshots work in PostgreSQL

- Every writing transaction publishes their xid in shared memory (`ProcArray`).
- When acquiring a snapshot (`GetSnapshotData()`) use it to fill in the following:
  - `xmax` - latest completed xid + 1

How snapshots work in PostgreSQL

- Every writing transaction publishes their xid in shared memory (`ProcArray`).
- When acquiring a snapshot (`GetSnapshotData()`) use it to fill in the following:
  - `xmax` - latest completed xid + 1
    - `xid >= xmax` -> future

How snapshots work in PostgreSQL

- Every writing transaction publishes their xid in shared memory (`ProcArray`).
- When acquiring a snapshot (`GetSnapshotData()`) use it to fill in the following:
  - `xmax` - latest completed xid + 1
    - `xid >= xmax` -> future
  - `xmin` - earliest running xid

How snapshots work in PostgreSQL

- Every writing transaction publishes their xid in shared memory (`ProcArray`).
- When acquiring a snapshot (`GetSnapshotData()`) use it to fill in the following:
  - `xmax` - latest completed xid + 1
    - `xid >= xmax` -> future
  - `xmin` - earliest running xid
    - `xid < xmin` -> past

How snapshots work in PostgreSQL

- Every writing transaction publishes their xid in shared memory (`ProcArray`).
- When acquiring a snapshot (`GetSnapshotData()`) use it to fill in the following:
  - `xmax` - latest completed xid + 1
    - `xid >= xmax` -> future
  - `xmin` - earliest running xid
    - `xid < xmin` -> past
  - `xip` - list of running transactions

How snapshots work in PostgreSQL

- Every writing transaction publishes their xid in shared memory (`ProcArray`).
- When acquiring a snapshot (`GetSnapshotData()`) use it to fill in the following:
  - `xmax` - latest completed xid + 1
    - `xid >= xmax` -> future
  - `xmin` - earliest running xid
    - `xid < xmin` -> past
  - `xip` - list of running transactions
    - If in this list, then future, otherwise past.

How snapshots work in PostgreSQL

- Every writing transaction publishes their xid in shared memory (`ProcArray`).
- When acquiring a snapshot (`GetSnapshotData()`) use it to fill in the following:
  - `xmax` - latest completed xid + 1
    - `xid >= xmax` -> future
  - `xmin` - earliest running xid
    - `xid < xmin` -> past
  - `xip` - list of running transactions
    - If in this list, then future, otherwise past.
- Still have to check `pg_xact` whether it was successful.

How snapshots work in PostgreSQL

- Every writing transaction publishes their xid in shared memory (`ProcArray`).
- When acquiring a snapshot (`GetSnapshotData()`) use it to fill in the following:
  - `xmax` - latest completed xid + 1
    - `xid >= xmax` -> future
  - `xmin` - earliest running xid
    - `xid < xmin` -> past
  - `xip` - list of running transactions
    - If in this list, then future, otherwise past.
- Still have to check `pg_xact` whether it was successful.
- Visibility order determined by `ProcArrayLock` acquisition.

ProcArray scalability issues

- Every commit acquires `ProcArrayLock` exclusively.
- Every read scans the whole `ProcArray` while holding a share lock.
- Size of `ProcArray` = number of connections.
  - More CPUs and more IO throughput means more connections needed.
- Larger proc array takes longer to scan.
- More writing transactions means more time spent locking exclusively.

Things we have done to make it faster

- New running transactions are published in a lock-free manner.
- Group commit batch updates ProcArray for many committers at once.
- Major improvements in PG14 by Andres Freund.
  - xids are stored as a dense array for faster scanning.
  - Snapshot contents are cached until next commit.

Things we have done to make it faster

- New running transactions are published in a lock-free manner.
- Group commit batch updates ProcArray for many committers at once.
- Major improvements in PG14 by Andres Freund.
  - xids are stored as a dense array for faster scanning.
  - Snapshot contents are cached until next commit.
- For most workloads works well enough

Things we could still do

- Vectorize the main loop in `GetSnapshotData()`.
- Lock free snapshot cache.
- Incremental snapshot creation.

Subtransactions

- Subtransactions mean multiple xids per transaction.
  - Potentially unlimited.
- Limited space in shared memory to track this.
- If filled up need a `Subtrans` lookup for every visibility check in `[xmin, xmax)`.

Consistency? Eventually...

- On primary the order of transactions is determined by ProcArray order
- On standby the order is determined by Commit WAL record order

Consistency? Eventually...

