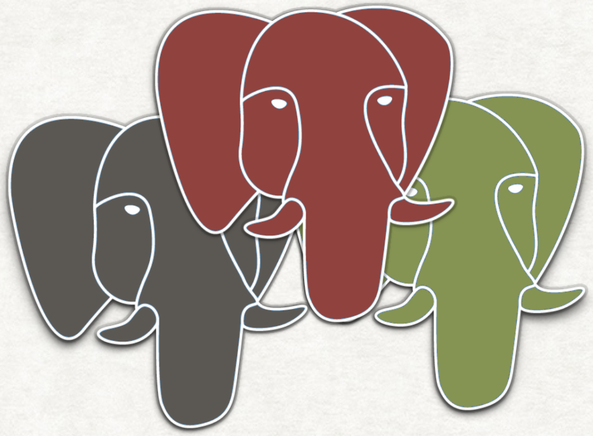
Of the building of a PostgreSQL cluster



Srihari Sriraman
nilenso

Stories

Each story shall cover

- Problem
- Quick fix
- Correct fix
- Root cause
- Lessons learnt

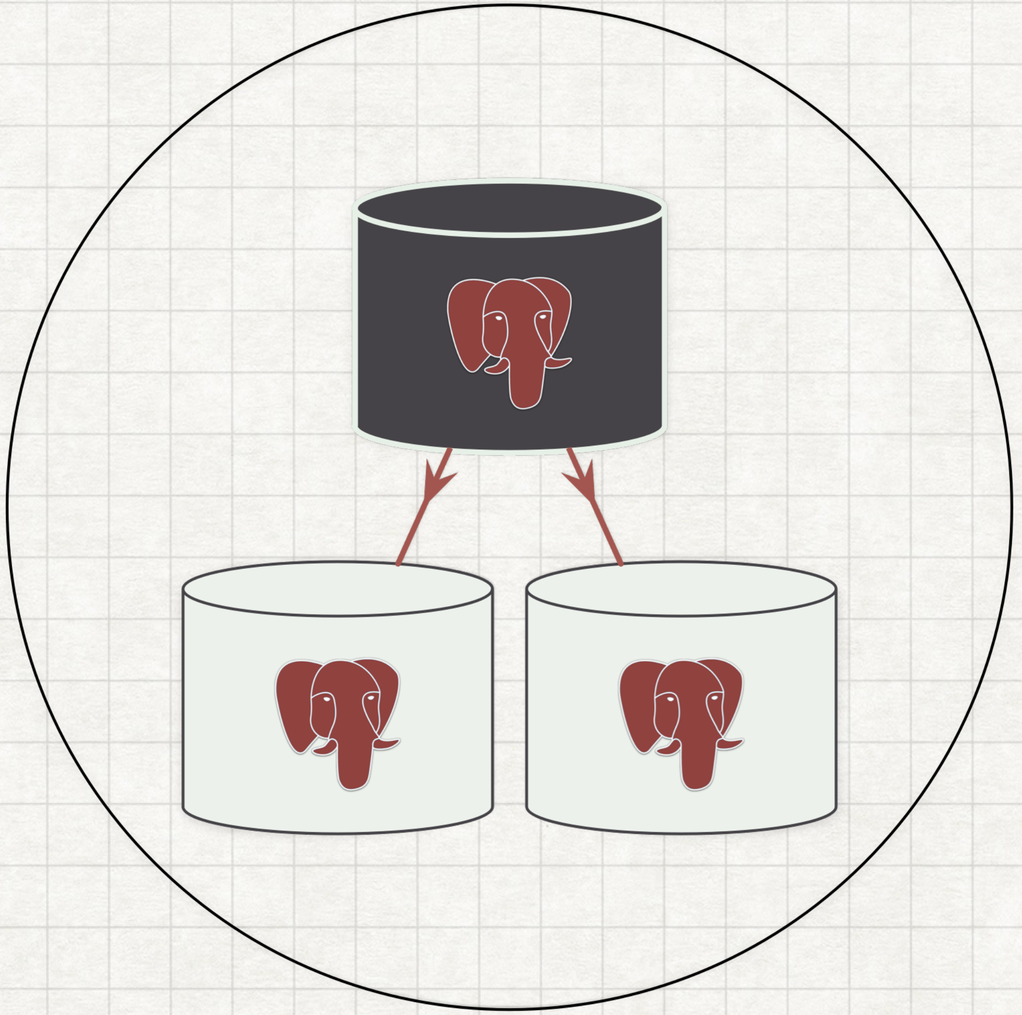
How impressive are thy numbers?

SLA	99.9% < 10ms
RPS	500
QPS	1.5k
TPS	4.5k
Daily Size Increase	8G
DB Size (OLTP)	600G
Biggest Table Size	104G
Biggest Index Size	112G
# Rows in biggest table	1.2 Billion

Machines	4 x i2.2xlarge
Memory	64G
Cores	8
Storage	1.5TB SSDs
Cloud Provider	AWS

STORY #1

We need a PostgreSQL cluster



We need reports on live data. We should probably use a **read replica** for running them.

Hmm, RDS doesn't support read replicas for PostgreSQL yet. But PostgreSQL has **synchronous replication** built in, we should be able to use that.

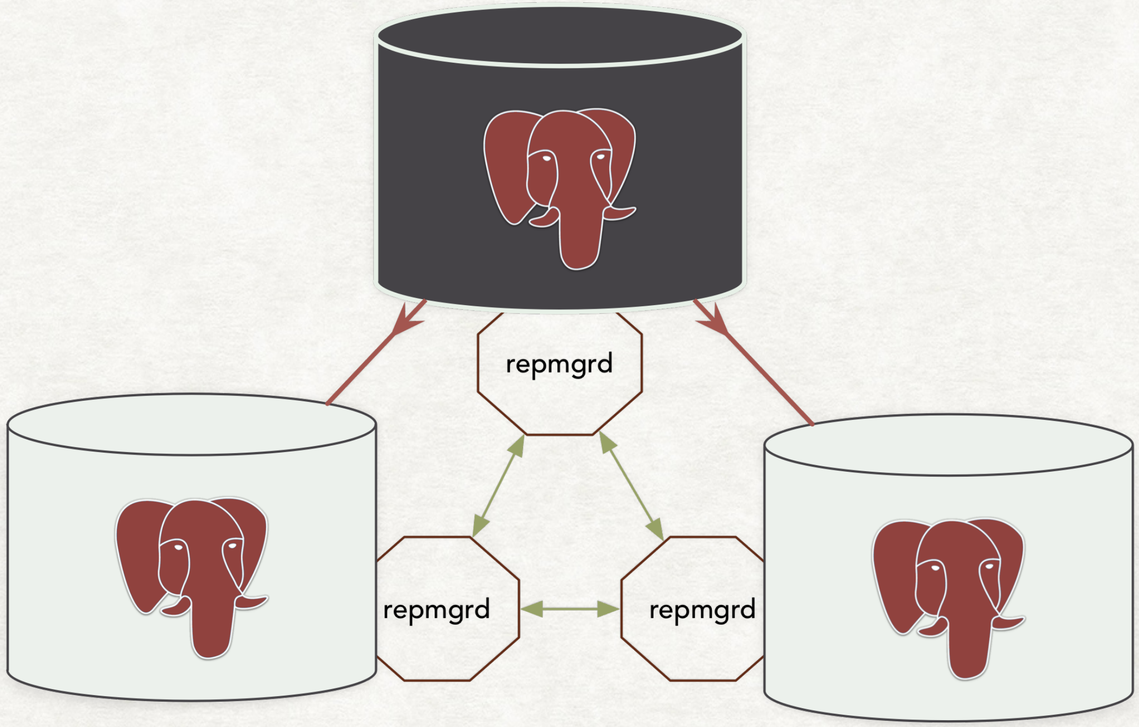
Also, we can’t be down for more than 5 seconds.

So not only do we need read replicas, we also need automatic failovers. What tools can I use to do this?

Tools, tools

Pgpool-II	It does connection pooling, replication management, load balancing, and request queuing	That seems like overkill for us.
Bucardo	It does multi-master, and asynchronous replication	We need synchronous replication, and we don’t need multi-master.
Repmgr	It does replication management, and automatic failover	Looks like what we want. Plus, it’s written by 2nd Quadrant.

The Repmgr setup



There is passwordless SSH access between all machines, and repmgrd runs on each of them, enabling automatic failovers.

The Repmgr setup

```
| id | type    | upstream_node_id | cluster | name              | priority | active |
|----+---------+------------------+---------+-------------------+----------+--------|
|  1 | master  |                  | prod    | api-1-prod        |      102 | t      |
|  2 | standby |                1 | prod    | api-2-prod        |      101 | f      |
|  3 | standby |                1 | prod    | reporting-0-prod  |       -1 | f      |
```

Repmgr maintains its own small database and table with the nodes and information on the replication state between them.

A few repmgr commands

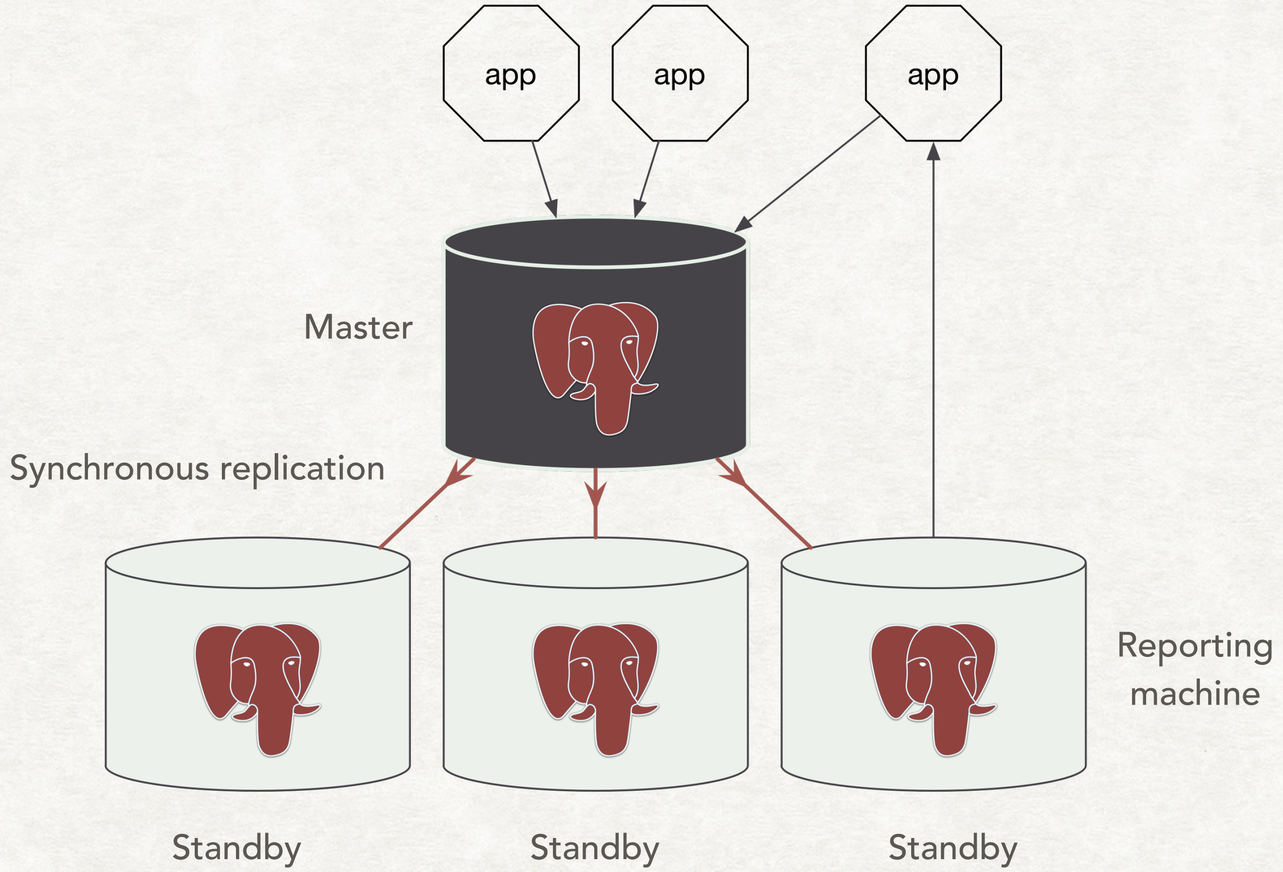
```
$ repmgr -f /apps/repmgr.conf standby register
```

```
master register
standby register

standby clone
standby promote
standby switchover
standby follow

cluster show
```

