Utilising ProxySQL for connection pooling in PHP

Tibor Korocz
Architect
Webinar
14.08.2018

PERCONA

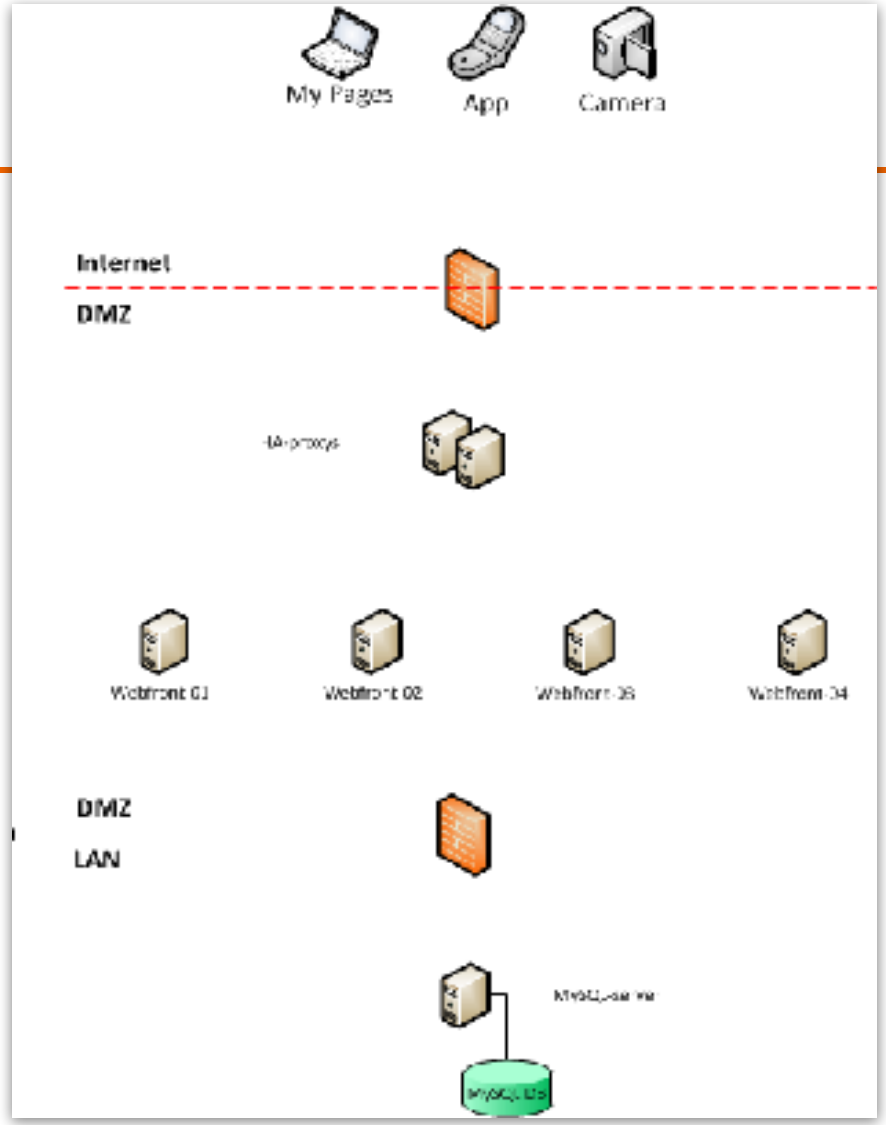
Who is using PHP?

What is our Problem?

Our setup and how we broke it...