- On primary the order of transactions is determined by ProcArray order
- On standby the order is determined by Commit WAL record order

```
CommitTransaction()
  RecordTransactionCommit()
    XactLogCommitRecord()
    if (synchronous_commit) // can be set per transaction
      XLogFlush()
      SyncRepWaitForLSN() // these two can take a looong time
  ProcArrayEndTransaction()
```


Pick three

- Commit order matches on primary and replica
- No wait when synchronous_commit = off
- Read-your-write consistency
- Single WAL record for commit

The non-synchronous commit

Problem scenario:

1. Client tries to book a room.
2. Synchronous commit blocks.
3. Client connection fails, commit becomes visible.
4. Client reconnects, checks that booking commit succeeded.
5. There is a failover, replica does not have that commit.

Does not distribute

- For sharded databases would be nice to get a consistent snapshot.
- Would like to have ACID for cross-shard transactions:
  - If a tx visible on shard A it should also be visible on B
  - Transactions should not disappear and re-appear
- Having a distributed ProcArray and global locking does not scale:
  - Snapshots get even bigger.
  - Even more write transactions.
  - More likelihood of node failures.
- Consistent order of durability and visibility becomes even more important.

CSN snapshots

Core idea

- On transaction commit assign a Commit Sequence Number (CSN).
  - Thinking of it as a commit time is not totally wrong.
- Should only go forward, not backward.
- Store this in a way that we can easily calculate `xid -> csn`.
- A snapshot is just the latest committed CSN value.

Not a new idea

- Similar stuff is present all over distributed database landscape.
  - Google Spanner
  - YugabyteDB
  - CockroachDB

What to use as the CSN

- Free to use anything that fulfills the requirements.
- Could use a simple counter (we already have it as `SerCommitSeqNo`).
- Could use Commit record WAL position (we need to assign it anyway).
- Could use a combination of a monotonic wall clock and logical counter (see HybridTime).

Invariants

1. After a commit becomes visible no transaction with a lower CSN can commit.
   - Needed for immutable snapshots.
2. After a commit returns, all new snapshots should get equal or bigger CSN.
   - Read your writes. (can be relaxed a bit by applying Lamport timestamp)
3. After a read completes all subsequent reads must see equal or higher CSN.
   - Transactions don’t disappear and re-appear.

Storing XID to CSN mapping

- Simplest answer: add CSN SLRU.
- 8 byte CSN @ 100k TPS = 800 KB/s
  - 100k TPS = wraparound in 6h
- We already have `CommitTs`, maybe combine?
- Can replace `Xact` SLRU, or could be compressed to `Xact` after global xmin.
- Might be a performance issue for mixed short-long tx workloads, see `SubTrans` SLRU.
  - Some ideas how to get around it.

Adding concurrency

Main workflow:

```
MyCSN = AcquireCSN();
RecordXidToCsnMapping(MyXid, MyCSN);
WaitForDurability()
UpdateVisibleCSN(MyCSN)
```

- Definitely want to do the wait concurrently.
- Updating visible CSN needs to happen in CSN AcquireCSN order.
- Build a queue, wait on anyone ahead of us, if we are first, release everyone already waiting behind us.
  - Analogous to ProcArray group update.
- Fixes the non-synchronous commit.

Visibility checks

- Lookup XID in CSN mapping, compare with value in snapshot.

```
bool
XidVisibleToSnapshot(TransactionId xid, Snapshot snapshot)
{
  return LookupXidCsn(xid) <= snapshot->csn;
}
```

- Sometimes can skip lookups:
  - xmax can be used to reject early
  - some approximation of xmin is also useful

Subtransactions

- Option 1: Tag every committed subxid with CSN on commit.
- Option 2: Carve a bit of CSN space to identify subtransactions.
  - Lookup parent on visibility check.
  - Update XID to CSN mapping.
- Can get rid of `SubTrans` SLRU?

Resolving the synchronous commit quadrilemma

- `synchronous_commit = off` will wait for durability of anyone that is commiting ahead of us.
- Add `synchronous_visibility` that allows user to “fire and forget” write transactions.
  - (working title)
  - When disabled, just skip waiting in `UpdateVisibleCSN()` and let someone else make this transaction visible.
  - Not possible in all cases (DDL)
- Optionally add a way to “see into the future” by reading non-durable transactions
  - `BEGIN ISOLATION LEVEL READ UNCOMMITTED`
- Only affects users that run a mix of `synchronous_commit` and expect to see the results.

