# Towards ACID scalable PostgreSQL with partitioning and logical replication

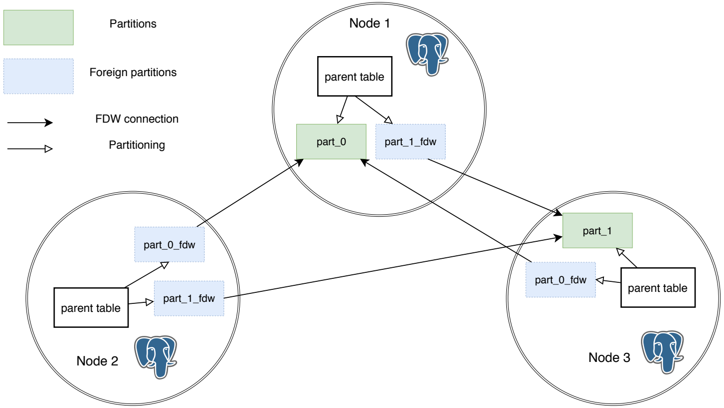
Arseny Sher / Anastasia Lubennikova

## The goal

- PostgreSQL with horizontally scalable
  - Performance – whether the bottleneck is CPU, RAM access or stable storage
  - Storage
- Mainly oriented on OLTP workloads
- Preserving ACID transactional semantics
- Providing fault tolerance, ideally – with automatic failover (HA)

## Scalability

Sharding gives scalability. The most straightforward way to get sharding in Postgres is to combine partitioning with postgres_fdw.



## Partitioning in PG 11 is cool

- Hash partitioning, allowing to change number of partitions;
- Partition pruning without checking constraint of every partition;
  - Well, currently UPDATE/DELETEs are pruned in range/list partitioning by exhaustive search, and for hash partitioning not pruned at all.
- Runtime partition pruning;
- Partition-wise joins;
- Foreign keys to partitioned tables;
- Automatic creation of parent indexes on partitions;
- Tuple routing and COPY for FDW (and postgres_fdw) partitions;
- postgres_fdw partition-wise aggregation;
- Something else?

pg_pathman extension has many of these features for PG 10 and 9.6.

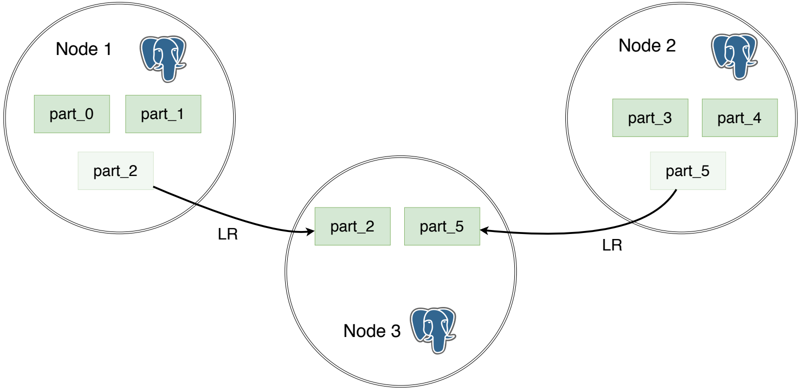
## How to shard

Main approaches are by hash and by range.

- Hash sharding practically always provides even load of partitions;
- With range sharding, it is easy to skew the load (e.g. if sharding key is timestamp and most queries work with latest data);
- Range scans over the sharding key are inefficient with hash sharding, unlike in range sharding

Currently pg_shardman supports only hash sharding.

## Rebalance and resharding



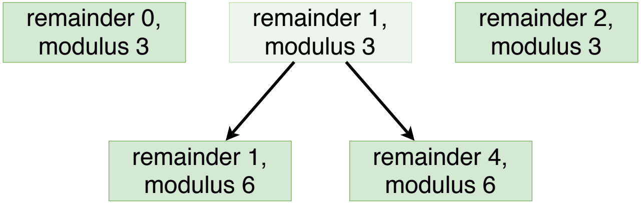
Rebalance without resharding; logical replication provides seamless migration.