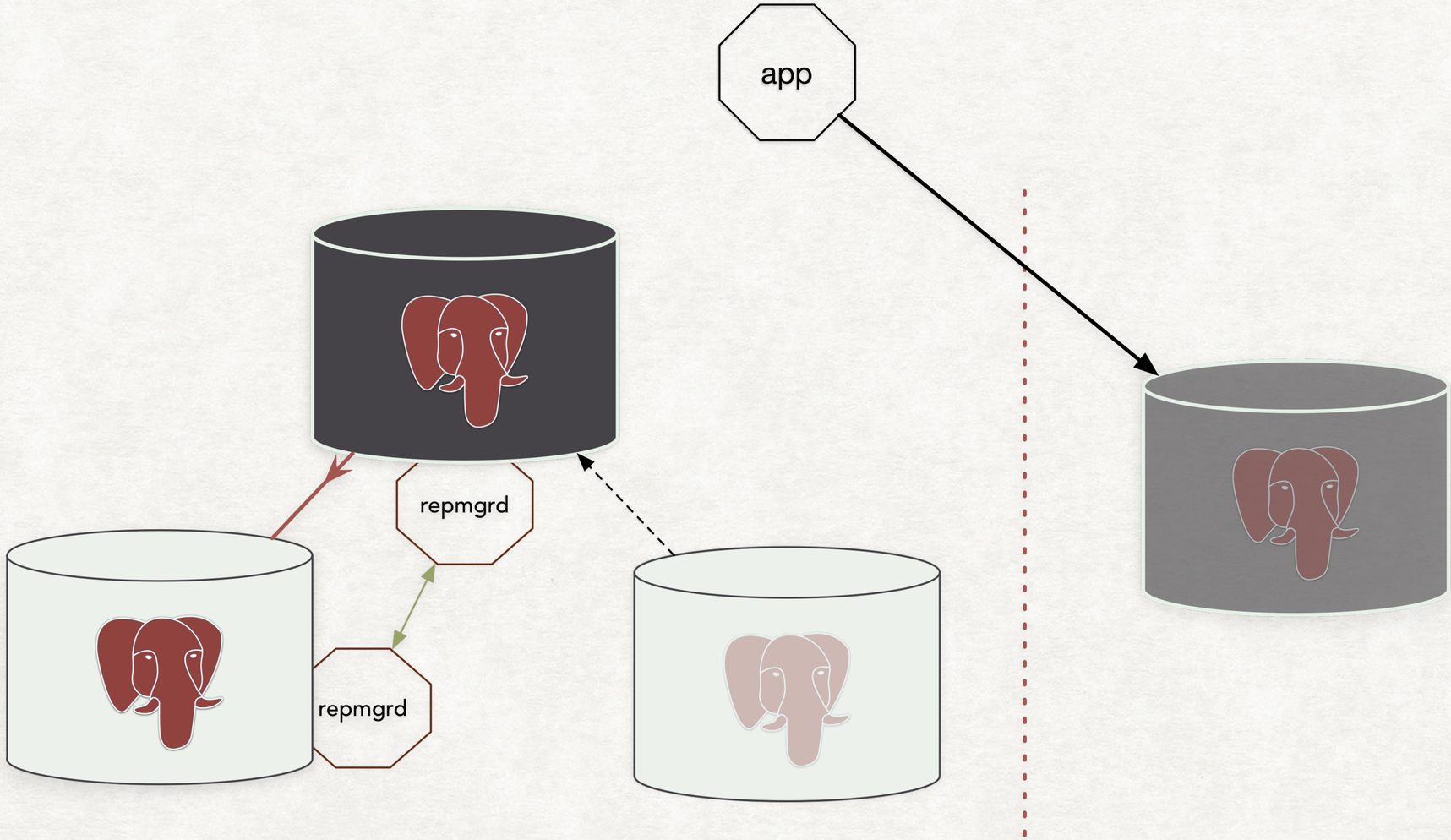
A brief look at the full setup



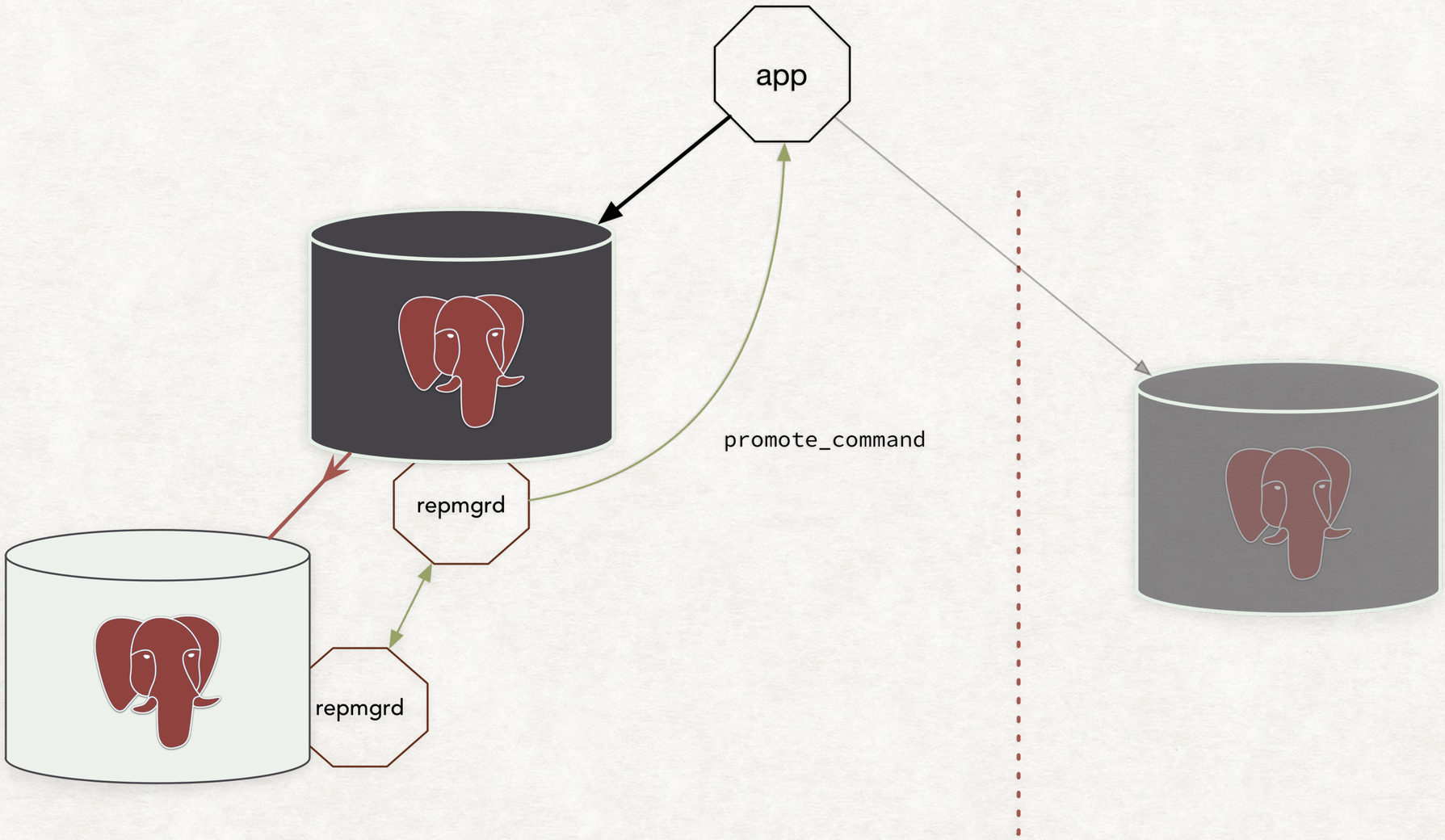
We have two standbys that we can failover to, and a reporting machine.
The applications write to the master, and can read from the standbys.

Oh! Repmgr doesn’t handle the communication between the application and the DB cluster. How do we know when a failover happens?

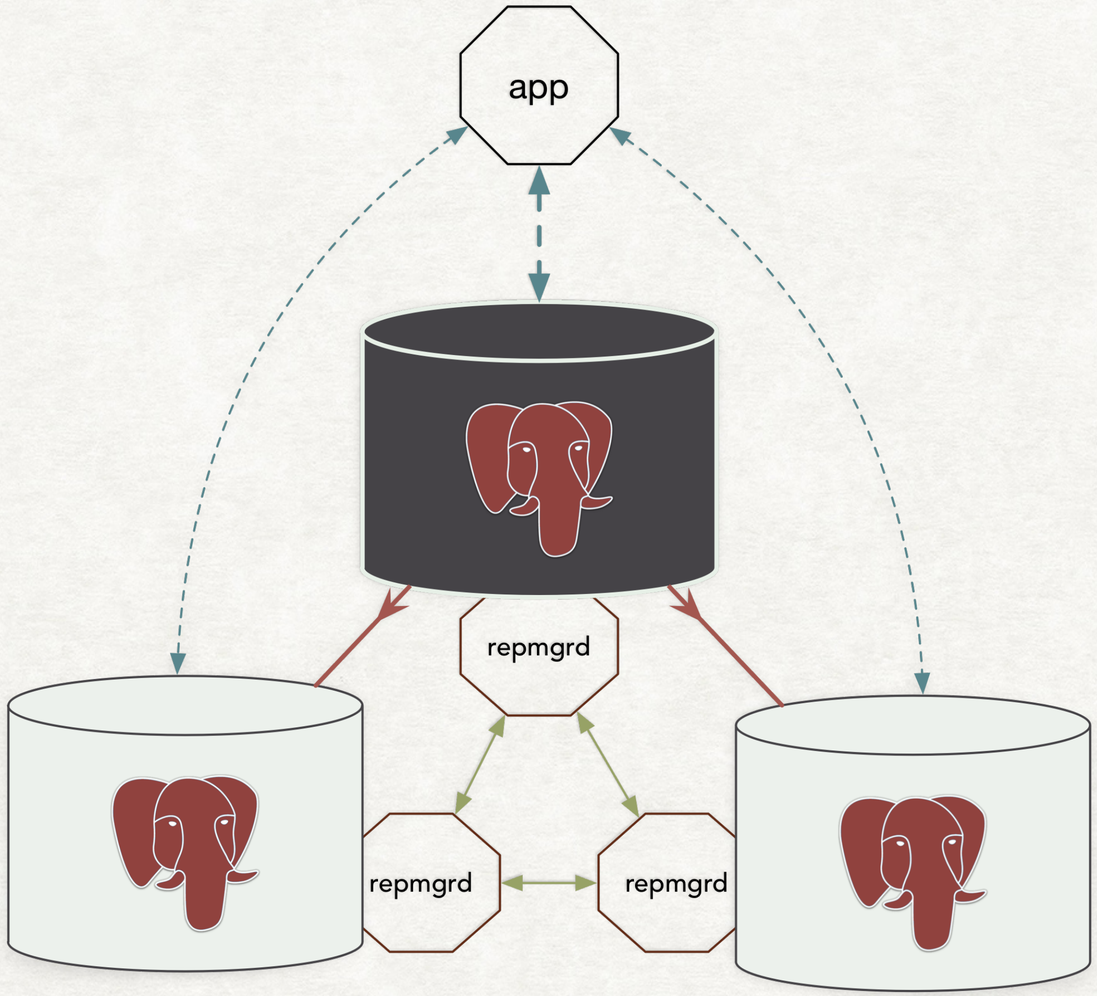
Let’s get them to talk to each other.



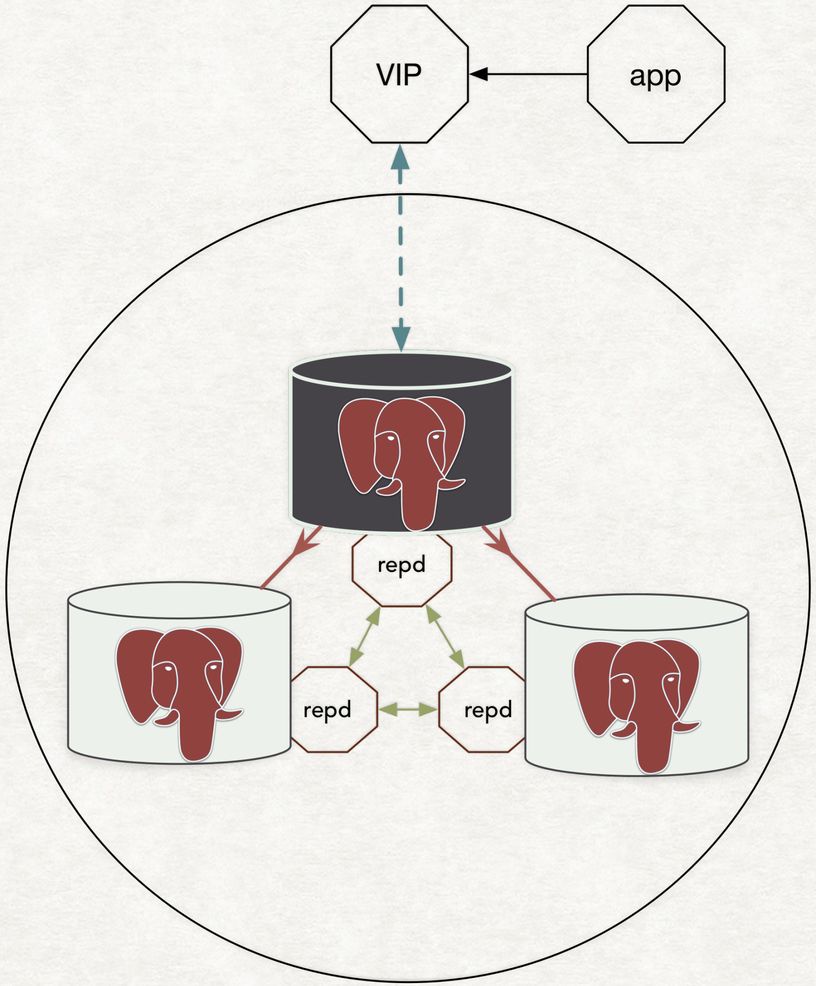
A failover is triggered when the master is inaccessible by any other node in the cluster. A standby is then promoted as the new master.



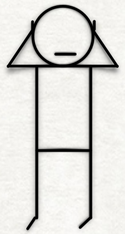
When a failover happens, the new master makes an API call, telling the application about the new cluster configuration, and the app fails over to it.



As a second line of defence, the application also polls the status of the cluster for any changes in the standbys' statuses.



A more sensible approach here is to use a Virtual IP, and use a retry mechanism within the application to handle failovers.



Oh no! AWS says a machine is unreachable, and it's our master DB!

I'm unable to ssh into the machine, even.

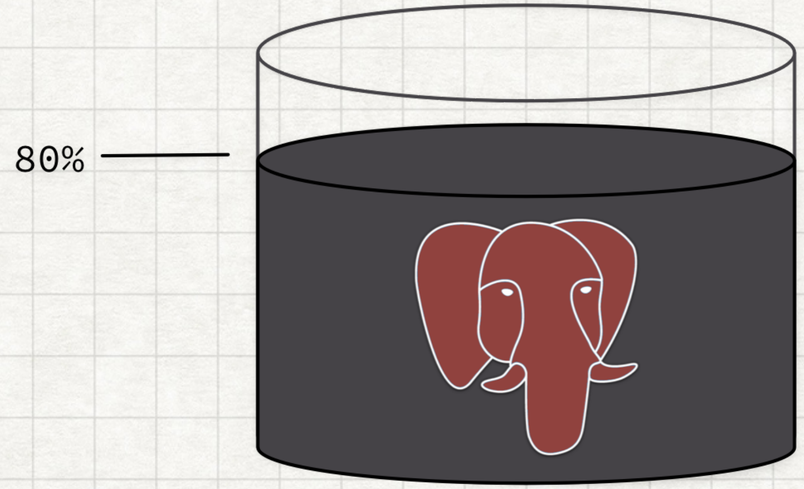
Ha. But a failover happened, and we're talking to the new master DB. We're all good.

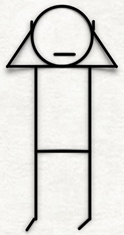
What have we learnt?

- Use Repmgr, it works.
- Repmgr does not communicate failovers to the application. We need to handle this ourselves.
- Use priorities to disable failing over to the reporting DB.
- Keeping the application aware of the cluster configuration could be messy.
- Use a VIP mechanism (ucarp) to instrument failovers.
- AWS might drop your box.
- Test failovers rigorously.

STORY #2

The disk is full



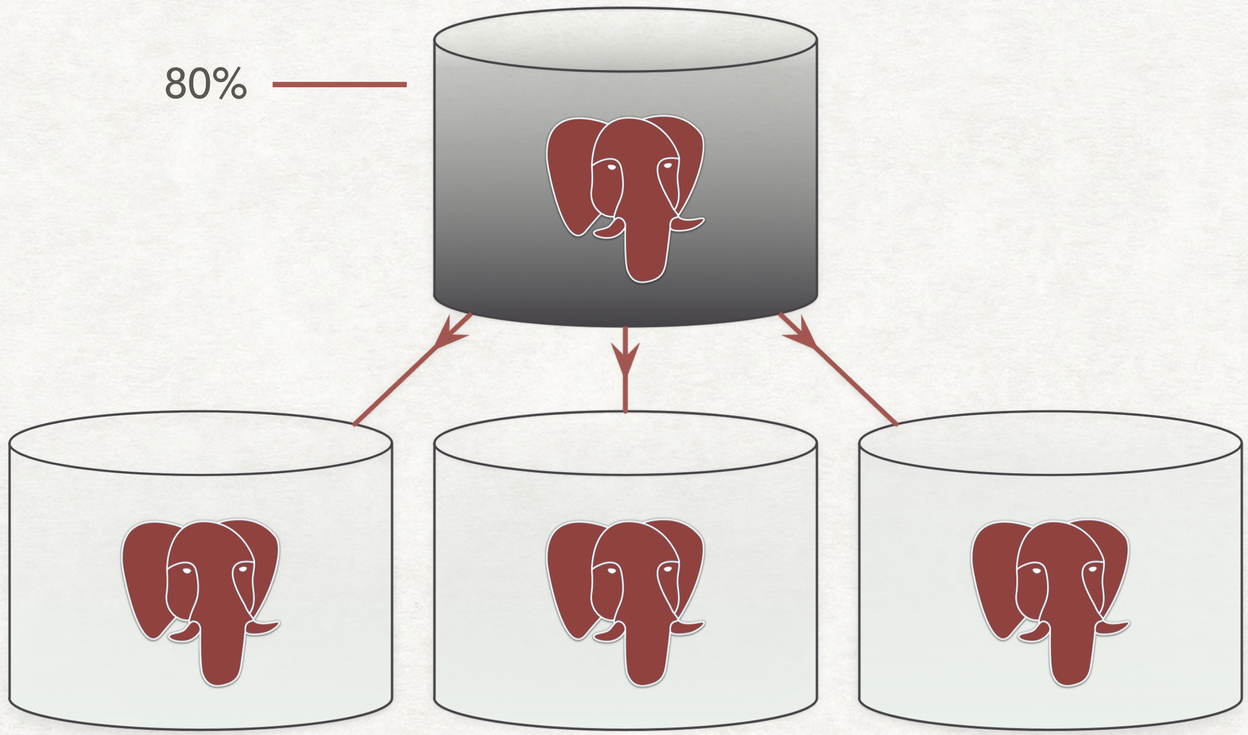


Oh nay! We art v'ry sloweth, and failing SLA.
We can loseth some data, but the s'rvice needeth to beest up. plz2fix.

The disk usage is at 80%, and the DB is crawling!

The disk usage wast only 72% last night of all, and i wast running the deletion script. How didst t wend up to 80% ov’rnight?

