Building a Lightweight High Availability Cluster Using RepMgr

Stephan Müller

June 29, 2018

Schedule

- Introduction
  - Postgres high availability options
  - Write ahead log and streaming replication
  - Built-in tools
- Cluster management with RepMgr
  - Configuration and usage
  - Automatic failover with RepMgrD
- Backup and Recovery with BarMan
  - Configuration and usage

- Wrap-up & Discussion
- Please ask questions

Personal Background

- IT Operations, since 2.5 years
- OLMeRO
  - Swiss market leader for internet solution for construction sector
  - Tender and construction site management
- renovero.ch
  - Craftmens' offerings for private customers
- Belongs to tamedia portfolio
  - Publishing company
  - Digital market places
- Mathematics and Computer Science in Berlin
  - Cryptography, Category Theory
- Thank you PGDay.ch'17

Postgres High Availability Options on Different Layers

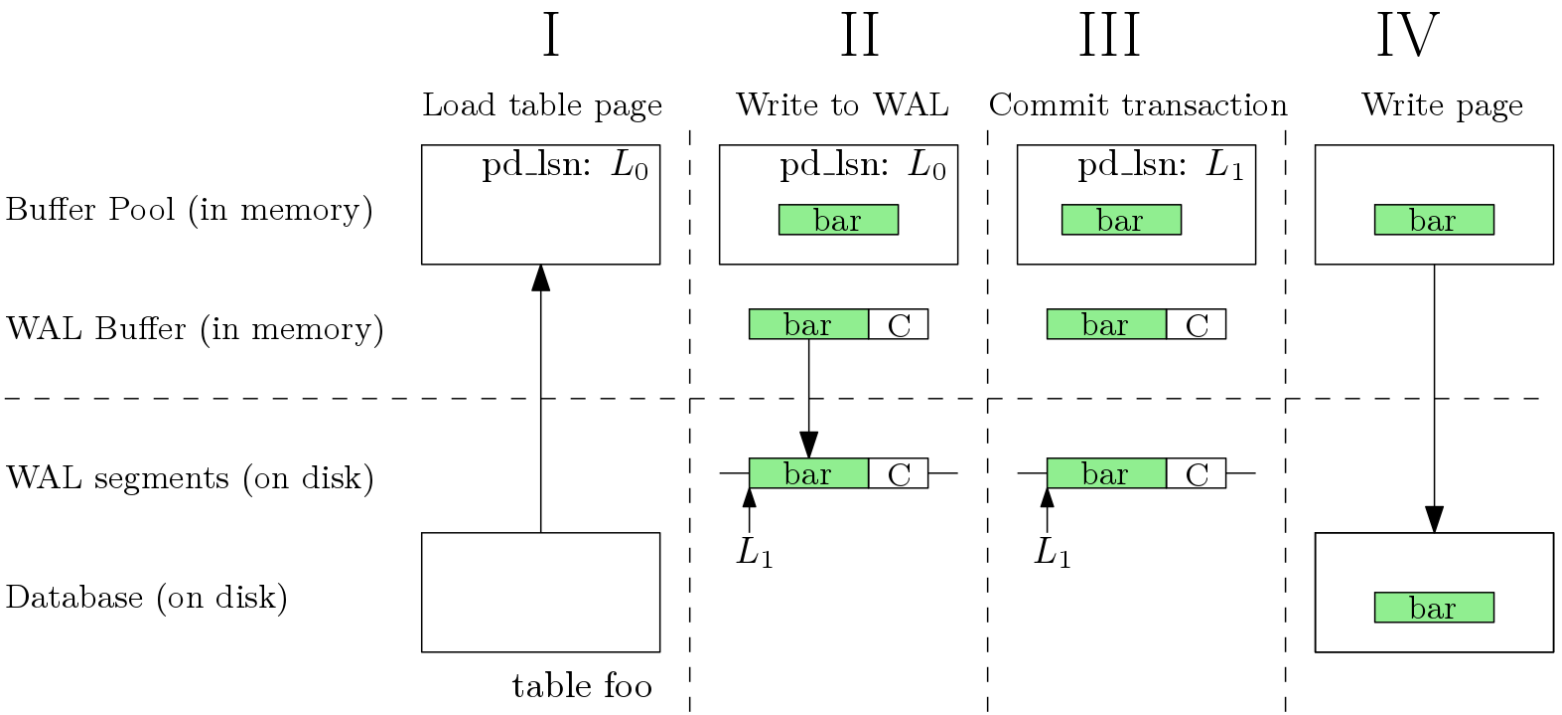
- Hardware
  - SAN
  - Transparent to OS and postgres
  - Fails spectacularly
- Operating system
  - Distributed Replicated Block Device (DRDB)
  - SAN in Software
- Database physical
  - WAL based: Log shipping ($\geq v8.3$)
  - WAL based: Streaming replication ($\geq v9.0$)
- Database logical
  - PGDay.ch'18: Harald Armin Massa $\rightarrow$ 11:00
  - FOSDEM'18: Magnus Hagander
- App-in-db
  - Slony-I (trigger based)
- Application