Regardless of sharding strategy, resharding should be rare.

## Resharding by splitting a partition

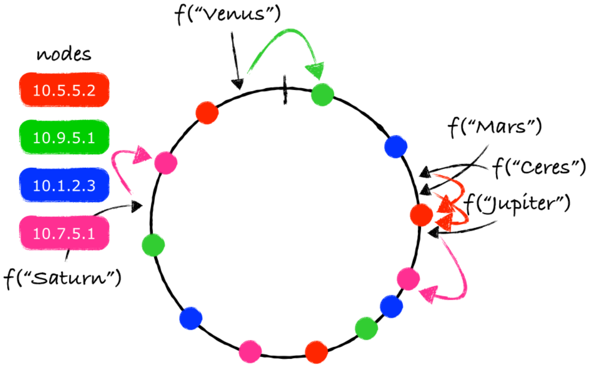
Splitting partition with range sharding is straightforward.

Hash-partitioning in 11 allows to elegantly split partition into two without touching the other ones.



# Resharding in hash case

Besides, we can implement well-known consistent hashing technique, allowing to add new partition taking a bit of data from existing ones, without reshuffling everything.



img source

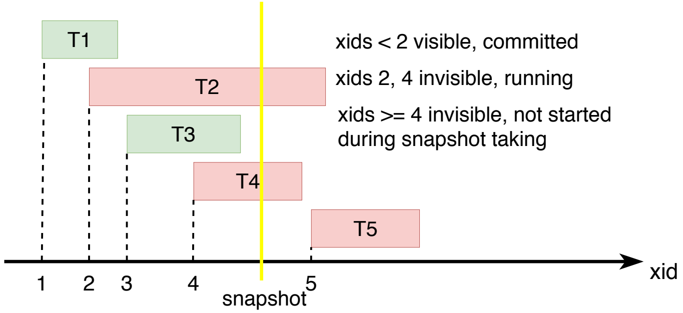
## Distributed transactions

Among ACID, we should care mostly about A (atomicity) and I (isolation).

- Durability is already handled by Postgres.
- C concerns business logic.

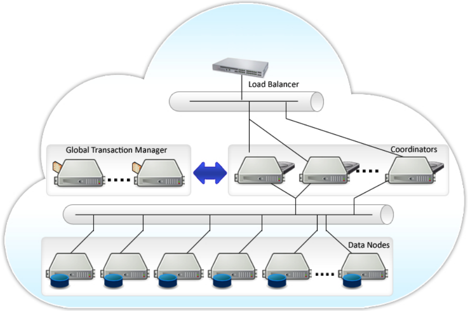
## Isolation

- Postgres employs MVCC for concurrency control. Transactions get snapshots determining which data they see; snapshot is basically a list of running transactions: these (and not yet started) xids are invisible, the rest are visible unless they were aborted.
- For that, array of running transactions is kept in shared memory (procarray).
- Transactions set their xid in procarray when they get it and remove xid when they are done.


T1
T2
T3
T4
T5
xids < 2 visible, committed
xids 2, 4 invisible, running
xids >= 4 invisible, not started
during snapshot taking
1
2
3
4
5
snapshot
xid

## GTM

Single global transaction manager assigning xids and snapshots is the most straightforward way to adapt this for the cluster. This is what e.g. Postgres-XL(C) projects family does.



This is fine for OLAP environment with low transaction rate, but on OLTP workloads GTM can be a bottleneck. We'd better find a decentralized solution...