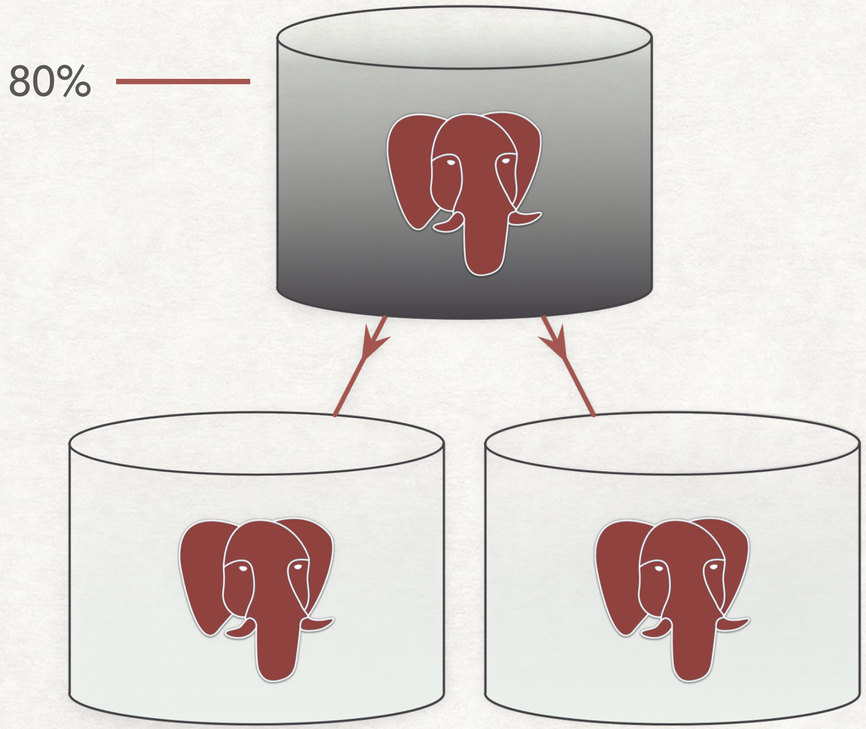
I needeth to fixeth the issue bef're debugging.

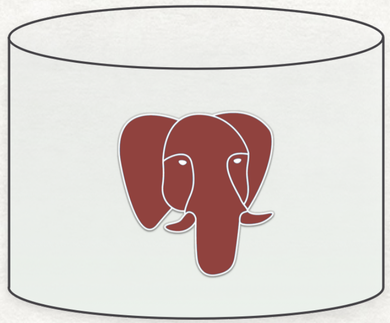


All the DB machines are at about 80% disk utilisation. We need to **truncate** a table to reclaim space immediately.

Restarting a standby as a standalone instance

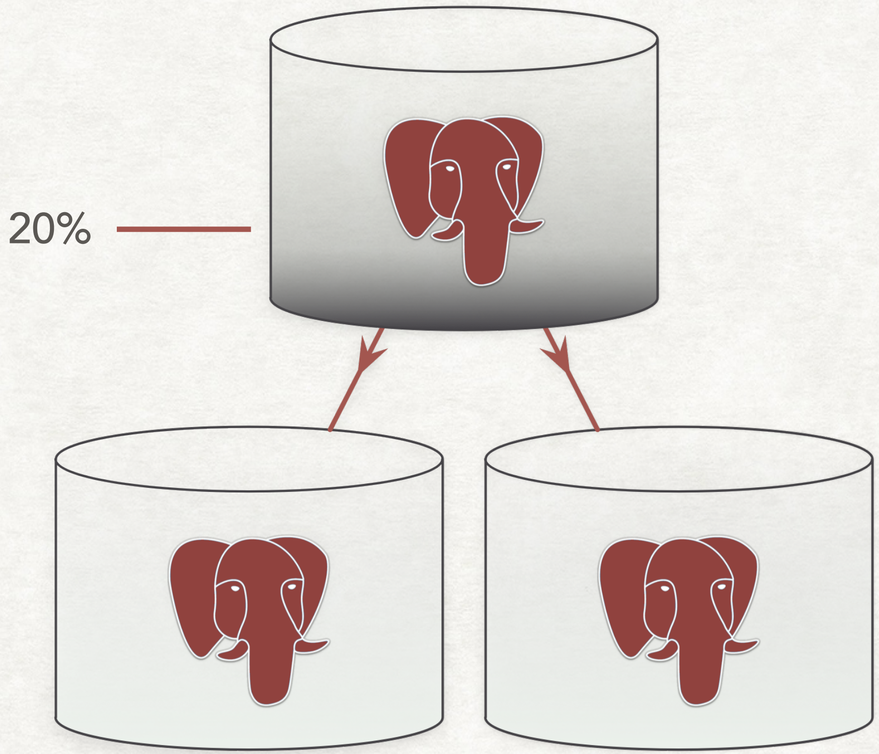
```
$ service app-db stop
$ repmgr -f /apps/repmgr.conf standby unregister
$ rm /db/dir/recovery.conf
$ service app-db start
```

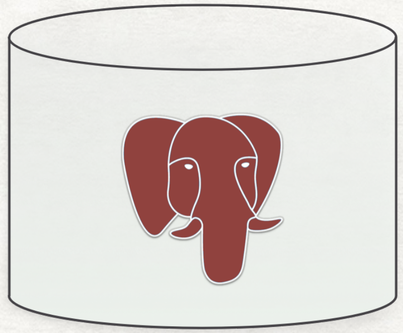




Standby node is out of the cluster

Once we have taken a standby out of the cluster, the data within it would be safe from any changes to master.





Deleted data is safe here, in a standalone instance

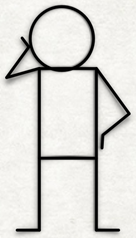
Now we can pull reports from the standalone instance while the application serves requests in time.

We thought we'd fail at 90%, and that wasn't going to happen for another week at least. So, why did we fail at 80%?

Oh, it's ZFS! ZFS best practices say *"Keep pool space under 80% utilization to maintain pool performance"*. **Never go over 80% on ZFS.**

Okay, but how did we fill the disk up by almost 10% overnight? We were only **deleting** from PostgreSQL.

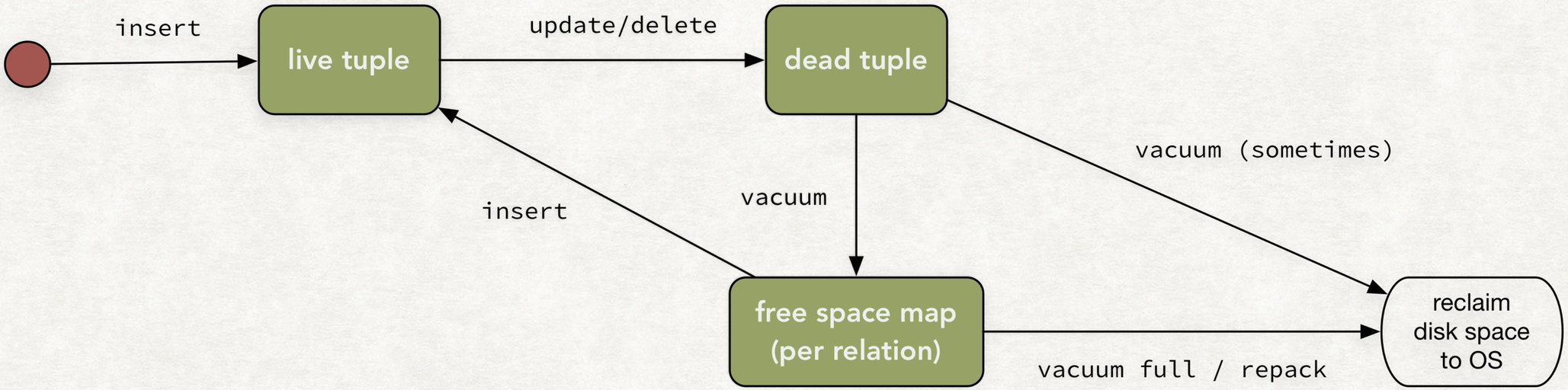
D'oh! Of course, PostgreSQL implements **MVCC**.



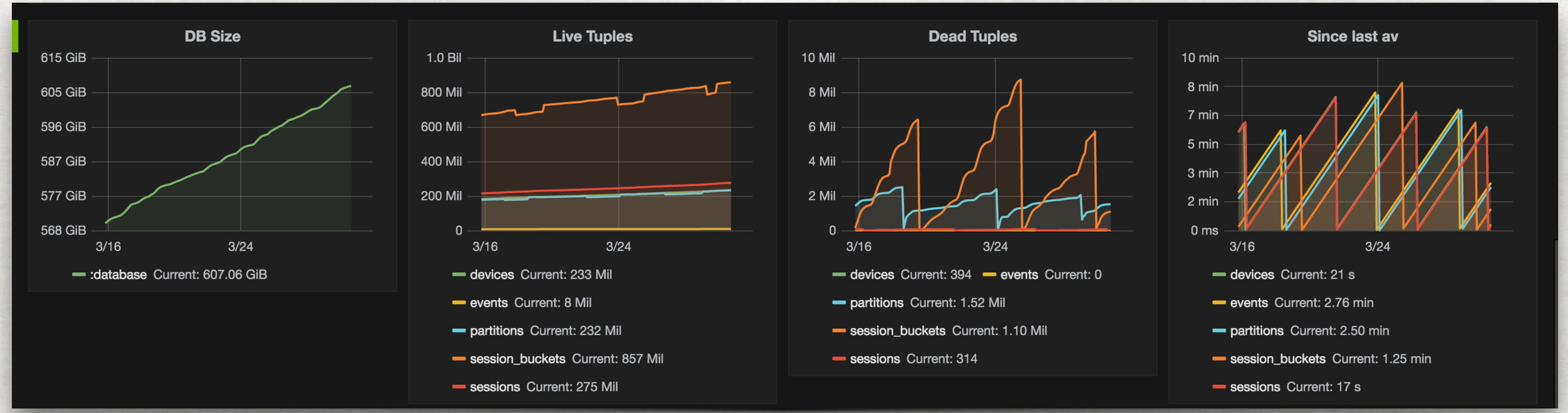
PostgreSQL MVCC implies

- `DELETE`s just mark the row *invisible* to future transactions.
- `UPDATE`s mark the old version invisible, and a new row is `INSERT`ed.
- `AUTOVACUUM` "removes" the invisible rows over time.
- The rows vacuumed are available as free space per relation to PostgreSQL.
- Actual disk space is not reclaimed to the OS during routine operations **.
- So, `DELETE`s will consume **more** space until the dead tuples are `AUTOVACUUM`ed.
- The default `AUTOVACUUM` worker configurations are ineffective for big tables. We have no choice but to make them **far more aggressive**.

A rough state diagram for Vacuum



A snapshot of the monitoring of vacuum

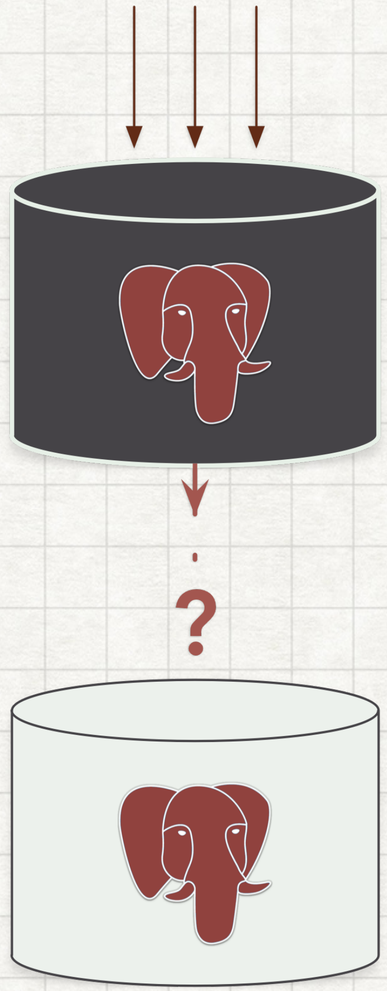


What have we learnt?

- Use standbys to have a **live backup** in case you need to flush data
- **80% is critical** for disk usage (on ZFS, but even generally for a database).
- Bulk deletion will cause immediate **increase** in disk usage, not decrease.
- Tune autovacuum workers aggressively, but carefully.
- Things to monitor: Disk usage, dead_tups, last_autovacuum

STORY #3

Unable to add a standby



Let’s get a better reporting box.

We have way more data now and more report queries too. So we need more IOPS, and more cores.

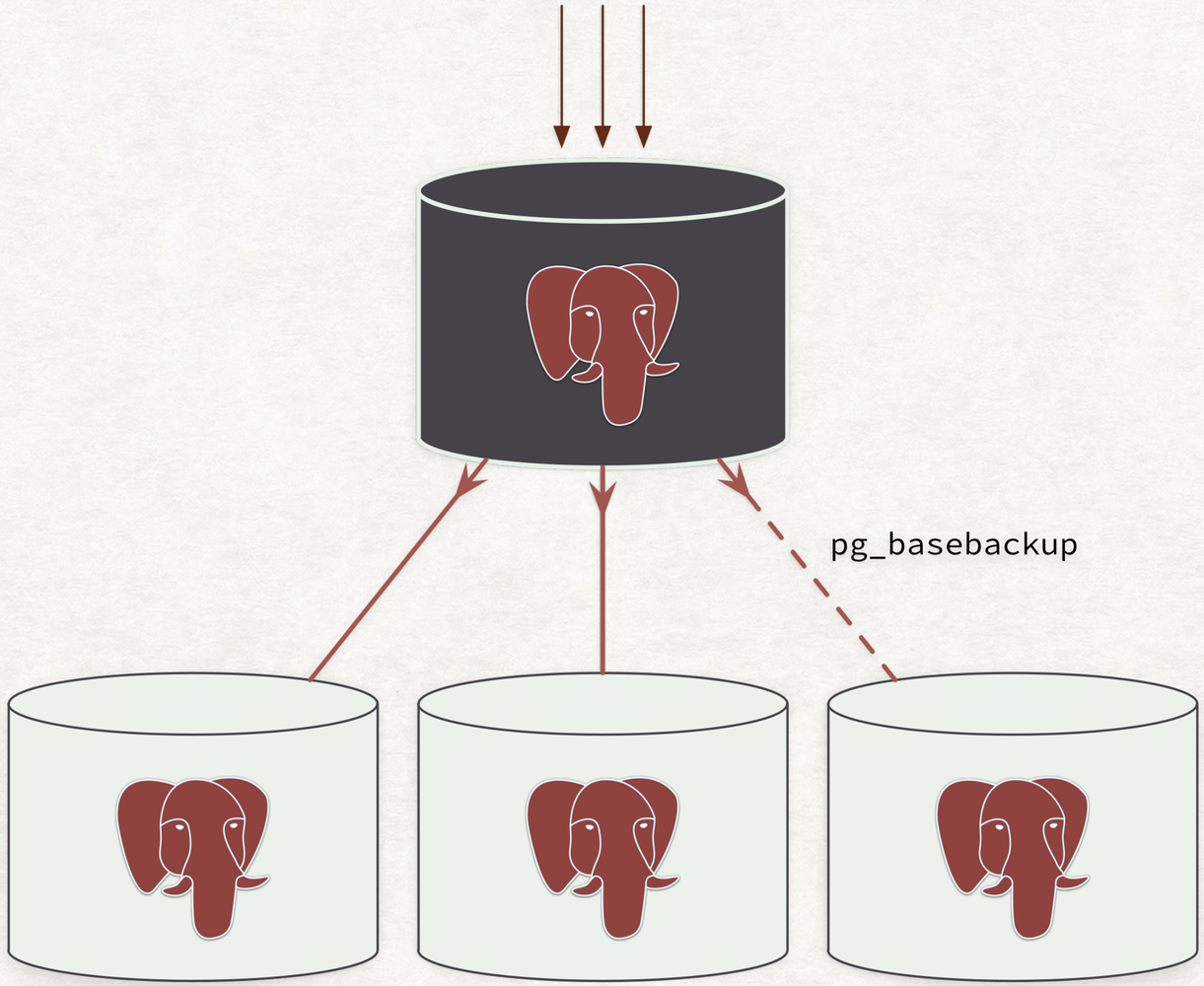
Easy Peasy! I’ll start the clone tonight.

```
LOG:  started streaming WAL from primary at
                        4038/29000000 on timeline 2
```