Introduction: Postgres Write Ahead Log

- Before committing any transaction (i.e. set state COMMITTED in clog), the transaction is written to WAL and flushed to disk
- One big virtual file (16 EB)
- Divided into logical files (4 GB)
- Divided into segments (16 MB)
  - This is what you see on your disk
  - pg_xlog/ $\underbrace{\text{0000000A}}_{\textit{timeline}}\,\underbrace{\text{0000083E}}_{\textit{block}}\,\underbrace{\text{000000B1}}_{\textit{segment}}$
- Divided into pages (8 KB)
- Contains xlog records with transaction data
- Log Sequence Number (LSN) is a byte address in WAL
- SELECT pg_current_xlog_location(); 83E/B18FE7C0
  - Address 8FE7C0 in segment 0000000A0000083E000000B1

Introduction: Postgres Write Ahead Log

- BEGIN; INSERT INTO foo VALUES('bar'); COMMIT;
- Each page has a pg_lsn attribute:
  - Contains the LSN of the last xlog record which modified that page



Recovery After a Crash Using the Write Ahead Log

- Your server just crashed
- After a restart:
- Uncommitted data?
  - It's lost.
- Committed but not yet written to db?
  - Start replaying missing records from WAL
    - Where to start?
    - Form last checkpoint. Location saved in pg_control file
    - pg_controldata /your/data/dir
- Corrupted page writes?
  - full_page_writes = on
  - Insert complete backup of pages into WAL
  - That makes your WAL so big: ~8K for each modified page
- In short: Write Ahead Log is the D in ACID

Write Ahead Log and Streaming Replication

- Idea: Copy WAL to other postgres servers
- Remote server indefinitely replays from WAL
  - Log Shipping: "Just copy WAL segments"
  - Streaming Replication: Copy individual xlog records
- Different levels of replication: synchronous_commit

off	Everywhere asynchronous
local	Locally synchronous, remote asynchronous
on	Wait until remote server has written to WAL
remote_apply	Wait until remote server has commited

- synchronous_standby_names
- Tradeoff: Saftey vs Performance
- Tunable on transaction level

Postgres Streaming Replication Benefits

- Built-in
- Easy to set up
- Hard to break
- Easy monitoring: All or nothing
  - SELECT * FROM pg_stat_replication;

pid	20841	
usename	repmgr	
application_name	db02	remote server
backend_xmin	294106915	
state	streaming	OK
sent_location	83E/F92947F0	
write_location	83E/F92947F0	in memory
flush_location	83E/F92947F0	on disk
replay_location	83E/F92947B8	applied to db
sync_state	async	
[...]		

Streaming Replication: Easy Setup

- Prepare primary:
  - postgres.conf

```
listen_addresses = '192.168.0.10'
max_wal_senders ≥ #nodes + 2
wal_level = replica
wal_log_hints = on                    for pg_rewind
```

- Special user:

```
CREATE ROLE repuser WITH REPLICATION
```

  - Dont forget hba.conf and your firewall
- Prepare standby:

```
pg_basebackup -h primary -P -U repuser -X -R
```