img source

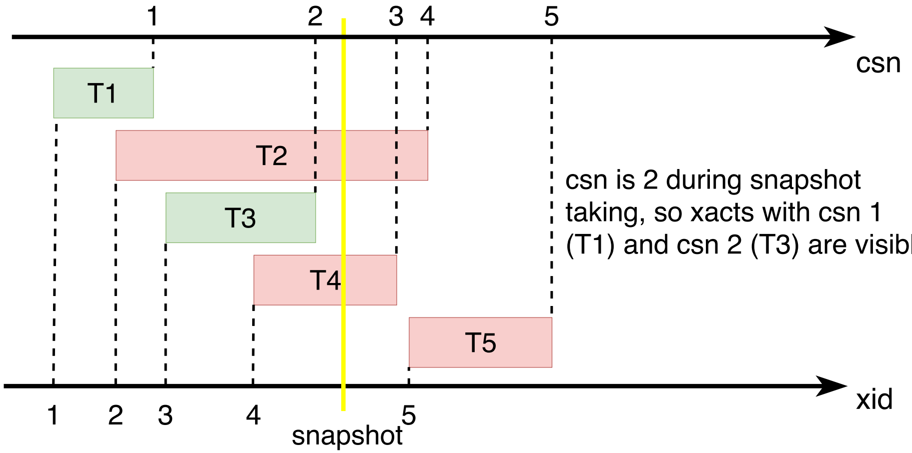
## Commit sequence number (CSN) MVCC approach

Currently, taking snapshot has $O(n)$ complexity where $n$ is number of backends: we need to copy array of running transactions.
CSN[1] is another MVCC implementation which avoids that. With CSN, there is an ever-increasing atomic CSN counter:

- To get snapshot, current value of the counter is remembered; it is essentially the snapshot.
- When xact finishes, it increments the counter and writes it to all tuples it had modified (in practice, xid -> csn map is used)
- Transaction with snapshot (csn value) $x$ sees actions of transaction with CSN $y$ if $x >= y$.

[1] There is a CSN patch into core PG with long history

# Commit sequence number (CSN) MVCC approach



csn is 2 during snapshot taking, so xacts with csn 1 (T1) and csn 2 (T3) are visible

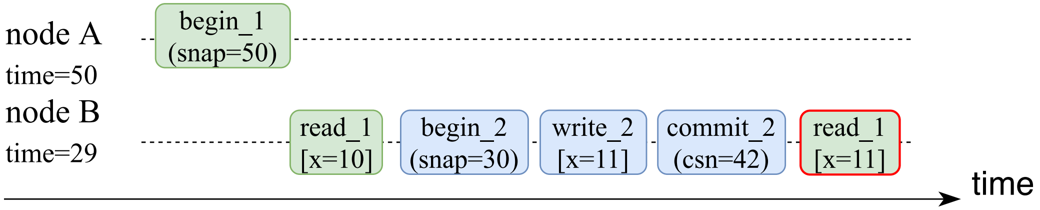
## Clock-SI: physical time as CSN[2]

In distributed environment, time can be used as CSN. This requires special care as time is not synced ideally.

Clock-SI prerequisite: time **never** goes back on a single node during its uptime – easy to implement and simplifies reasoning. However, time is allowed to go forward with arbitrary speed on different nodes.

[2] Stas Kelvich has posted the patch to -hackers

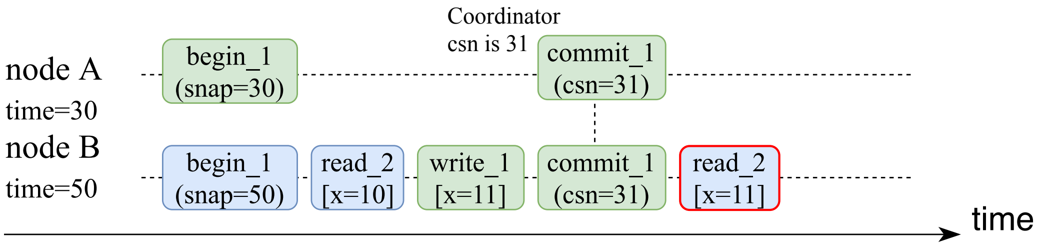
## What can go wrong if time is out of sync: case 1



Snapshot 50 of xact 1 is in the future from node B point of view, so it is not stable: $x$ changed its value.