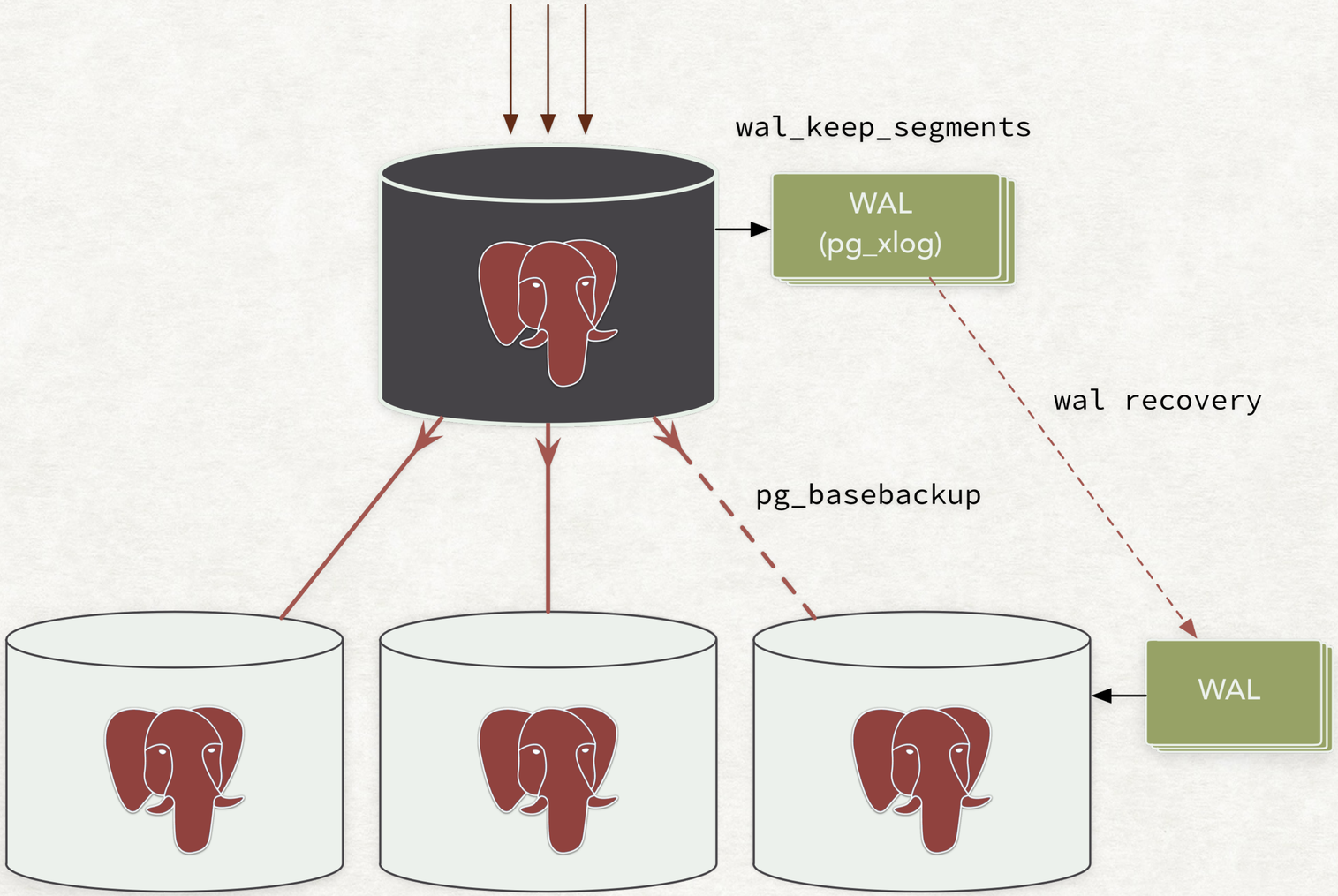
There, it’s transferring the data at 40M/s. We should have our shiny new box up and running in the morning.

```
> FATAL:  could not receive data from WAL stream: ERROR:  requested WAL segment 000000020000403800000029 has already been removed
```

Oh no. I had to run reports on this machine today!



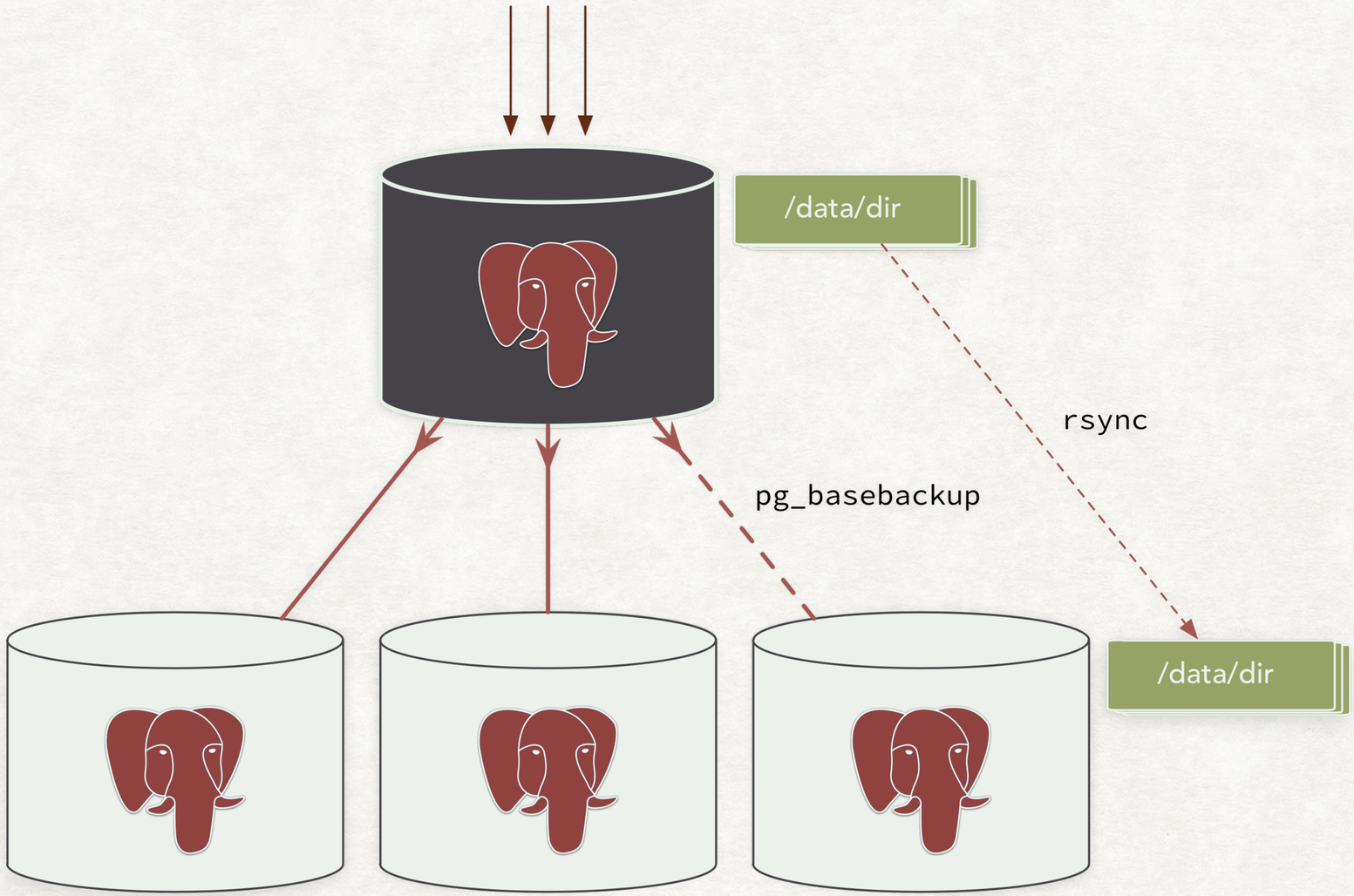
During pg_basebackup, the master DB continues to receive data, but this data is not transmitted until after.



WAL data is streamed later from the WAL on the master. If any WAL is discarded before the standby reads it, the recovery fails.

Hmm, it looks like we generated more than 8G last night while the clone was happening, and `wal_keep_segments` wasn't high enough.

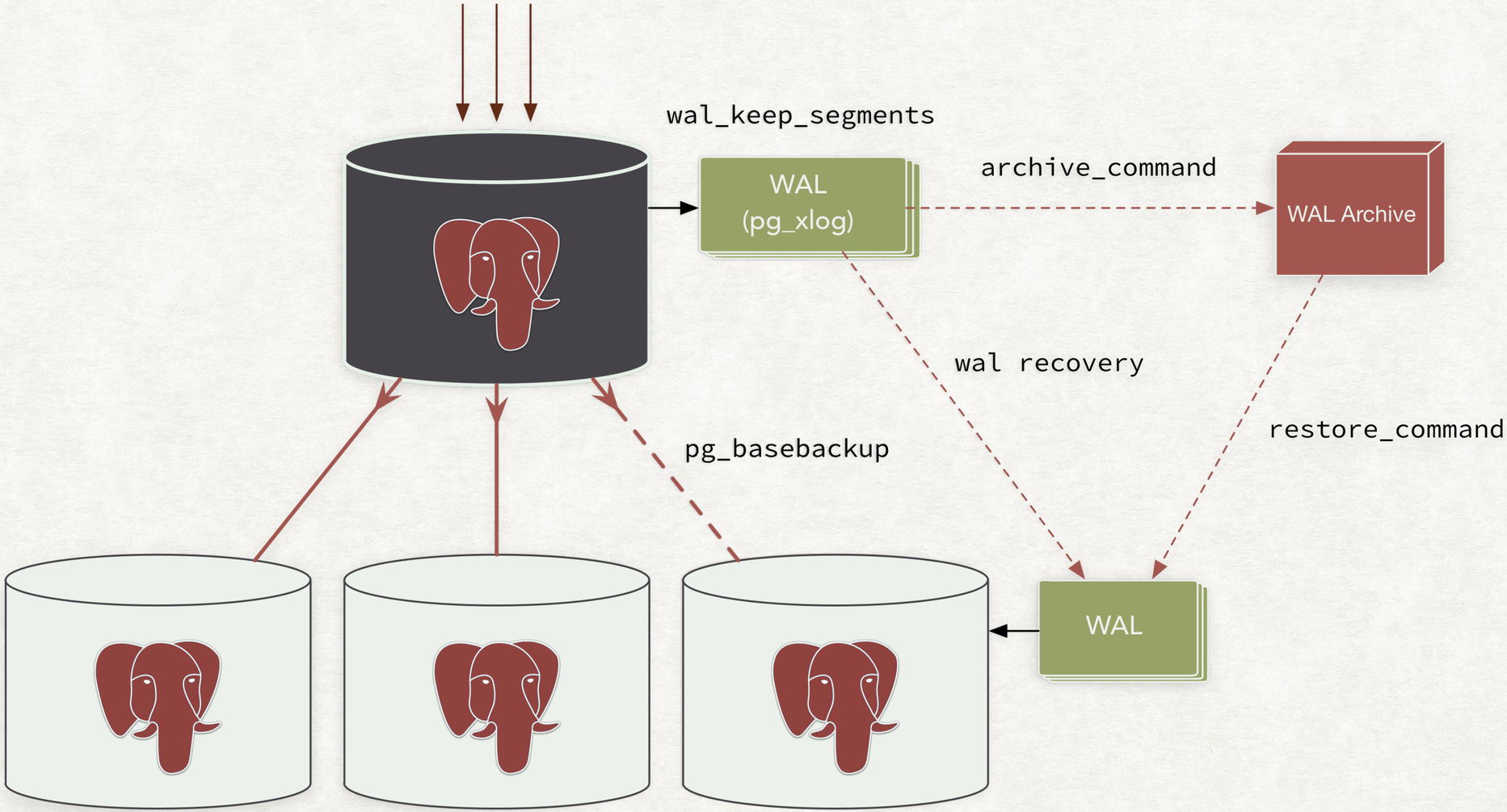
I think the `--rsync-only` option in Repmgr should help here, since I have most of the data on this machine already.



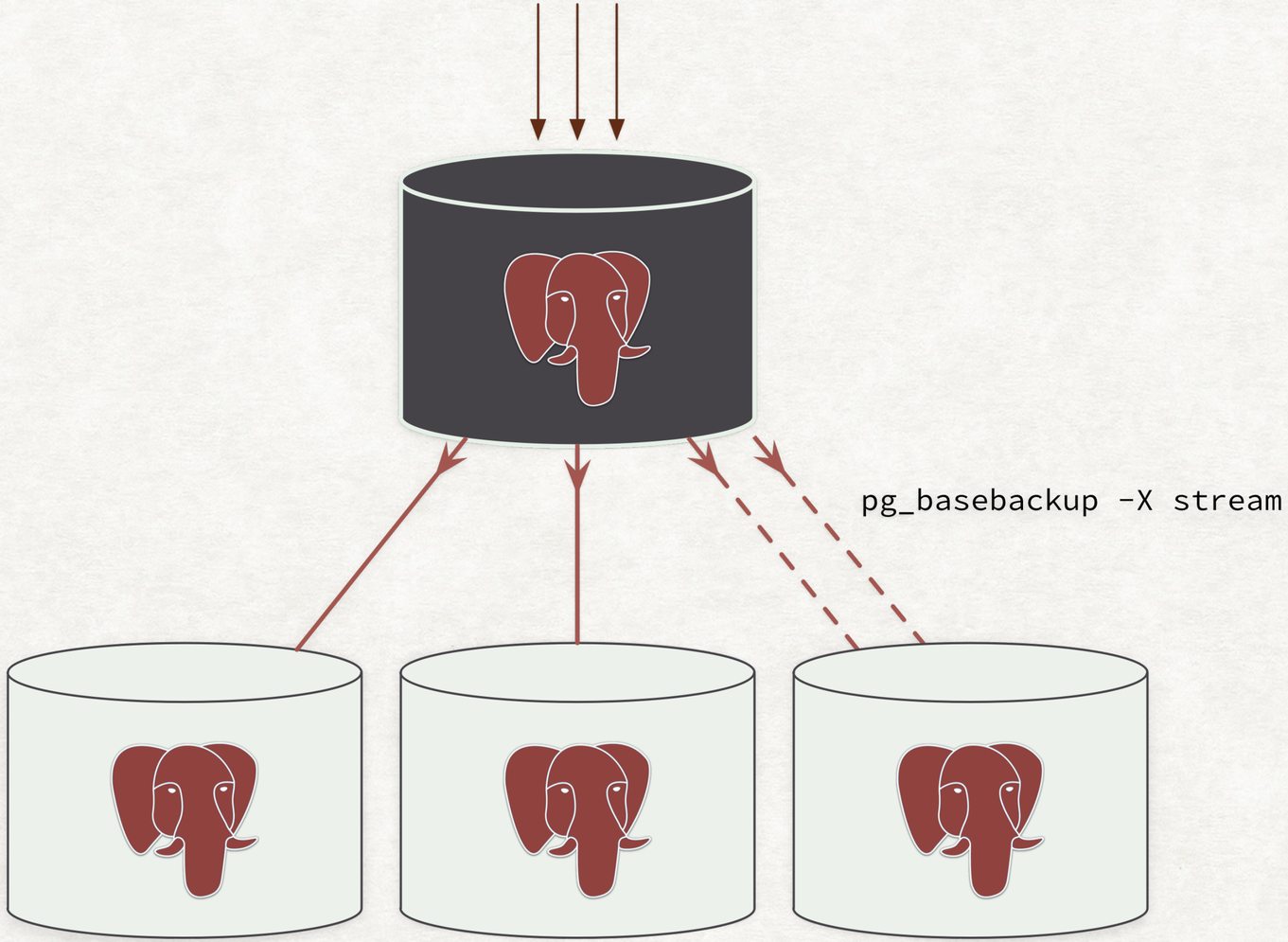
Performing a checksum on every file of a large database to sync a few missing gigabytes of WAL is quite inefficient.

Oh well. It's almost the weekend, and traffic is low. I could probably start a fresh clone, and keep `wal_keep_segments` higher for now.