  - postgres.conf:

```
hot_standby = on
```

  - Adjust recovery.conf
- Done. Ok, it is more complicated but not much

Cluster Management Solutions

At the end of the day: **You want an easy failover solution.**

- Patroni
  - Focuses on automatic failover
  - Based on etcd / zookeeper
- RepMgr
  - Wraps built-in commands
  - Focuses on manual failover
  - Automatic failover with repmgrd
  - Very slim
- PAF (postgres automatic failover)
  - Focuses on automatic failover
  - Based on corosync / pacemaker
  - Using virtual IPs

Overview: RepMgr (Replication Manager)

- `https://repmgr.org/` (Source on github)
- Developed by 2ndQuadrant, written in C
- Packaged for most distributions
  - Use 2ndQuadrant repository
  - Depending on your postgres version:

    ```
    dnf install repmgr96          (or repmgr10, etc)
    ```

  - Few dependencies to build from source
  - Well documented
- Only manual failover (i.e. switchover)
- Tuneable to automatic failover
- Plays well with BarMan (Backup and Recovery Manager)

Setting up RepMgr on Primary

- Start with your primary postgres node
- Create repmgr user (superuser or replication privilege)

```
createuser -s repmgr
```

- Create db for metadata

```
createdb repmgr -O repmgr
```

- Adjust hba.conf
  - Allow repmgr user to connect to its db, local and remotely
- Prepare repmgr.conf

```
node_id    =  1
node_name  =  db01                          dont use role names
conninfo   =  'host=db01.olmero.ch
               user=repmgr
               dbname=repmgr'
```

RepMgr Usage: Start a Cluster

- General pattern: repmgr [options] <object> <verb>

  object $\in \{$primary, standby, node, cluster , witness$\}$

  verb $\in \{$ register , clone, follow , switchover , check, show, . . .$\}$
- Register primary node

```
repmgr primary register
```

  - Installs some extensions
  - Adds entry to repmgr database

```
SELECT * FROM repmgr.nodes;
    node_id           | 1
    upstream_node_id  |
    active            | t
    node_name         | db01
    type              | primary
    location          | default
    priority          | 30
    conninfo          | host=db01.olmero.ch dbname=repmgr user=repmgr
    repluser          | repmgr
    slot_name         |
    config_file       | /etc/repmgr.conf
```

RepMgr Usage: Adding Nodes to Your Cluster

- Start with empty data directory
- Copy and modify repmgr.conf from primary:

```
node_id   = 2
node_name = db02
conninfo  = 'host=db02.olmero.ch
             user=repmgr
             dbname=repmgr'
```

- Clone primary server

```
repmgr -h db01.olmero.ch standby clone
```

  - Executes a basebackup

    ```
    pg_basebackup -h node1 -U repmgr -X stream
    ```

  - Prepares recovery.conf

RepMgr Usage: Adding Nodes to Your Cluster (cont)

- recovery.conf:

```
standby_mode = 'on'
recovery_target_timeline = 'latest'
primary_conninfo = 'host = db01.olmero.ch
                    user = repmgr
                    application_name = db02'
restore_command = '/usr/bin/barman-wal-restore
                   barman olmero %f %p'
```

- Start postgres server - Done.
- Streaming replication is running

RepMgr Usage: Change Primary

- View your cluster: (run on any node)

```
repmgr cluster show
```

ID	Name	Role	Status	Upstream	Location
1	db01	primary	* running		default
2	db02	standby	running	db01	default
3	db03	standby	running	db01	default

- Switch over to other primary: (run on new primary)

repmgr standby switchover

  - You want to start with a healthy cluster
  - Shutdown primary (service_stop_command)
  - Promote local (service_promote_command)
  - pg_rewind old primary
  - Restart and rejoin old primary

Manual Failover with RepMgr

- Promote a standby:
  - Make sure your old primary is dead **and will stay dead**
  - Choose a standby and run