- First rule of Clock-SI prevents that: transaction with snapshot from the future must wait until it becomes present in local time.
  - Since time is allowed to go with arbitrary speed, we actually can just pull the clocks hands forward instead of waiting =)

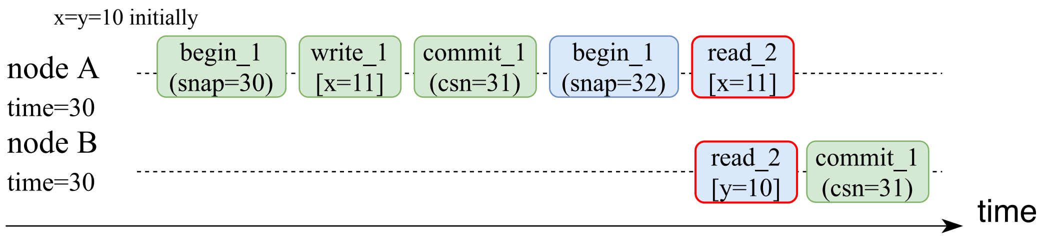
## What can go wrong if time is out of sync: case 2



Xact 1 committed with its coordinator CSN 31, modifying existing snapshot 50 of xact 2.

- Second rule of Clock-SI prevents that: CSN must be max of timestamps of all xact participants, not just coordinator's.

## Third problem: distributed xacts don't commit instantly everywhere



Xact 1 is already committed on A, but not yet committed on B when xact 2 comes into play: it sees updated *x*, but not updated *y*.

- Third rule of Clock-SI addresses this by making commit two-phased: xact is marked `InDoubt` on all nodes first and only then committed. Anyone trying to look at tuples modified by `InDoubt` xact must wait until its commit/abort.

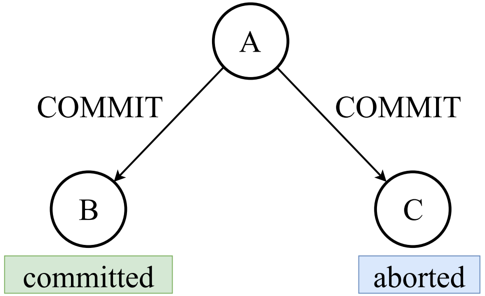
## We are still not fully covered yet

These 3 rules form Clock-SI; by following them, we get snapshot isolation semantics. Similar approach is used in CockroachDB and Google Spanner.

There is still one fundamental issue: recency guarantee. In PG, snapshot surely includes all xacts that were committed before snapshot creation. Here, snapshots generated on nodes with lagging clocks might not include latest xacts.

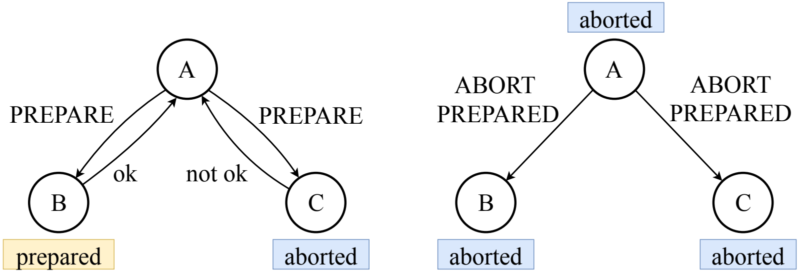
- Google Spanner relies on atomic clocks ensuring small skew: all xacts just wait the max skew ($\sim$ 7ms) after commit before ack'ing transaction to the client.
- CockroachDB relies on hard bound for clock skew too; however, always waiting is impractical since no special hardware is assumed and max skew can be 100-250ms. Instead, transaction is restarted if tuple's csn is in $[snapshot, snapshot + max\_skew]$ window.
    - Node commits suicide (hopefully, fast enough) if it discovers that max clock skew was exceeded.