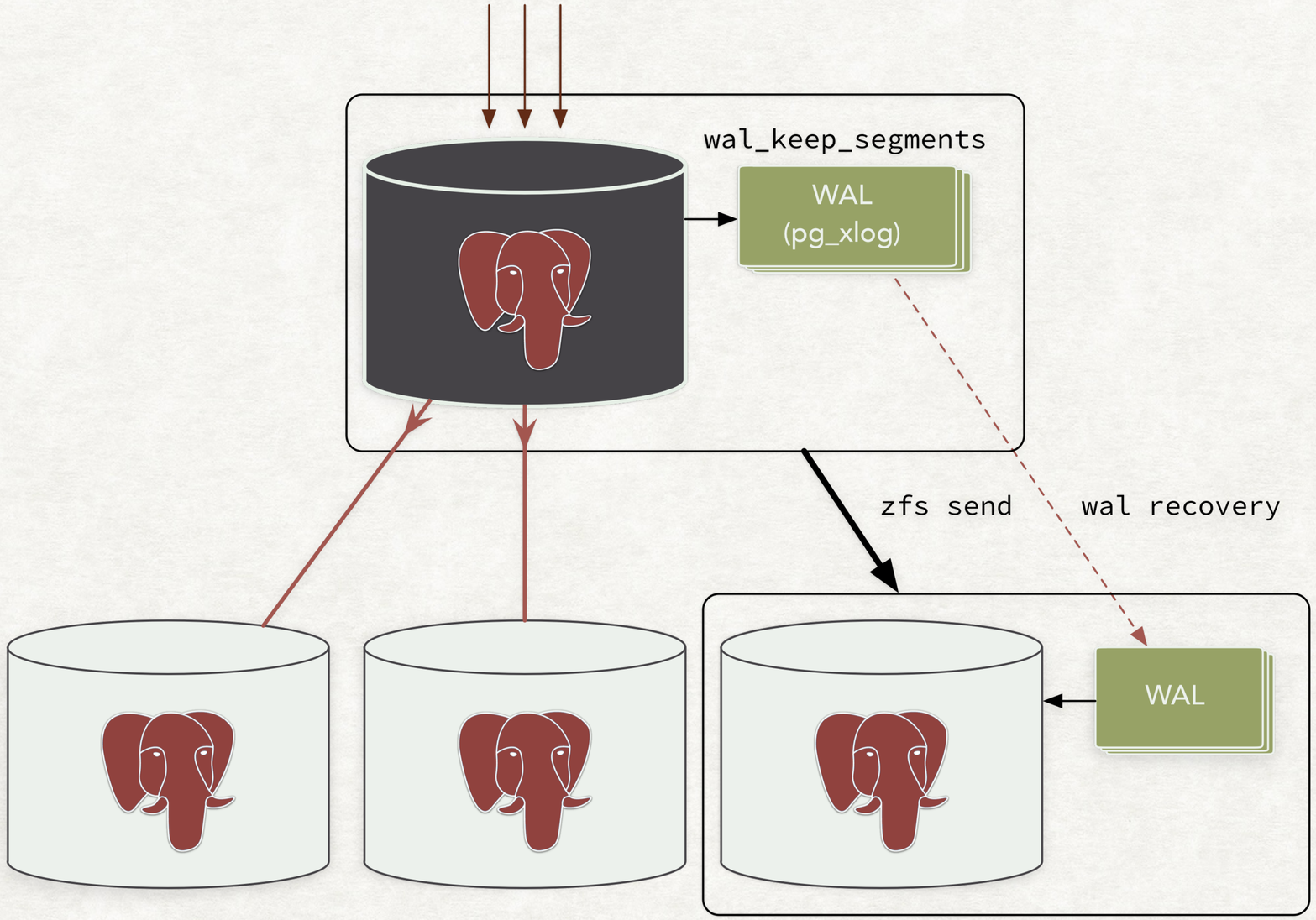
Okay, that should work for now.
But how do we fix it so it doesn't happen again?



One way to fix this is to archive WALs and recover from the archive
if the required WALs are unavailable on master.



Another way is to stream the transaction log in parallel while running the base backup. This is available since PostgreSQL 9.2.



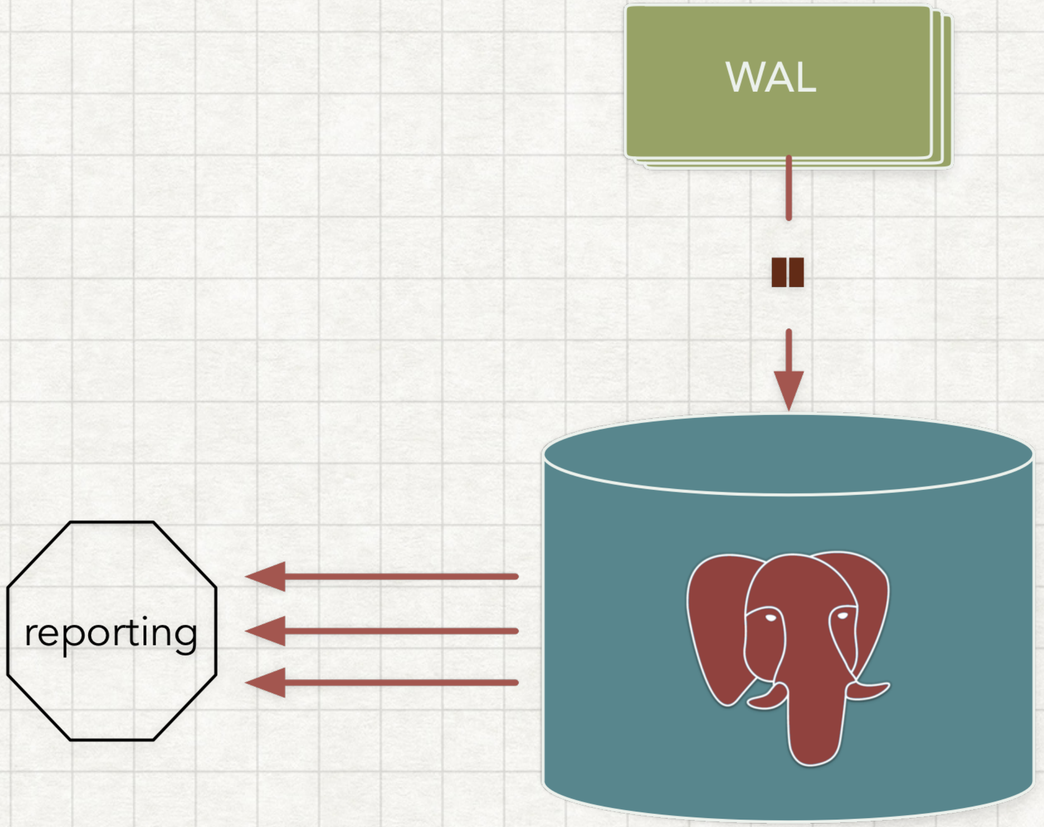
Yet another way is to use filesystem backups, and then let the standby catch up.

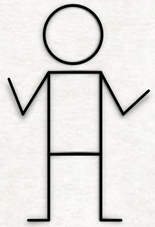
What have we learnt?

- Cloning a standby can take **longer than expected** sometimes. Be prepared.
- WAL recovery is an integral part of the clone. Think about it, prepare for it.
- **WAL archives** are a good way to backup data in a replay-able manner
- Doing a fresh base backup **might be faster** than rsync-ing the data dir.
- Filesystem backups are also an efficient way to get base backups
- Things to monitor: network throughput, DB load and disk i/o on master

STORY #4

Too many long running queries

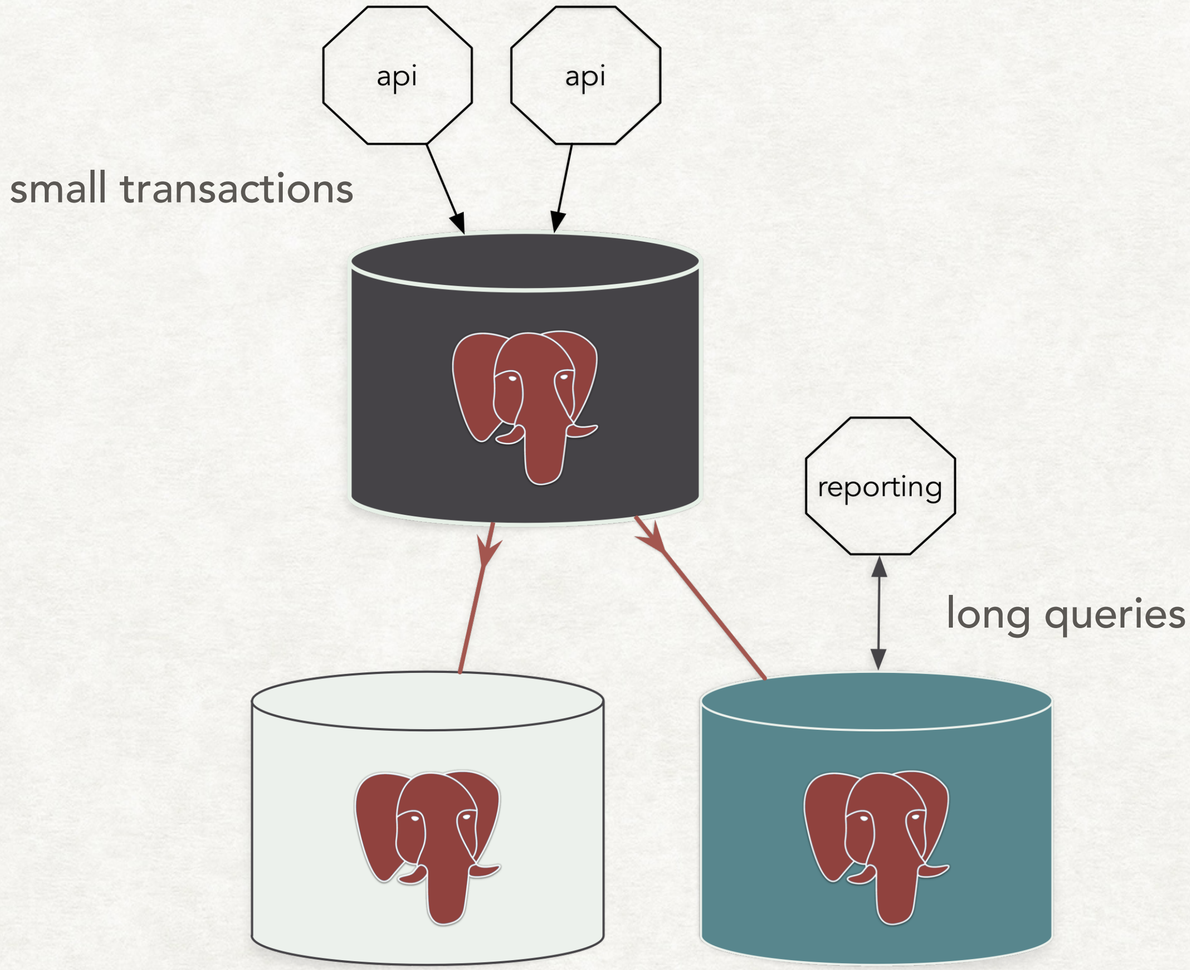




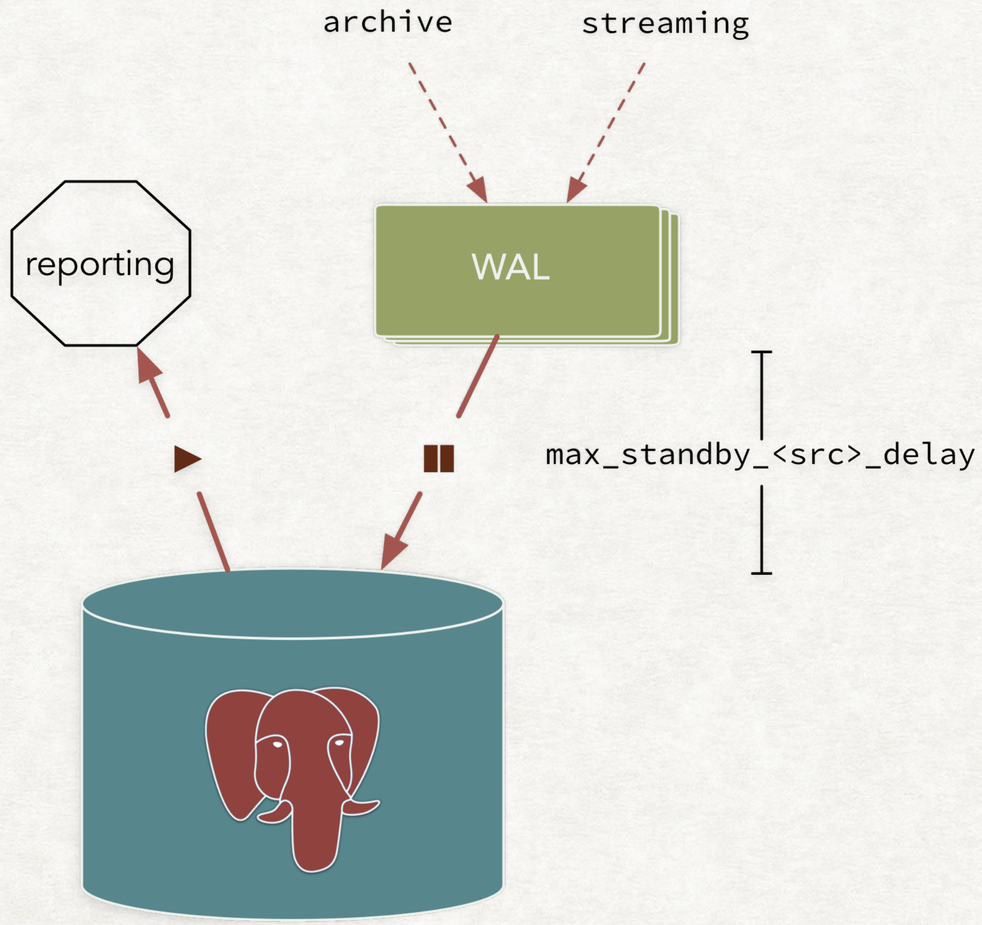
Hey, we're getting many 500s while running reports now. What's going on?

Oh no, too many long queries!

```
ERROR: canceling statement due to conflict with recovery
Detail: User query might have needed to see row versions
                                        that must be removed
```



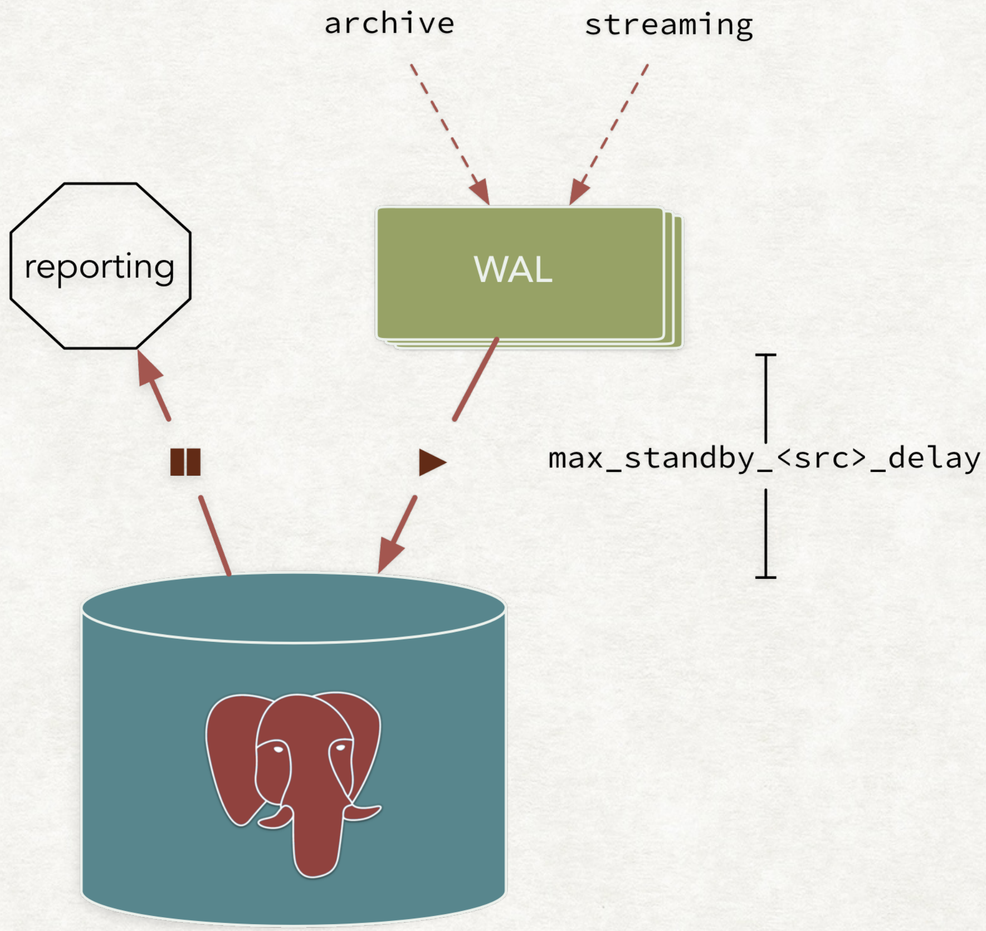
Reporting DB is a part of the cluster because we need live reports.
Generally, it is not a good idea to run report queries on an API DB.