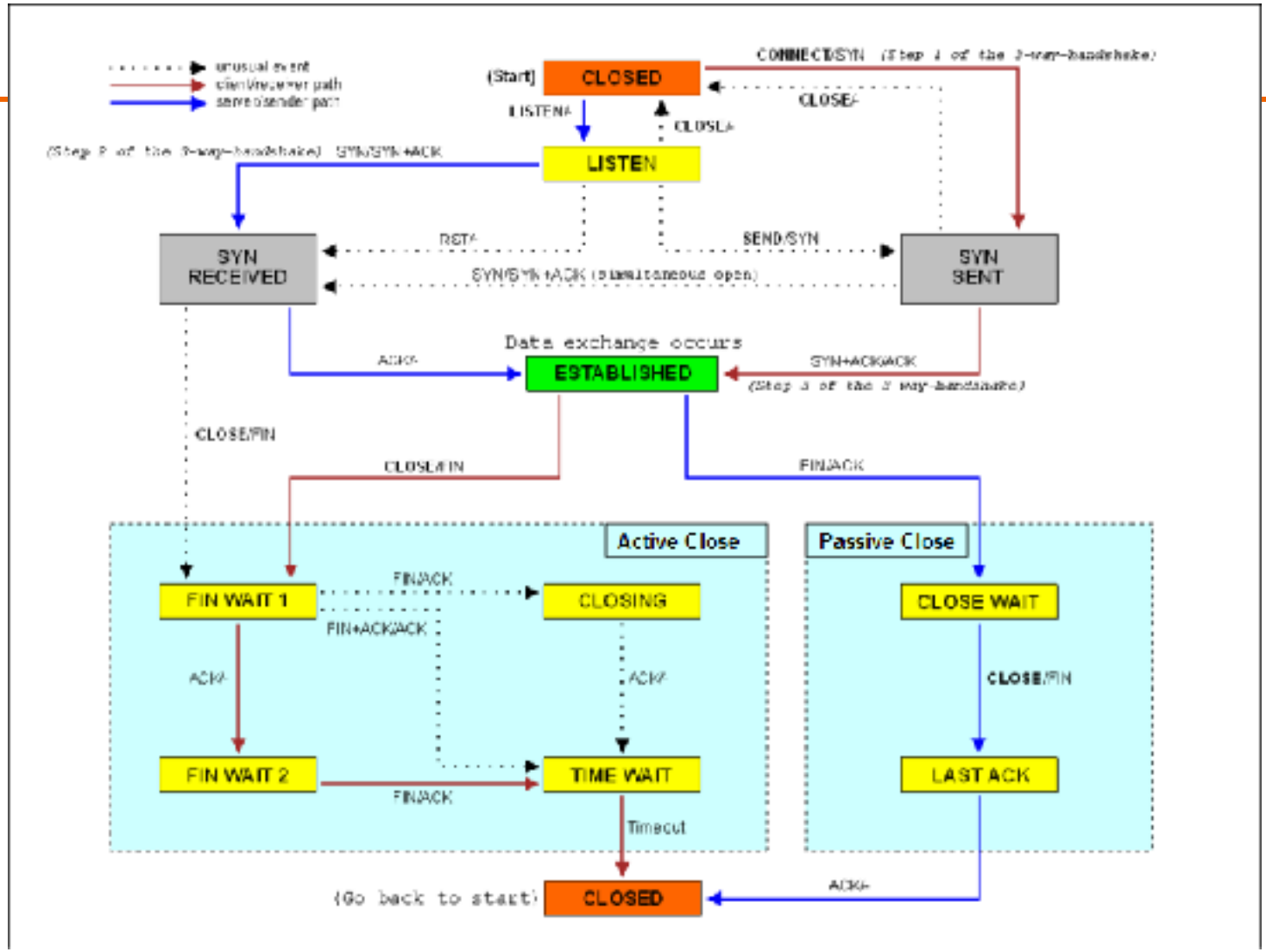
The Environment

- ~12k hardware clients
- ~4k “software” clients (users)
- ~6k qps
- ~3k prepared statement calls/sec
- ~500 transactions/sec
- ~400 tcp open/sec
- 3:rd party application, written in PHP, running on seven web servers



PERCONA

TCP States



PERCONA

TCP States

- After **Closing** they stay in **TIME_WAIT** state.
- **TIME_WAIT** timeout is 1-2 minutes (the connections stay in that state.)
- cat /proc/sys/net/ipv4/ip_local_port_range - 32768 60999
  - That is around ~ 28231 local ports. It sounds a lot
- Let's say every connections stay there 1 minute.
  - 28231 / 60 ~ **470** , you will run out local ports if you open more than 470 connections per second.
- When you have 12k hardware clients sending data in every seconds that is not that lot.
  - PHP Warning: mysqli::mysqli(): (HY000/2002): Cannot assign requested address in /root/phptest/test_credentials.php on line 33

The problem(s)

- No connection pooling in PHP, leading to
  - Firewall port exhaustion
  - Source port exhaustion on web-servers
  - High number of tcp-connection requests
  - High threadpool activity
- Ineffective use of prepared statements
  - 1 prepare/bind/execute/close
  - small number of unique statements