## Atomicity



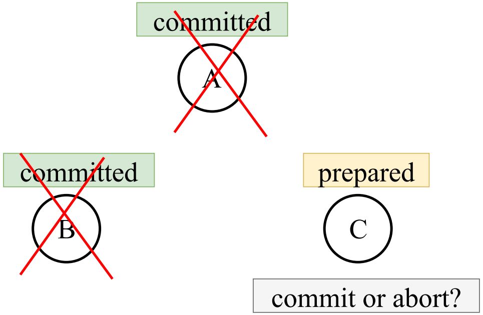
Without 2PC, transaction might end up committed on some nodes and aborted on others.

## Atomicity: 2PC



- Special state PREPARED is introduced, in which node is ready to commit transaction, but still can abort.
- We first PREPARE xact on all participants, and only then commit.
- PostgreSQL already provides 2PC infrastructure, pg_shardman makes use of it.

## Problem: 2PC is a blocking protocol



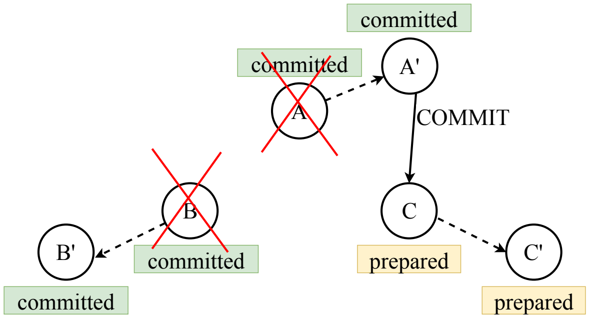
If coordinator has failed during commit, we won't be able to learn the status of transaction and resolve hanging PREPAREs on participants.
We can't abort xact on node C (probably it was committed on A and B?) and can't commit it (probably it is was aborted on A and B?).

## Problem: 2PC is a blocking protocol

- There are more complex protocols like Paxos Commit. The general idea is to persist each node PREPARE decision on majority of nodes before committing anything. If we implement it, we will be able to resolve hanged PREPAREs even if coordinator is down.
- But... failed coordinator holds data too, so we need to provide redundancy and failover anyway.

## Problem: 2PC is a blocking protocol

And if we provide redundancy and failover, we can persist on replicas decision about transaction fate as well.

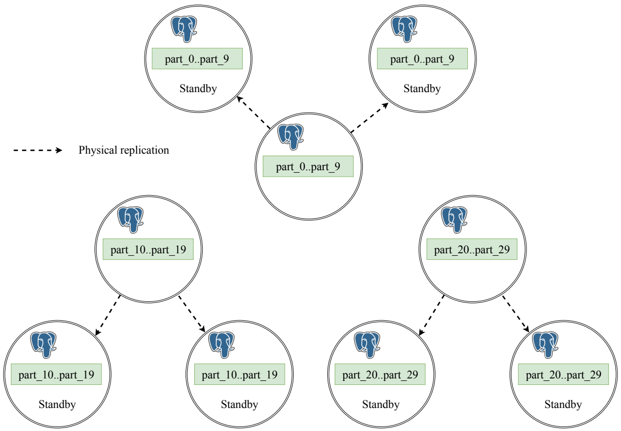


## Fault tolerance

- Again, it makes sense to reuse existing infrastructure. Postgres offers physical and logical replicaion.
- One immediate issue with physical replication is instance placement.

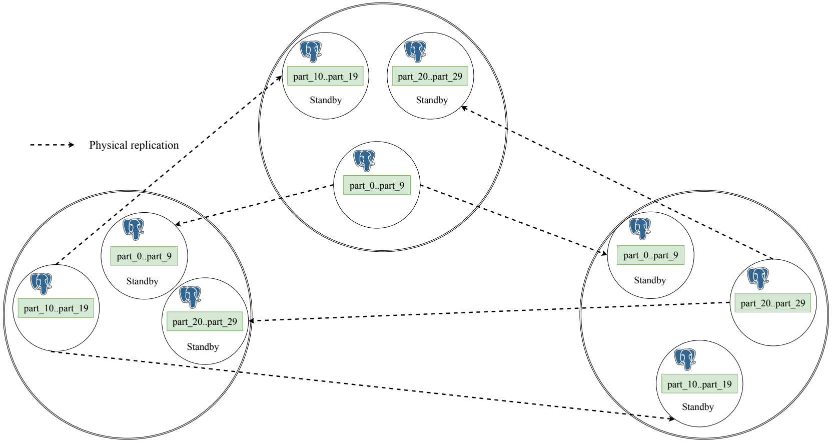
## Physical replication

Either we need `redundancy` times more nodes, most of them staying idle...