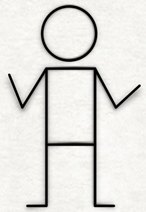
When queries are reading certain versions of rows that are changing in incoming WAL data, the WAL replay is paused.

```
ERROR: canceling statement due
  to conflict with recovery
```

```
FATAL:  the database system
    is in recovery mode
```

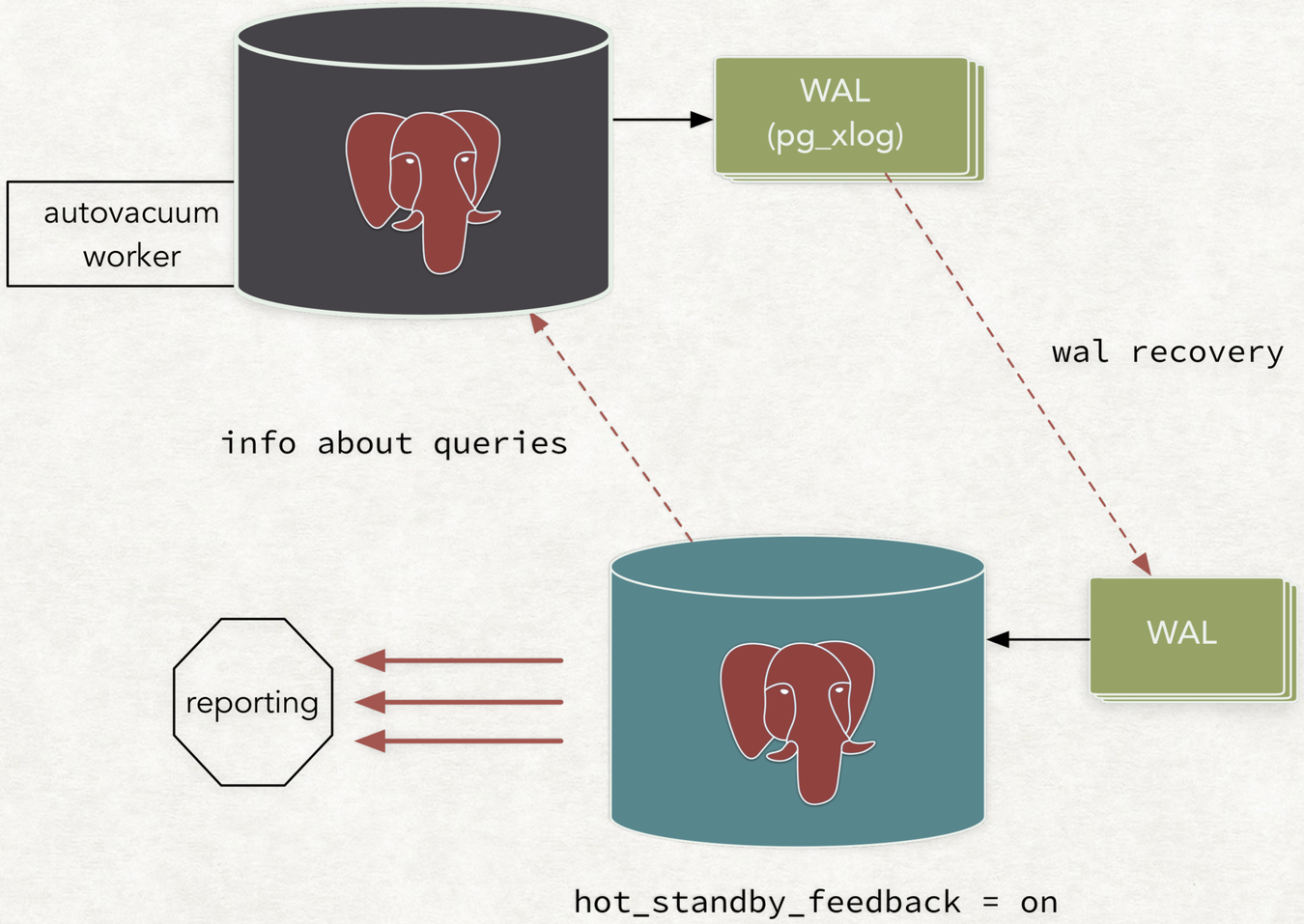


PostgreSQL ensures that you're never lagging back too much by cancelling queries that exceed the configured delay time.



Fix it quick, and fix it forever.

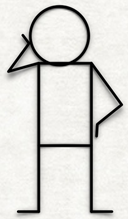
For now, we can just increase `max_standby_streaming_delay`. But, is it okay if the primary gets bloated a bit based on the queries we run?



`hot_standby_feedback` will ensure the primary does not vacuum the rows currently being read on standby, thereby preventing conflict.

No, let’s not do that. We have enough bloat already.

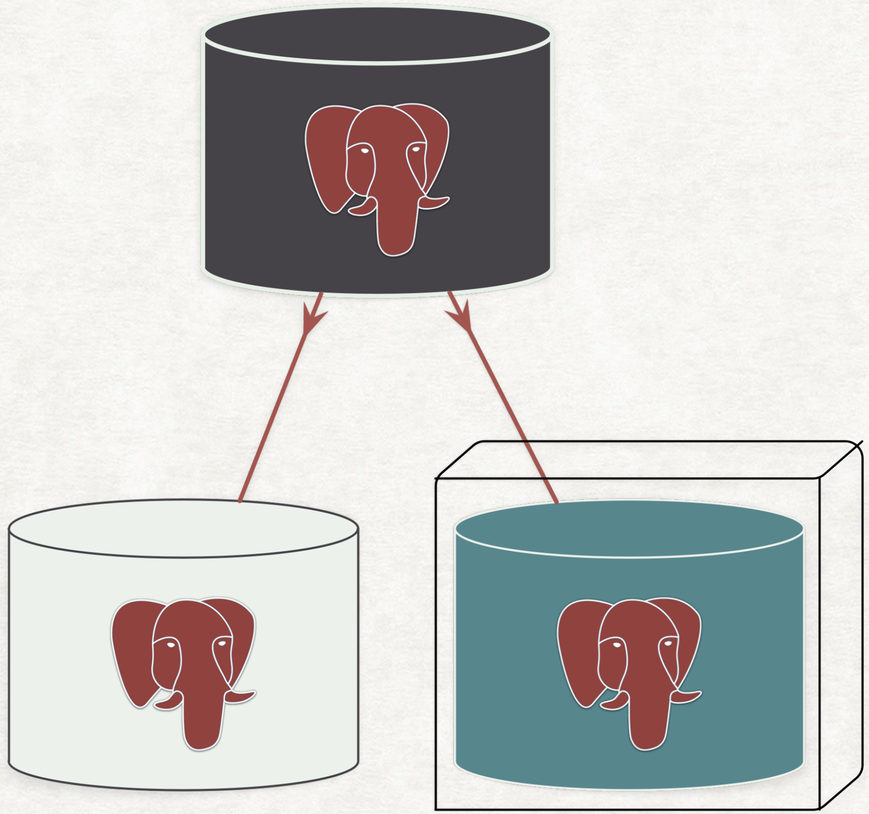
Then we don’t have much choice; we’ll have to make our queries much faster.



Say, shouldn't we be using a **star schema** or **partitions** for our reporting database anyway?

Streaming replication, remember? We can't change the schema for the reporting database alone.

But, we **can** change the hardware underneath.



Reporting box can benefit from heavier, chunkier I/O and parallelism.

```
> IOPS
> ZFS record-size
> Cores
```

PostgreSQL replication does not work across schemas, versions or architectures. However, we can change the underlying h/w and f/s.

What have we learnt?

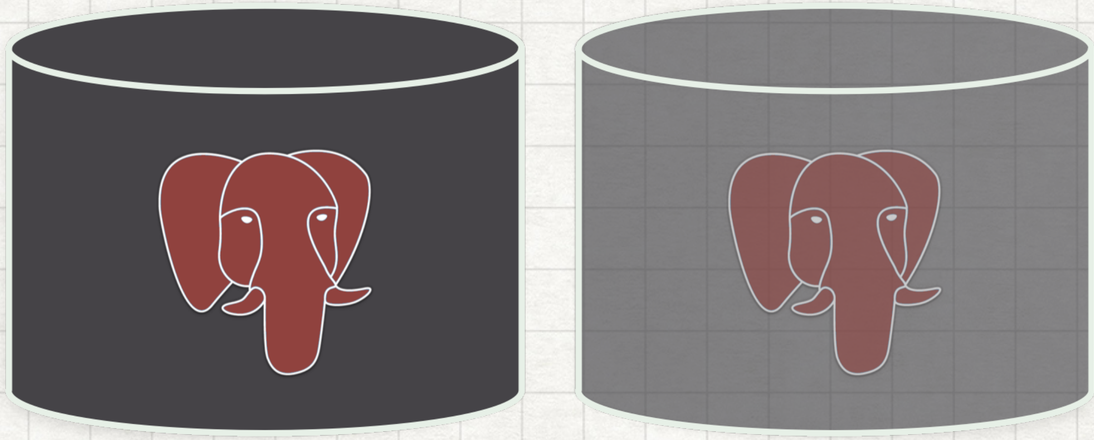
- Regulate the **number of slow queries** running on standbys. They might be cancelled.
- We **cannot use different schemas** on masters and standbys.
- Applying back pressure on primary is an option, but use it carefully.
- We can change the underlying **filesystem or hardware** without affecting replication.
- Things to monitor: replication lag, slow queries, bloat on important tables, vacuum

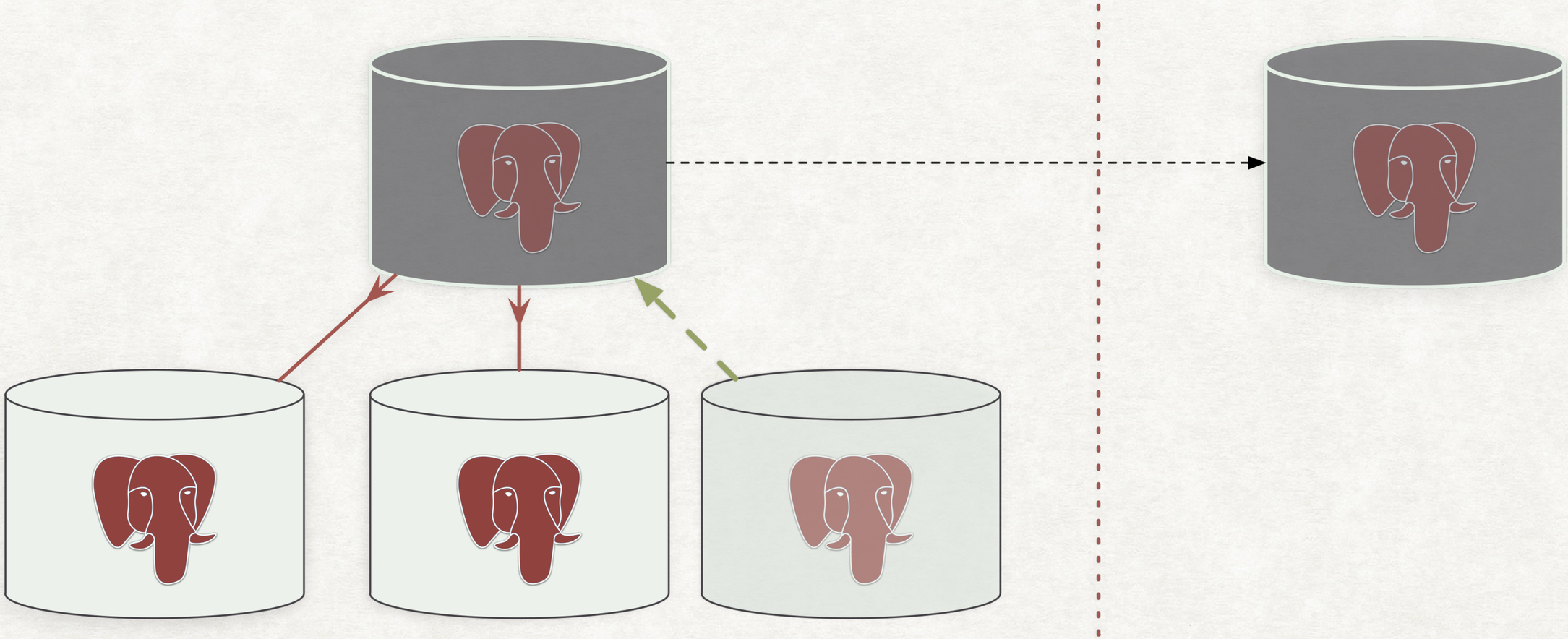
Other Solutions

- **`hot_standby_feedback:`** trade off bloat for longer queries
- **Partitioning:** implies heavier transactions, but enables parallel I/O.
- **Logical replication:** transform SQL stream for the reporting schema
- **Load balance:** distribute load across multiple reporting machines

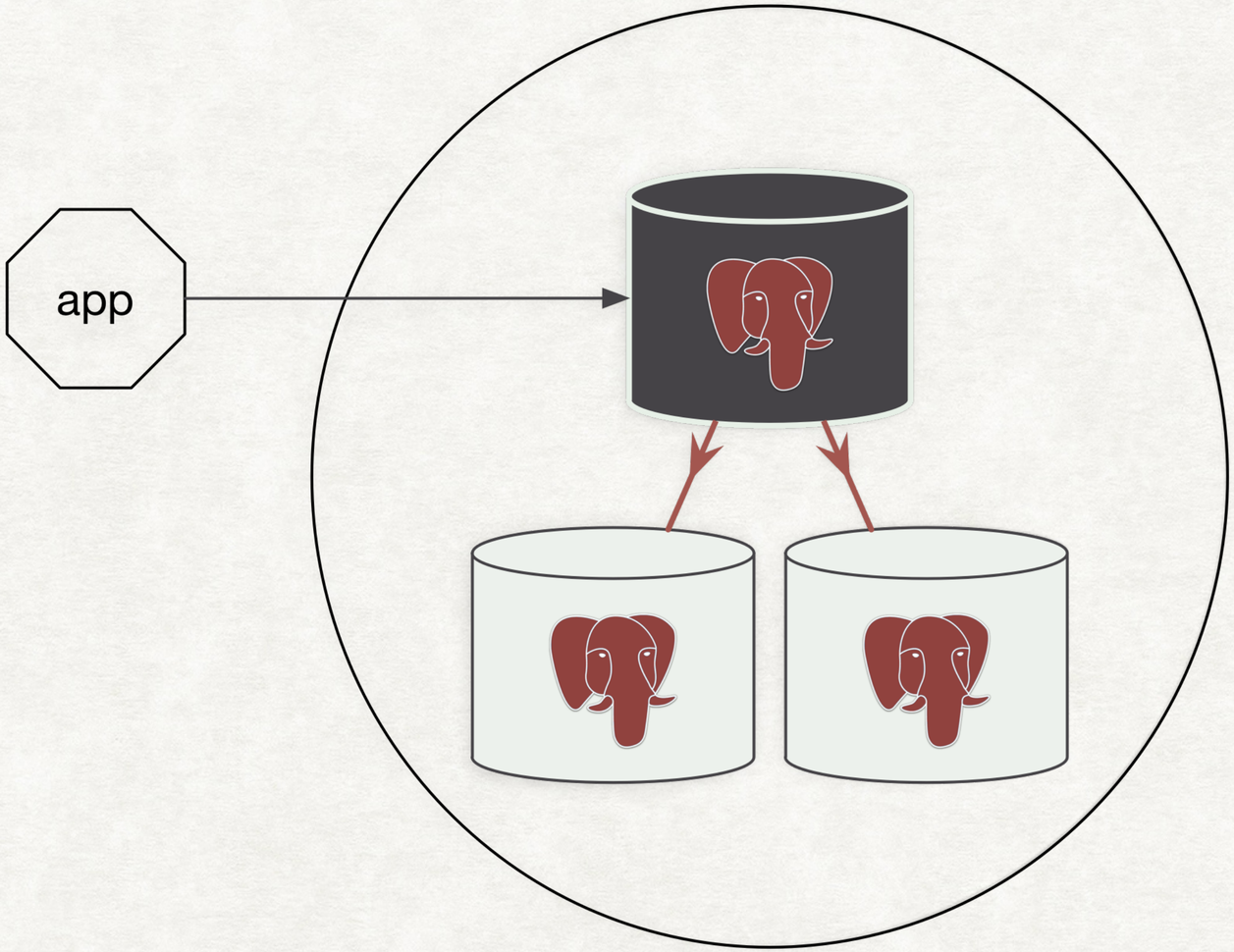
SMALL #1

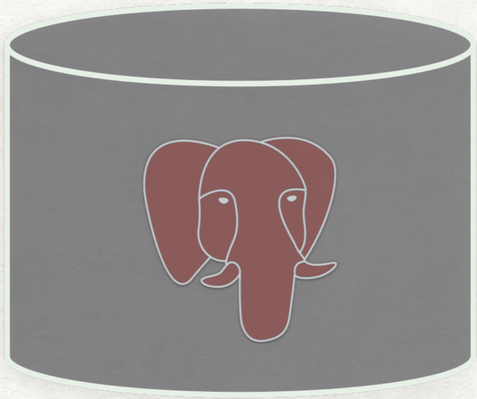
Split Brain





Assume there’s a network partition, and the master is unreachable. A failover kicks in.