The problem(s)

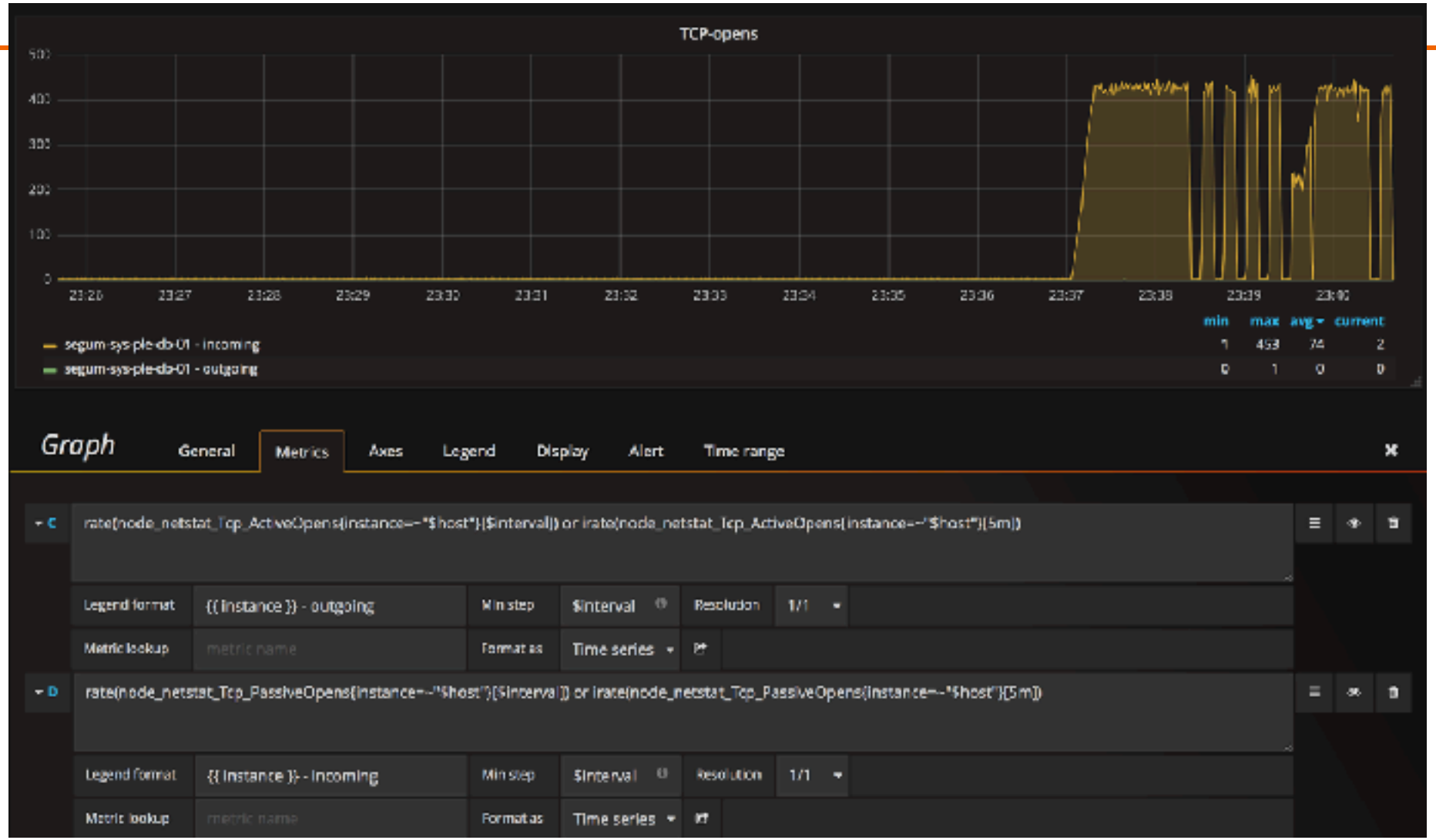
- “Closed source” - proprietary PHP
- Each page call creates one to many database sessions
- Needs to ‘co-exist’ with other critical application servers - no way of setting up an isolated environment for this application
- Business demands 24/7 uptime
- Hardware clients sends status updates only once - no resends

How can we montior that?