In a distributed system

- Some synchronization is needed to ensure that all other commits < commit CSN are done committing.
- Coordinating with every node on every commit is undesirable.
- Spanner uses realtime clock dervived timestamp with error bounds for CSN.
  - On commit waits until everybody in cluster must agree commit timestamp is in the past.
  - Adds non-trivial amount of latency.
- YugabyteDB HybridTime uses NTP clock with a logical counter on top. Ensures this never goes backwards.
  - By eagerly sharing this can get illusion of consistency with no waiting.
  - Can still do the waiting if so desired.
- Reading from replicas still needs to wait for WAL replay.

Hybrid snapshots

Preface

Maybe this complexity is not needed.

Almost certainly a bad idea for initial version.

Demonstrates potential solutions to problems that may or may not become important.

Core insight

Given a snapshot (xmin, csn, xmax) we can build the same snapshot that we would have gotten from `ProcArray` by scanning XidToCSN mapping.

```
for (xid = xmin; xid < xmax; xid++)
    if (LookupXidCsn(xid) > snapshot->csn)
        xip[n++] = xid;
```


Core insight part 2

Snapshots can be converted incrementally by keeping track of a threshold:

```
struct SnapshotData
{
    TransactionId xmax
    CSN csn;
    TransactionId csn_xmin;
    TransactionId *xip;
    TransactionId xmin;
    //...
}
```


But why?

- XidToCSN lookups can get expensive when done per row.
- Only have to do when looking at rows touched between `[xmin, xmax)`
- Long running transactions can make this range big -> lots of lookups.
- If we stick long running transactions in a separate xip array we can tighten the range.
  - Less CSN lookups needed.
- By having a limited xid range where lookups are needed can use a ring buffer to store CSNs.
  - One indirection less.
  - Simpler to make lock free.
- Long running read transactions will also need conversion.

Shared memory structures

- L1 mapping: `CSN ringbuffer[N]`
- Has the following “clock hands”
  - nextXid - next slot to hand out
  - csnXmin - every running transaction before this is in L2
  - globalCSNXmin - every snapshot has higher csnxmin
- L2 mapping:

```
XidCSNPair longTx[]
```


Built on hope

- Hopefully most transactions will have committed when we have to move `csnXmin` hand past them.
- Hopefully most snapshots are released before we hit their csnXmin.

Built on hope

- Hopefully most transactions will have committed when we have to move `csnXmin` hand past them.
- Hopefully most snapshots are released before we hit their csnXmin.

A large enough ringbuffer will ensure hopes come true.

Common operations

- AssignTransactionId():
  - If there is room, bump nextXid
  - If not, make some.
- GetSnapshotData()
  - Read visible_csn, xmax, csn_xmin
  - Scan L2 for long running transactions. (could do this lazily?)
  - Publish csn_xmin
- CommitTransaction()
  - If we are still in L1, write CSN
  - If we are in L2, look up our entry, tag with CSN

Batch operation

- Every now and then move clock hands up.
  - Reduce contention on shared datastructures by this factor.
- Scan L1 from csnXmin for still running transactions, move them to L2.
  - Write Xact entries for the rest
- Scan procarray for new global CSNXmin.
- Signal all snapshot holders with old enough `csnXmin` to advance their CSN xmin.
- To advance snapshot `csnXmin`, scan L1.
- A few extra clock hands and careful coordination can make a lot of this lock free.

Visibility check

```
bool
XidVisibleToSnapshot(TransactionId xid, Snapshot snapshot)
{
    if (xid > xmax) return false;
    if (xid < xmin) return true;
    if (xid < snapshot->csn_xmin)
        return pg_lfind32(xid, snapshot->xip, snapshot->xcnt);

    csn = pg_atomic_read_u64(&ringBuffer[xid % RING_SIZE]);
    read_barrier();
    if (pg_atomic_read_u64(ringBufferCtl->globalCSNXmin) > xid)
        goto evicted;
    return csn <= snapshot->csn;
    evicted: // TBD
```


Existing work

Andrey Lepikhov, et. al

- Last state: 2021-11-19.
- Thread “Global Snapshots”.
- CSN is assigned during ProcArrayEndTransaction().
- CSN assignment is WAL logged.
- On replicas transaction visibility is postponed until visibility record arrives.

Heikki Linnakangas

- Based on “Global Snaphshots” patch.
- Heavily changed to only use CSN based snapshots on standbys.
- CSN = Commit LSN.
- Main goal is to get rid of KnownAssignedXids hackery.
- Reduced scope to get something more easily committable.
- Intention to get it into 18.

Recap

Where are we today

- Our current snapshot mechanism has some major issues when running across multiple machines.
- Some of those issues are implementation details exposed as bad semantics.
- CSN snapshots provide an easy to reason about way to fix those problems.
- CSN snapshots also make it easier to implement distributed transactions.
- Performance penalty/gain remains to be proven by an actual implementation.

Thanks!