    ```
    repmgr standby promote
    ```

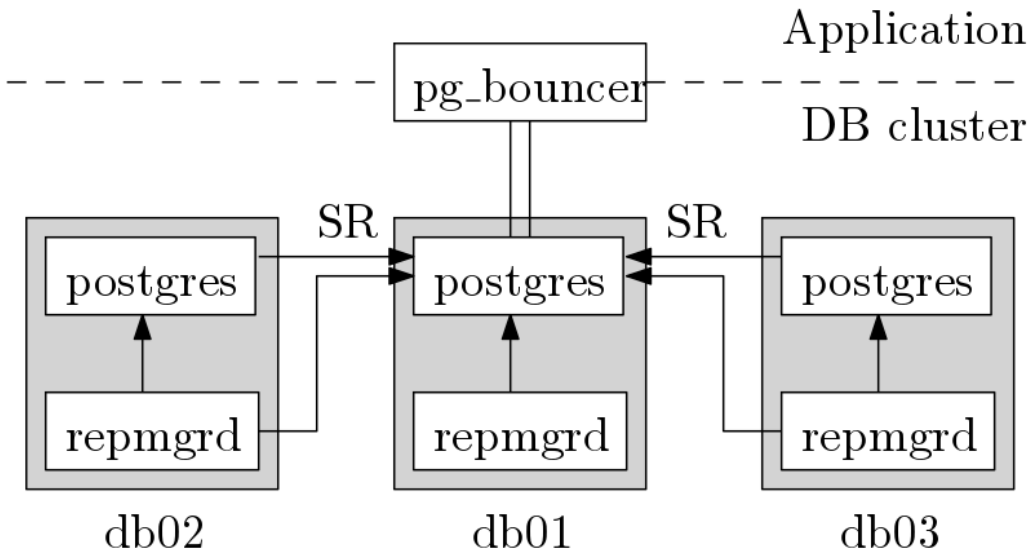
  - Calls service_promote_command from repmgr.conf
- Change the upstream node for your other standbys

  ```
  repmgr standby follow
  ```

- Tell your applications about the new master
  - Use a connection pooler to separate your application and database
  - For example: pg_bouncer
- Your old primary is trashed
  - Delete and clone from new primary

Automatic Failover with RepMgr: Overview

- A repmgrd runs on each postgres node
- repmgrd uses metadata table from repmgr db
  - It knows your postgres cluster
  - But it is not aware of other repmgrds
  - The repmgrds are not a cluster themselves (unlike etcd)
- repmgrd PQpings the clusters primary and its "local" node
- On failure: repmgrd on a standby promote its local node



Automatic Failover with RepMgr: Configuration

- Shared configuration: /etc/repmgr.conf

```
failover = automatic
priority = 100
reconnect_attempts = 10
reconnect_interval = 20
promote_command = repmgr standby promote  # No
```

- Lastest LSN overrules priority
- **No fencing!** Only rudimentary checks are done
- Use a wrapper to do all the logic:

```
promote_command = /your/fancy/failover/script.py
```

  - STONITH in software
  - Eventually call repmgr standby promote
  - In doubt, leave it out

BarMan: Backup and Recovery Manager

- `https://www.pgbarman.org/`
- Developed by 2ndQuadrant, written in Python 2
- Packaged for most distributions
  - dnf install barman
  - dnf install barman-cli (on your postges nodes)
- Physical backups
  - Fast recovery
  - Point In Time Recovery (PITR)
  - No logical backups
- Onsite and offsite backups possible
- Restore functionality

BarMan: Overview

- Think: "**A postgres node without postgres**"
- Copies your data directory
  - pg_basebackup
  - rsync
- Uses streaming replication for continuous WAL archiving
  - pg_receivexlog
- On barmans disk:

```
/data1/barman/olmero/base:
20180626T013002/                    your data dir
20180627T013002/

/data1/barman/olmero/wals:
[...]
0000002E0000084B/                   all wal segments
0000002E0000084C/
0000002E0000084D/
0000002E.history
```

BarMan: Configuration

- Everything in barman.conf

```
[olmero]
                conninfo = host=db01.olmero.ch user=barman
                           dbname=postgres
      streaming_conninfo = host=db01.olmero.ch user=barman

            backup_method = rsync
              ssh_command = ssh postgres@db01.olmero.ch -c arcfour
            reuse_backup = link
            parallel_jobs = 4

      streaming_archiver = on              ; stream wals
                slot_name = barman01       ; use a replication slot
```

- Point barman to your postgres primary
- Additionally:
  - Passwordless SSH login
  - DB connection with replication privilege

BarMan: Commandline Usage

- barman backup olmero
  - Basebackup via rsync
  - Start pg_receivexlog
- barman list backups olmero