We can use PMM

- The metrics are collected.
- But they are not visualized.
- We need a custom dashboards for TCP connections.
  - rate(node_netstat_Tcp_ActiveOpens{instance=~"$host"}[$interval]) or irate(node_netstat_Tcp_ActiveOpens{instance=~"$host"}[5m])

We can use PMM



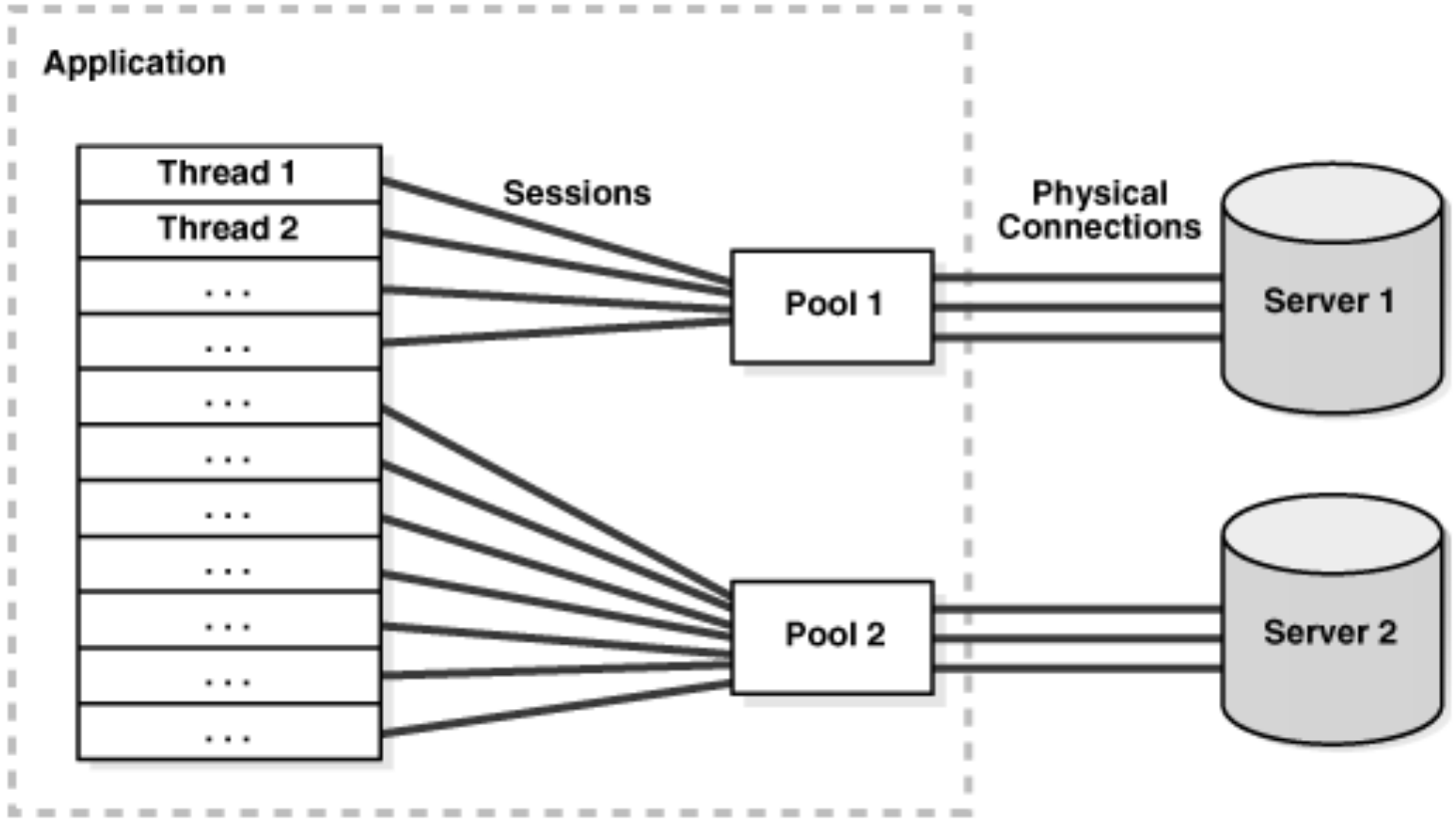
PERCONA

What is connection pooling?