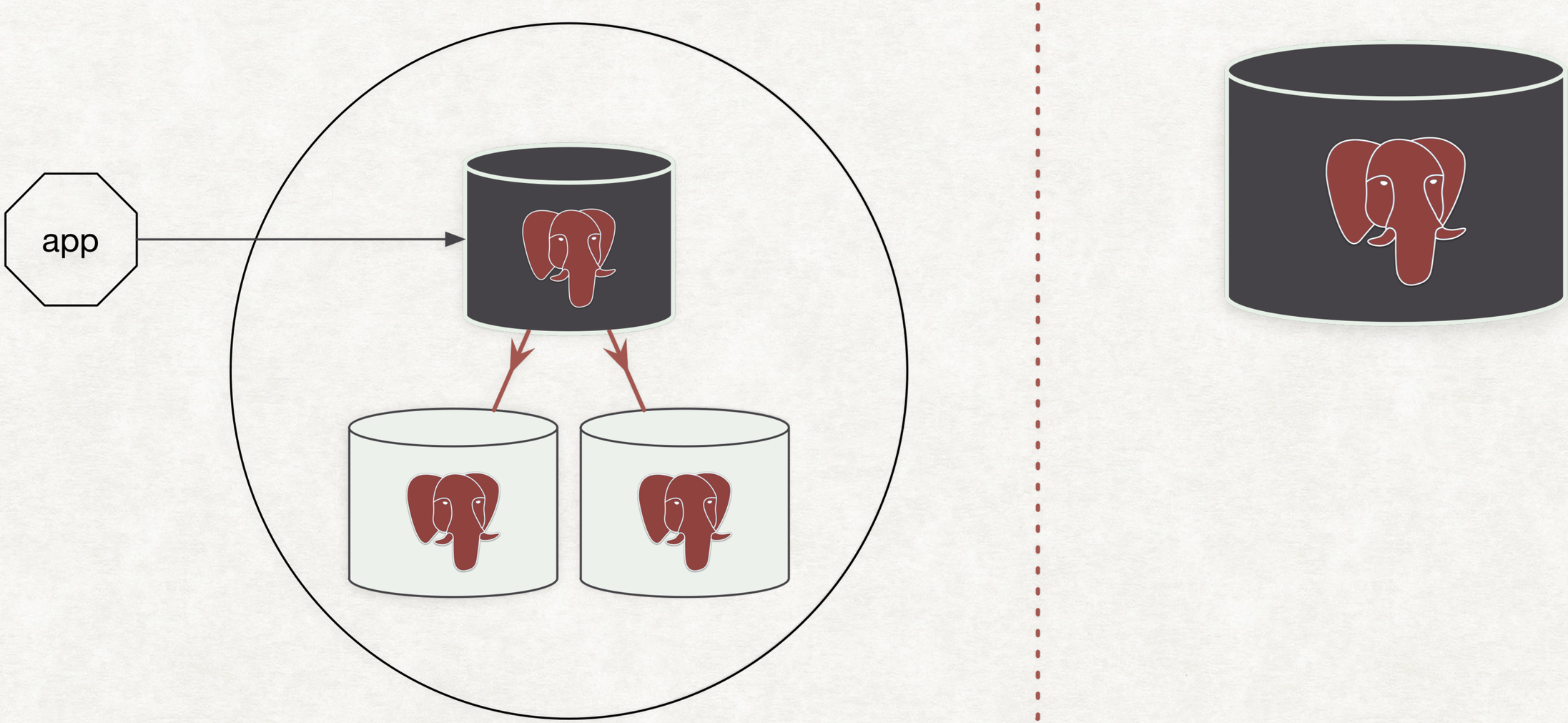




The failover happens successfully, and the application talks to the new and correct primary.

id	type	upstream_node_id	cluster	name	priority	active
1	FAILED		prod	api-0-prod	102	f
2	master		prod	api-1-prod	101	t
3	standby	2	prod	api-2-prod	100	t
3	standby	2	prod	reporting-0-prod	-1	t

Repmgr marks the node failed, as can be seen in the `repl_nodes` table.



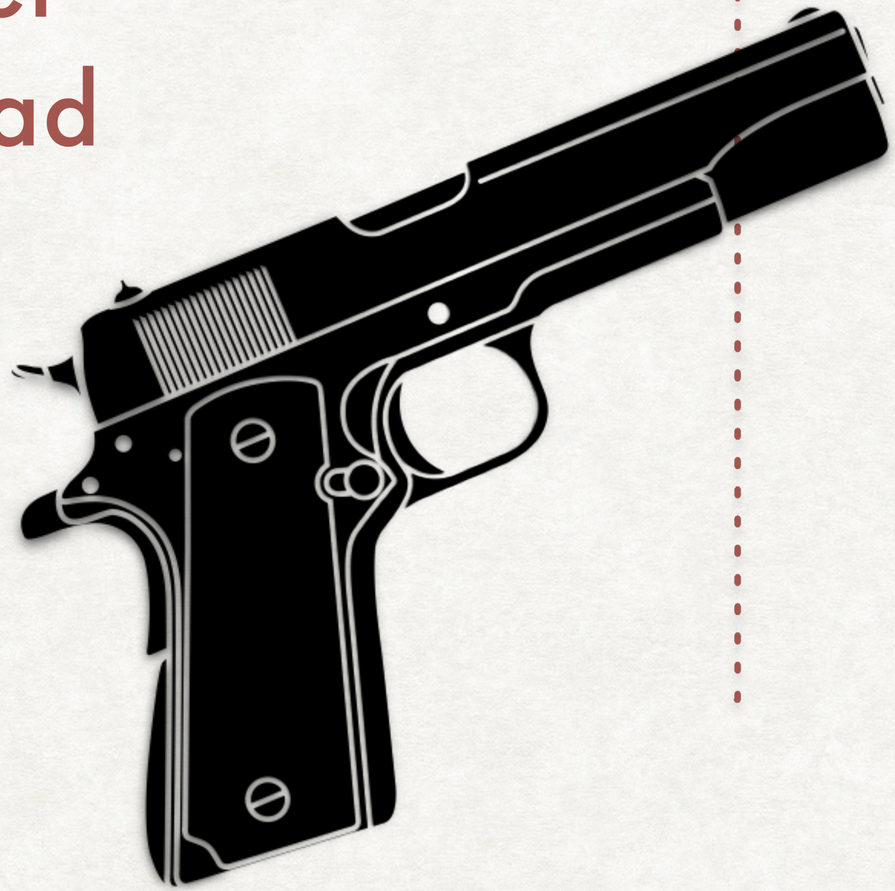
But then, the master that went down, comes back up,
and we have two masters!

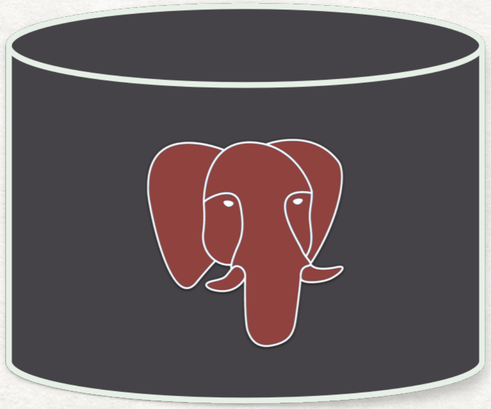
id	type	upstream_node_id	cluster	name	priority	active
1	**master**		**prod**	**api-0-prod**	**102**	**t**
2	master		prod	api-1-prod	101	t
3	standby	2	prod	api-2-prod	-1	t
3	standby	2	prod	reporting-0-prod	-1	t

Now repmgr shows both nodes as masters, and we have a **split brain**.