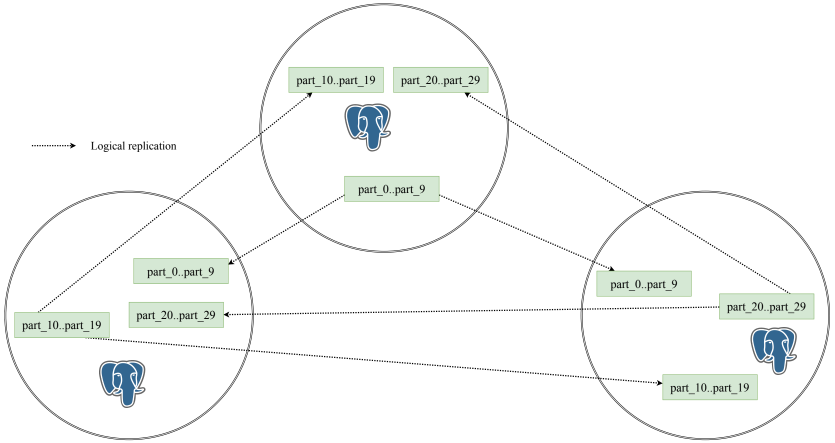


## Physical replication

Or we need to place multiple PG instances on each physical node, which is ugly.



## Logical replication



- pg_shardman currently provides this, with manual failover

## Fault tolerance and HA

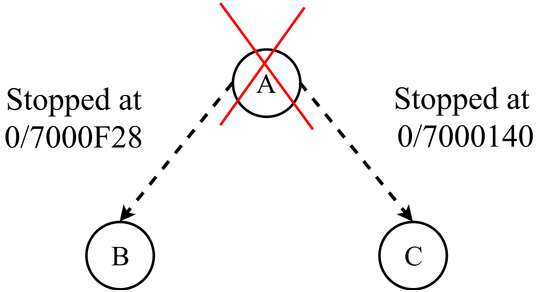
Regardless of replication method, there are two important milestones:

- Manual failover: DBA says 'this node has failed', and replicas are promoted.
- Automatic failover, when cluster does this on its own.
  - In practice, this provides HA, because failover is very fast. This allows to bypass CAP theorem: in CAP, availability means 'live node must successfully answer to any query'.

Generally, failover involves

- Choosing replica which will be the new master
- Configuring replication from it to the rest of replicas
- Announcing it as a new master to all nodes

## Problem: replicas might be in different states



- Easy to handle with physical replication, LSNs are the same
- For logical replication, we need to learn since which LSN to start, by looking at mappings to old master's LSN.
  - pg_shardman can synchronize replicas, but does it in much more hackish way.

## Getting the failed node back

- To get failed node back to the cluster without copying all the data from scratch, we must be able to *rewind* latest changes which hadn't gotten to the next master.
- For physical replication there is `pg_rewind` utility doing exactly that. For logical replication... this is yet to be done.

## 2PC and logical replication

Logical replication also complicates 2PC.

- PREPARE must be decoded [3]
- Initial tablesync needs some care

With physical replication, 2PC just works across replicas.

[3] patch in progress

## Automatic failover

- Whenever a bunch of nodes need to reach an agreement on something, there is a distributed consensus problem solved by algorithms like Paxos.
- More practical algorithms implement replicated state machines approach, giving fault tolerant distributed log: Raft, Multi-Paxos. Servers apply this log, computing the same state.
- In Postgres, we already have the log with commands changing the state: WAL. We can adapt Raft to it and get automatic failover.
  - CockroachDB, Google Spanner, MongoDB do something very similar.