Connection Pooling

- Opening and closing a connection for each request is costly and wastes resources.
- Connection pooling is a collection of the connections.
- The application can reuse the same connection again.
- This reduces the overhead associated with connecting to the database to service individual requests.

Connection Pooling



Connection Pooling

- Java has connection pooling.
- Python has connection pooling.
- But PHP does not have a proper connection pooling.
  - Without connection pooling the chance to get contentions issues is significantly higher. It does not matter what kind of application do you use.

Possible solutions

Possible Solutions

- rewrite PHP/port application
  - supplier unwilling to take the development cost
  - in-house development team unwilling to take maintenance responsibility
  - operations team unwilling to maintain a separate “branch” for patching
- ip-tables
  - NAT complexifies the environment
  - requires different configuration across portals
  - adds to firewall strain
- Tune kernel parameters
- Unix Sockets

Possible Solutions - Unix Sockets

- Unix domain socket or IPC socket (inter-process communication socket).
- Exchanging data between processes executing on the same host operating system.
- Rather than using an underlying network protocol, all communication occurs entirely within the operating system kernel.
- Uses the file system as their address name space.
- Two processes can communicate by opening the same socket.

How ProxySQL can help us?

ProxySQL Setup

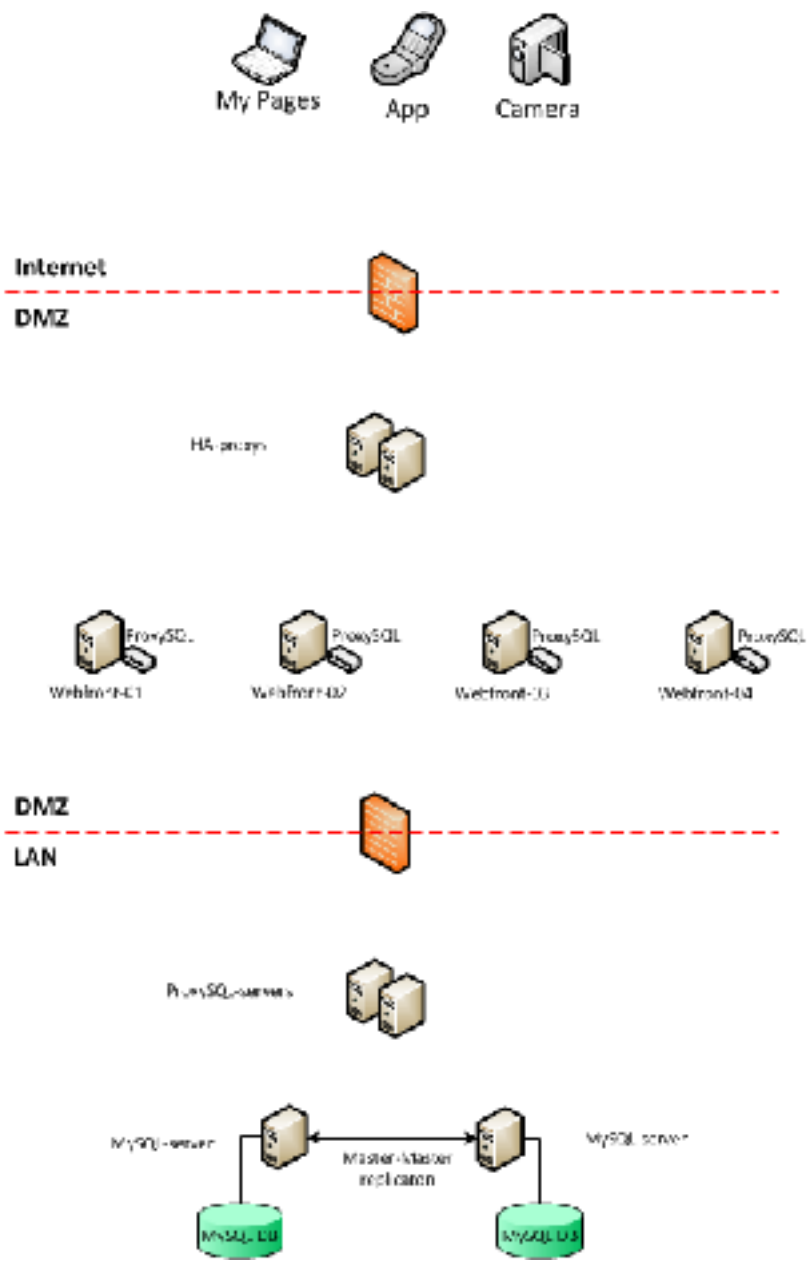
- Running locally on all application server.
- Listening on socket at /tmp/proxysql.sock
- The application connects through socket to ProxySQL.