STONITH

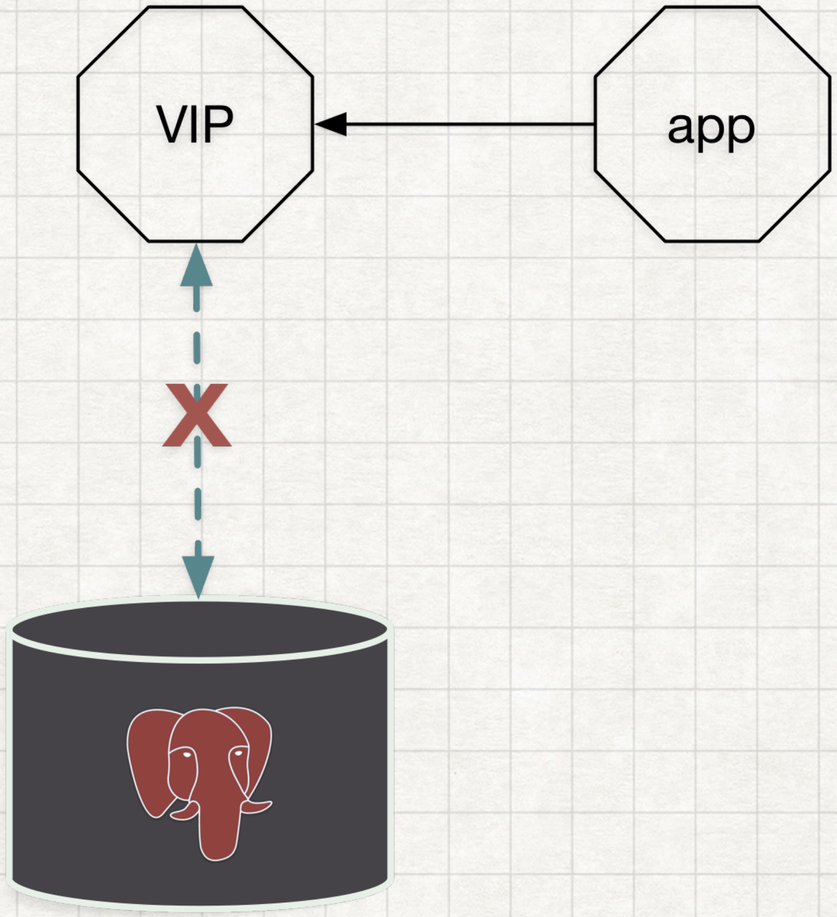
Shoot the other node in the head

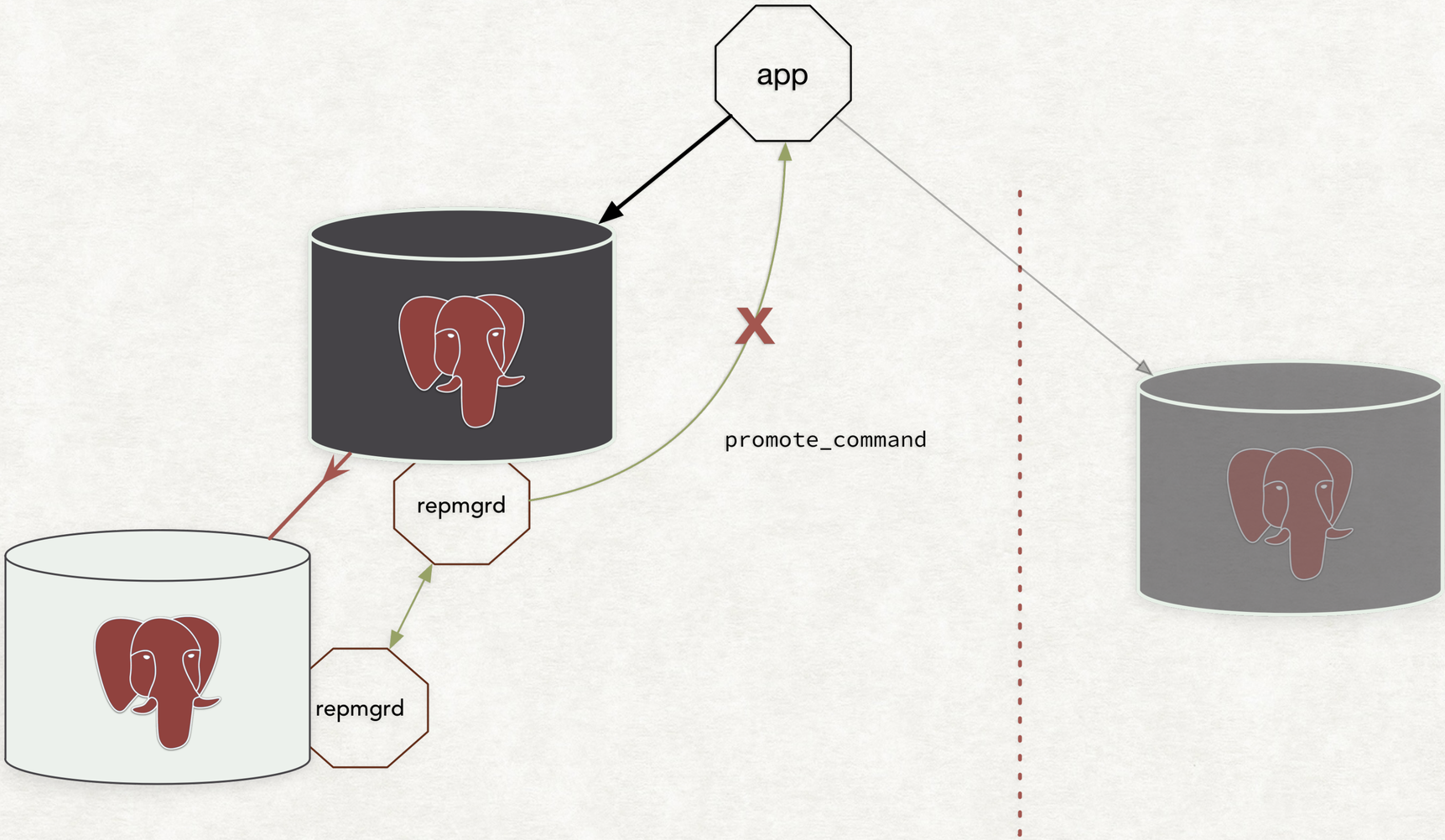




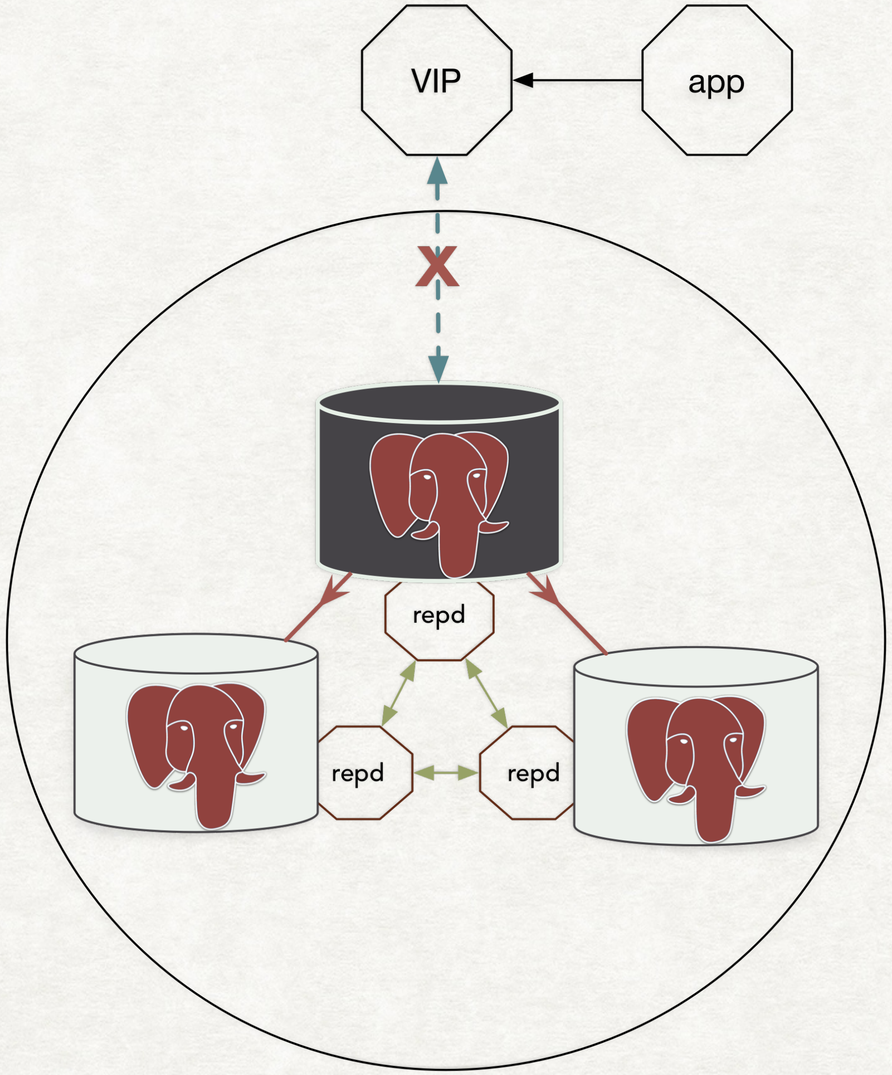
SMALL #2

The app doesn't know about a failover

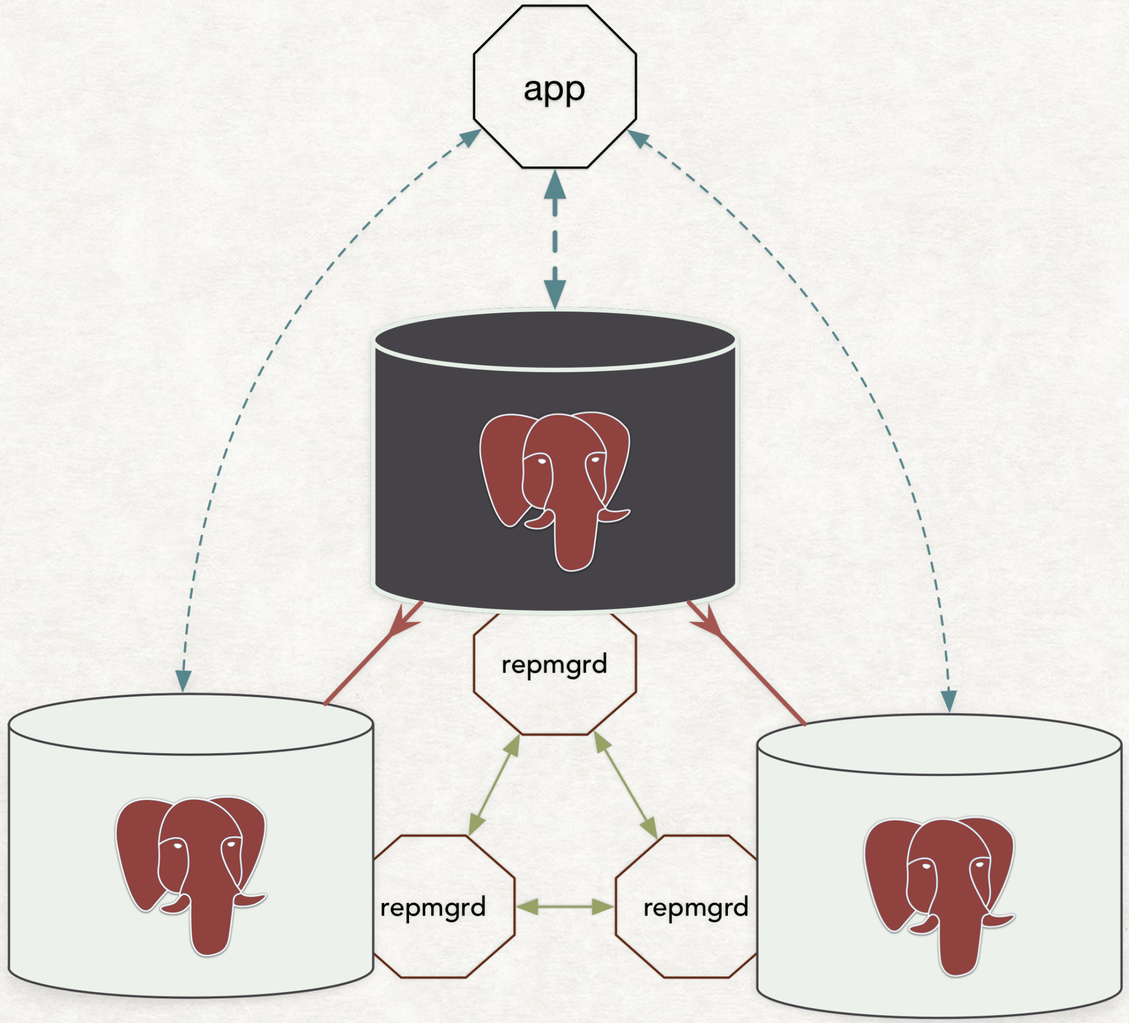




A network failure or a bug in the promote_command might cause the application to not failover correctly.



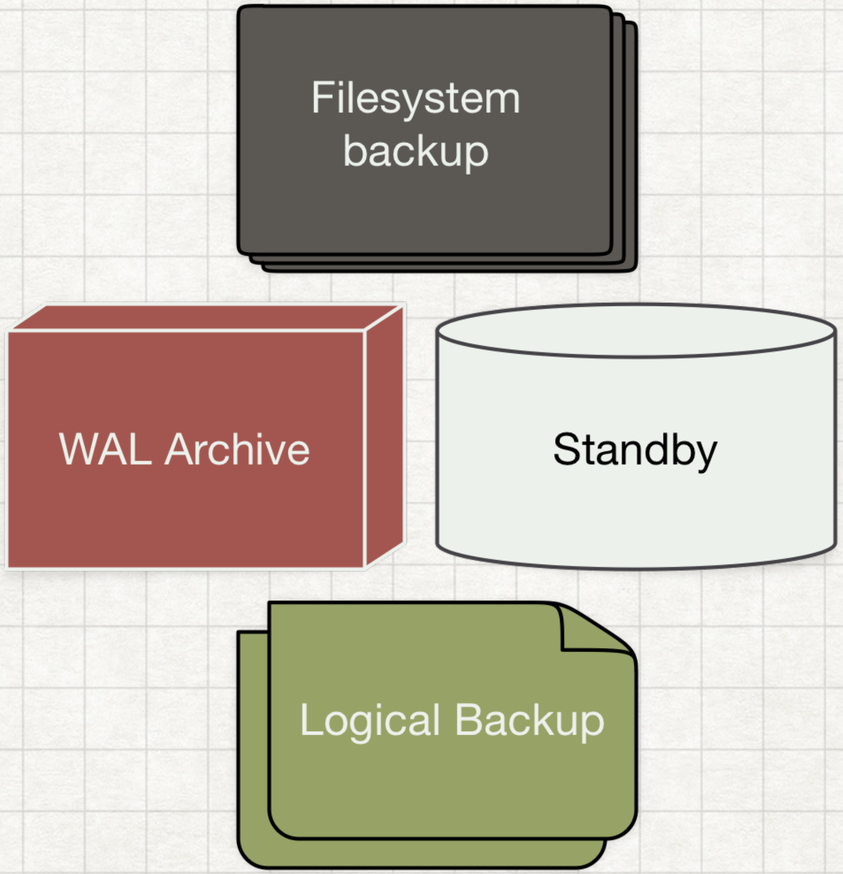
If the Virtual IP switch does not happen correctly, and there was a failover in the cluster underneath, we have the same problem.

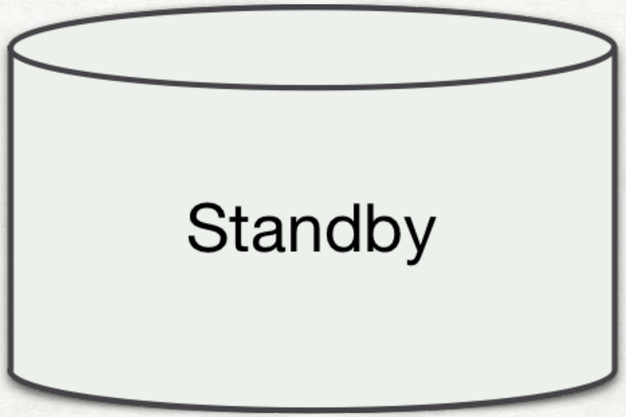


One way to fix this would be to have multiple lines of defence in detecting a failover. The poll strategy described earlier is one such solution.

SMALL #3

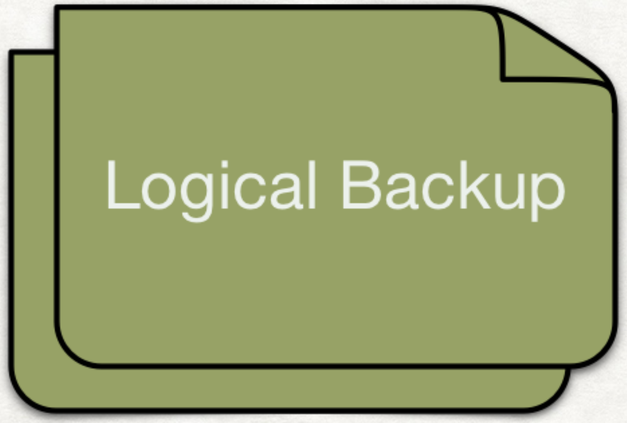
Let's talk about backups





+ Integral, already live, backup size is DB size
− Deletes/truncates cascade, not rewind-able

+ Replayable, helps resurrecting standbys
− Backup size, network bandwidth, redo time

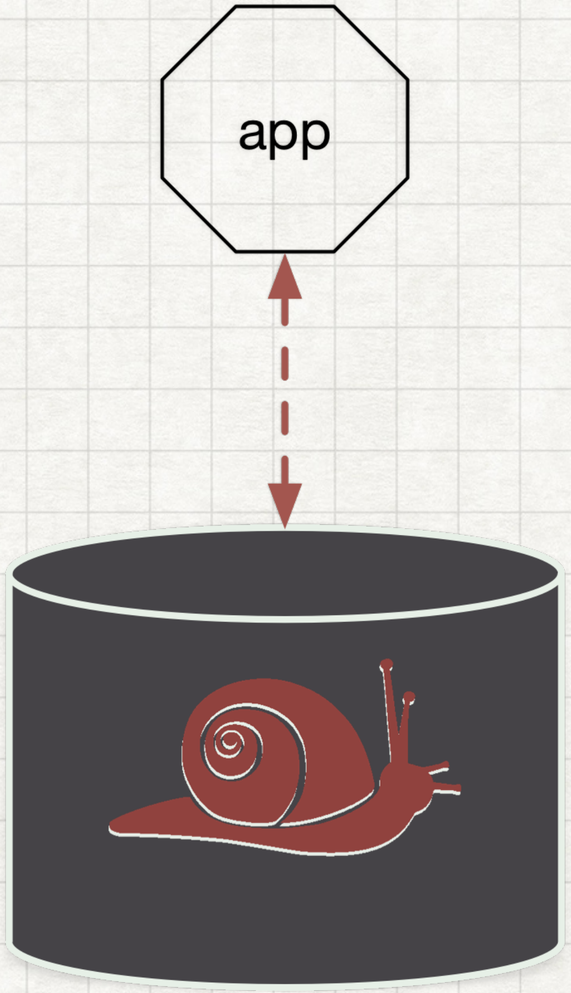


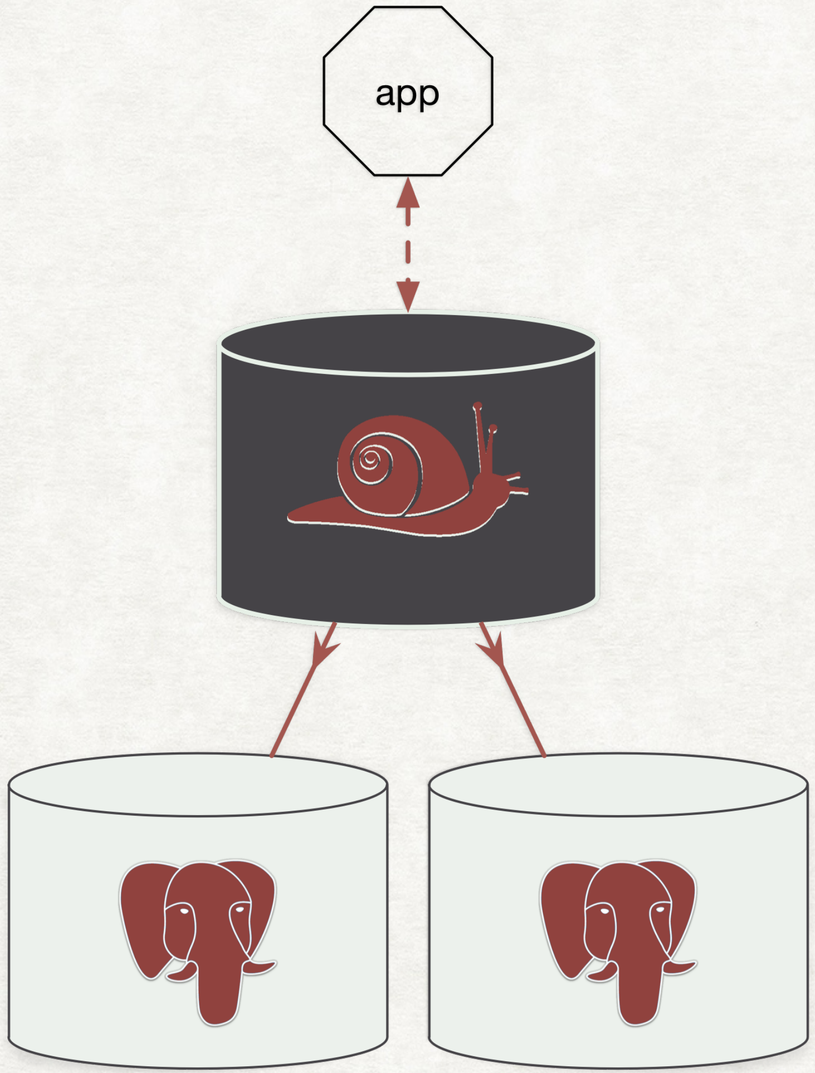
+ Integral, selective, cross architecture
− Slow, high disk I/O, requires replication pause

+ Fast, cheap, versioned
− Integrity risk, restoration time, disk space bloat

SMALL #4

The primary is slow, not dead





The primary is slow because disk I/O has degraded. But, this doesn’t trigger a failover.

Possibly, one of the standbys could do a better job being the master.

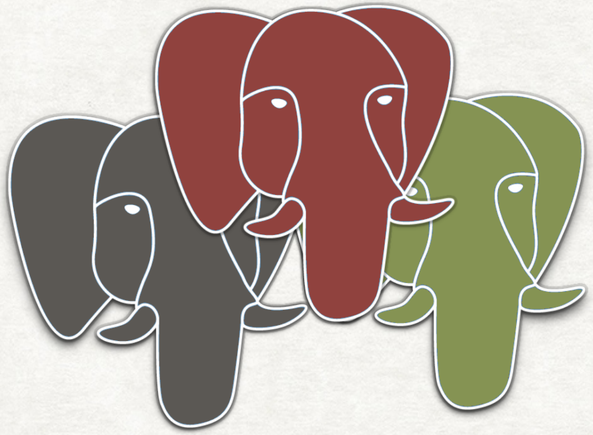
What would you do, to detect and fix the issue?

What have we learnt?

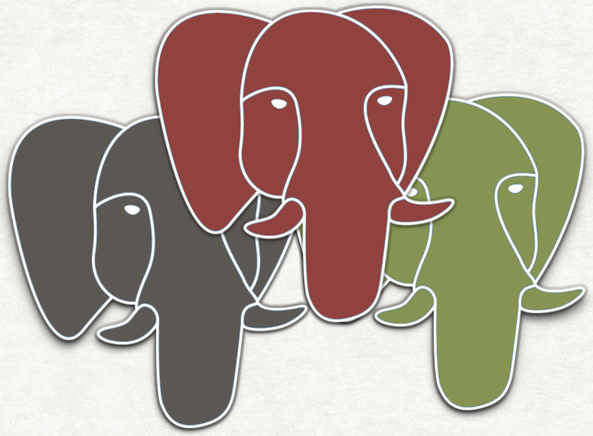
- **PostgreSQL clusters exist.**
  - They are beautiful, useful, and we can build and instrument them ourselves.

- **Disasters happen.**
  - We should protect the cluster against known failure scenarios and rigorously test it. And when disasters do strike, we should be prepared to deal with them.

Thank you!



Of the building of a PostgreSQL cluster



Srihari Sriraman
nilenso