## Automatic failover

As an alternative, instead of building consensus algorithm right into replication code we can make use of external consensus.

- We need a fault-tolerant store supporting CAS operations on it – systems like etcd/consul/zookeper provide it. Internally, they still run consensus algorithm between their nodes. Current cluster conf is stored there and atomically updated.
    - It makes easy to store only two copies of data, quorum of data nodes is not needed.
    - Easy to handle membership changes.
- Projects Stolon and Patroni implement HA for physical replication in this way.

## Some more issues with logical replication

- No support for DDL (ok, just need to add it).
- Double latency: transaction is not sent to replica until COMMIT is written into WAL.
- Big transactions are spilled to (and read back from) disk, halving disk throughput even for one decoder.
  - Probably the best way to address these two issues is to teach apply worker to switch between transactions and send/apply them on the fly, much like in recovery. Greetings to connection pooling and autonomous transactions.

## More logical replication performance issues

- Updates/deletes amplification.
  - With logical replication, updates are executed on replicas independently. Indexes must be traversed for locating tuples and updating indexes themselves on both master and each replica; this is especially expensive if indexes don't fit into RAM. With physical replication, we just know which pages to update.
- Decoding itself eats CPU.
- Each walsender decodes the whole WAL: decoding results are not reused between walsenders, and uninteresting WAL still needs to be digested. Performance drops significantly with > 5-10 walsenders.
  - Probably we could make backends point to walsenders, which parts of WAL they need to read...
  - This limits the ability to balance the load after failure: less replicas => more work would fall on each one

## Replication summary

With logical replication

- Synchronization after failure and rewind requires some sweating;
- There is no DDL yet;
- 2PC needs special care;
- There are serious performance issues;

Physical replication

- Leads to multiple instances placement on each physical node (probably main problem here is just ugliness);
- Failover without external daemons is impossible because someone needs to run `pg_basebackup` and `pg_rewind`.

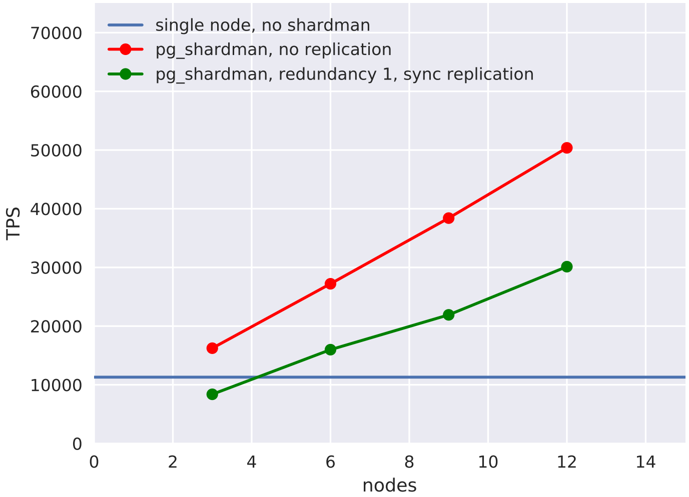
## pg_shardman currently: available to play around

- Hash-sharding with native partitioning (slow DML as partitions are not pruned) or pg_pathman (old version).
- Clock-SI providing distributed snapshot isolation.
- 2PC providing distributed atomicity.
- Redundancy via logical replication with manual failover.

https://github.com/postgrespro/pg_shardman

## Some numbers

pgbench -N on ec2 c3.2xlarge (8 cores) nodes; scale 10, data in tmpfs, optimal number of clients



## Random performance notes

- Scaling after around 100 nodes will need squats with connection pooling, because postgres_fdw doesn't reuse connections between backends: totally we have num of clients * num of nodes connections on each node.
- While OLTP is mostly fine, OLAP will immediately choke because postgres_fdw doesn't support parallel execution yet. Community works hard on this.
- Probably next distant goal for OLAP is reshuffling for efficient joins.
- postgres_fdw currently uses `WaitEvenSet` API very inefficiently, I'll post a patch.