Relevant features

- seamless integration
  - reconfigure all portals to connect to unix-socket
  - proxySQL “proxying” connections from unix-socket to backend DB
- connection pooling
  - application closes connection to proxySQL, not to DB
  - proxySQL reuses db-connections for the next connect
- prepared statement reuse
  - proxySQL filters out “statement close”/”prepare statement” when possible

The new environment

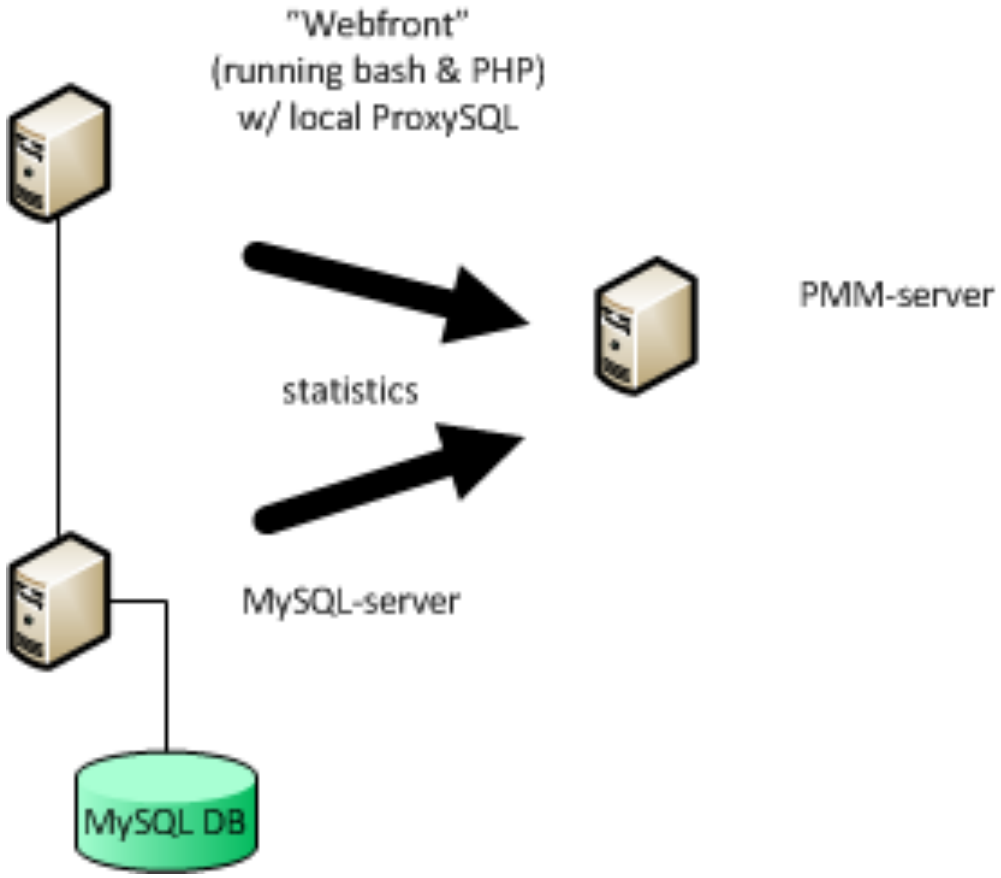
- ~12k hardware clients
- ~4k “software” clients (users)
- ~6k qps
- ~3k prepared statement calls/sec
  - ~3k “execute statement”/sec
  - ~50 “prepare statement”/sec
- ~500 transactions/sec
- **~10 tcp open/sec**
- 3:rd party application, written in PHP, running on seven web servers, with locally installed proxySQL
- 1 central proxySQL for backend switch



PERCONA

Demo Time

The test environment



Q & A

Thanks for your attention!

Champions of Unbiased Open Source Database Solutions