  20180627 Wed Jun 27 04:40:39 - Size: 468.3 GiB - WAL Size: 8.5 GiB
  20180626 Tue Jun 26 04:58:48 - Size: 468.4 GiB - WAL Size: 9.5 GiB

- barman check olmero −−nagios

  BARMAN OK - Ready to serve the Espresso backup for olmero

- barman replication −status show
  - Pretty print "SELECT ∗ FROM pg_stat_replication;"

BarMan: How to Restore a Backup

- Restore from backup:

```
barman recover olmero latest
              /data/dir
              --remote-ssh-command "ssh postgres@db01"
              <recovery-target>
```

- Use appropriate recovery target

```
--target-time "Wed Jan 01 09:30:00 2018"
--target-xid 128278783
--target-name "foo"        # SELECT pg_create_restore_point('foo')
--target-immediate         # only recover base backup
```

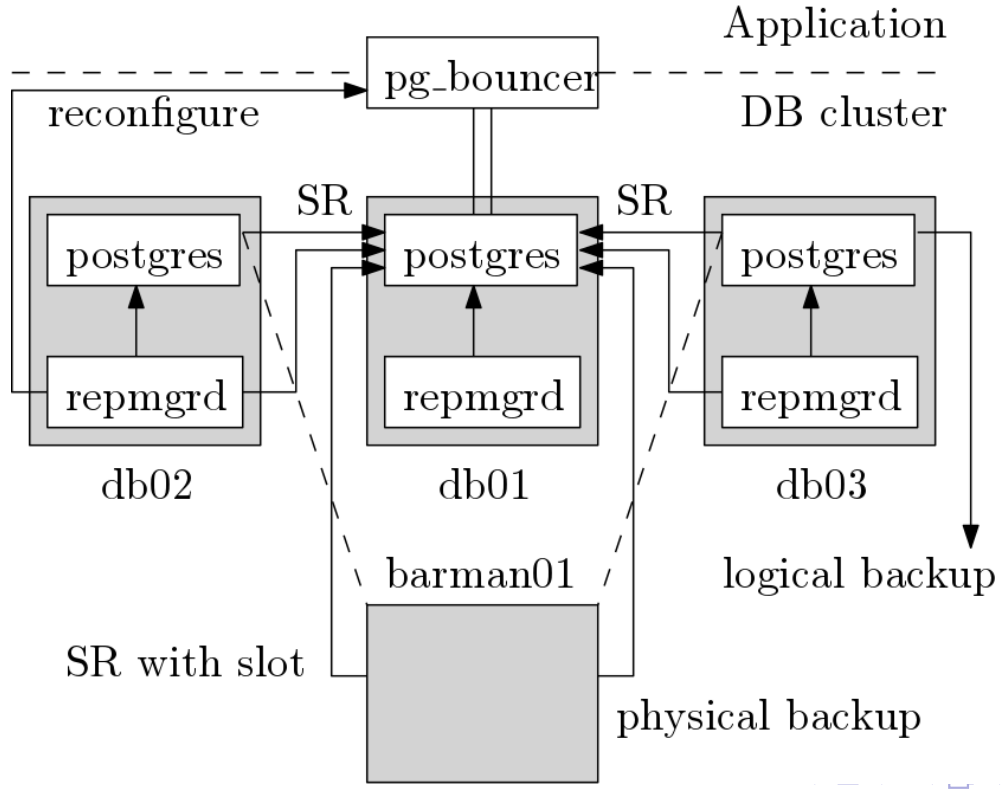
- Restores basebackup via rsync
- Prepares recovery.conf:
  - barman-wal-restore -U barman barman01 olmero %f %p
- Start your postgres server

BarMan and Failover

- Barman has no daemons, no extra processes
  - Everything is a cron job
- Barman is not aware of your cluster
- Check regularly for a new primary
  - You have to write a custom script
    - Adjust config
    - Start streaming from new primary
      barman receive-wal –create-slot olmero
      barman switch-wal olmero
- If your primary changed
  - Timeline will change, no confusion in wal segments
  - Make a new basebackup

Wrap up - Picture at OLMeRO

- repmgr as wrapper arround built-in features
- Very flexible, very slim
- BYOS: You have to bring your own failover logic
  - This is very hard
- Plays well with barman



Thank You

Questions and Discussion