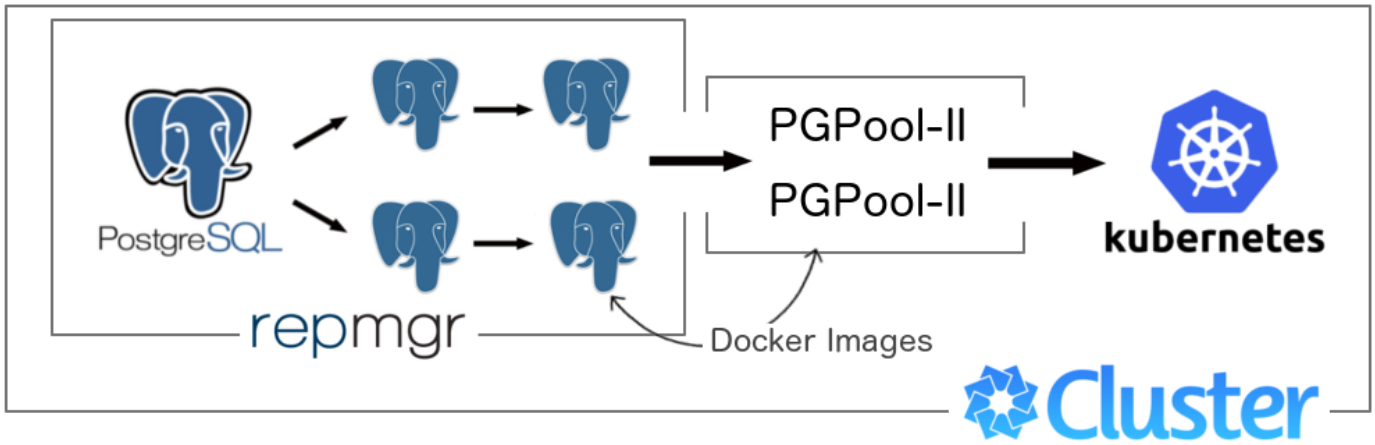
Forming a PostgreSQL cluster within Kubernetes

Dmitriy Paunin Follow
Aug 1, 2016 · 9 min read



When you work on a database management system, trust and hope is what matters the most. Well, really, this is what a database is! You've got to trust it to store your data, and hope it won't just fail one day. The title conveys the whole point — a place where data is kept, the main task is to STORE. And the saddest thing, as always, once these beliefs crash into the ruins of a downed production database.

"So how do you avoid it?" , you ask? "By not deploying anything on the servers", I answer. Nothing unless it can recover itself, at least

temporarily but reliably and quickly.

In this article, I will try to talk about my experience of configuring a built-to-last-a-lifetime Postgresql cluster within another failover solution from Google — Kubernetes (aka k8s).

Task

Almost any application needs to have a data store. This is a necessity. Making this data store resistant to faults in a network or on physical servers is common courtesy of a good system architect. Another aspect is high availability of a service, even with a number of heavy competing service requests, which means easy scalability upon necessity.

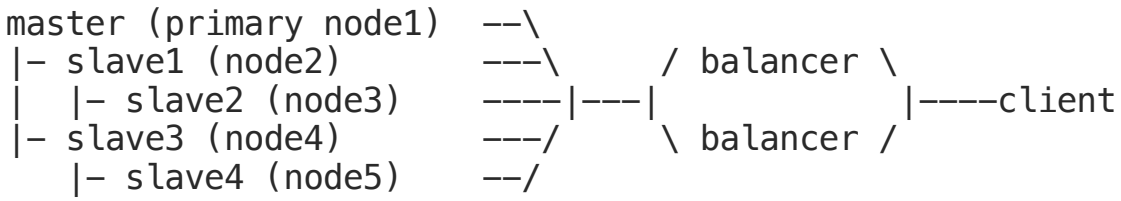
So here are our problems to solve:

- **Physically** distributed service
- **Load balancing**
- Unlimited **scaling** by adding new nodes
- **Automatic recovery** from failures, deleting, and loss of connection between the nodes
- **Design with no single point of failure**

Additional points dictated by the religious beliefs of the author:

- **Postgres** (the most academic and consistent solution for RDBMS among the free available tools)
- **Docker** wrapper
- **Kubernetes** infrastructure description

In the diagram, it will look like this:



Assume the input data:

- Increased number of read requests (in relation to a database record)

- Linear load growth up to 2x of the average

Googled solution

Having had some experience in solving IT problems, I decided to ask the collective mind of Google, “postgres cluster kubernetes” — what I got was a bunch of information junk, then I tried “postgres cluster docker” — not much better, and, finally, “postgres cluster”, which showed up a few options I have chosen to work with.

What got me upset is the lack of decent Docker images along with descriptions of any options for clustering, not to mention Kubernetes. By the way, there weren’t many options for Mysql either, but still one of them is worth mentioning. Like this example in the official k8s

repository for Galera(Mysql cluster)

Google has made it clear that I have to look for a solution myself and more than that, do it manually (sigh, at least I have a bunch of scattered articles to browse through).

Bad and unacceptable solutions

Let me make it clear, all the points in this paragraph are based on my subjective opinion and therefore might seem quite viable to you. However, based on my experience, I had to cut them off.

Pgpool. Why Pgpool is not always good?

Whenever somebody proposes a universal solution (for anything), I

always think that such things are cumbersome and hard to maintain. Well, same with Pgpool, which hasn't become an exception because it can do almost everything:

- **Load balancing**
- **Keeping a bunch of connections** to optimize database connectivity and access speed
- Support for **various replication options** (stream, slony)
- **Auto-determination of Primary server** for recording, which is important in reorganization of roles in a cluster
- Support for failover/failback
- Proprietary master-master replication
- Coordinated work of multiple Pgpool nodes to eliminate single point of failure

I only found the first four points useful. Speaking of the remaining three, these are the problems they present:

- Recovery with the help of Pgpool2 offers no decision-making system for choosing the next master — the entire logic has to be described in the failover/failback commands
- With the master-master replication, write time doubles regardless of the number of nodes. Well, at least it doesn't grow linearly
- How to build a cascading cluster (when a slave is reading from the

previous slave) is a total riddle

- Of course it's cool that Pgpool connects the neighboring nodes and can quickly become the master node in case a neighboring one has failed, but for me this problem is perfectly solved by Kubernetes, which ensures the same behavior for literally any service installed on it

Slony. And the elephants go away...

Actually, having read and compared this (www.slony.info) with Streaming Replication, which is already well-known and runs out-of-the-box, the decision to not even think about the elephants came easily.

On top of that, on the very first page of the website the guys state that "PostgreSQL 9.0 includes streaming replication, which, for a number of use cases, is likely to be simpler and more convenient than Slony-I", going on to say, "There are, however, three characteristic kinds of cases where you'll need something like Slony-I", which are the following:

- partial replication
- interaction with other systems (like Londiste and Bucardo)
- additional behavior when replicating

To put it simply, I don't think Slony is good enough, unless you need to take care of those three specific tasks.

Master-master replication. Not all replications are good enough

After sorting out the options for ideal bidirectional replication, it appeared to me that the efforts it takes to implement such replication are simply incompatible with crucial functions of some applications. Not to mention the speed, there is a limitation in handling transactions and complex queries (SELECT FOR UPDATE, etc.).

Perhaps I'm not as versed in this matter, but the drawbacks of the solution that I've seen happened to be enough to leave the idea. And yet, after putting my thinking-caps on I figured out that a system with enhanced write operation will need a completely different technology rather than relational databases.

Consolidating and assembling the solution

In the examples below you will see what a solution should ideally look like, while the code will show my own implementation. To create a cluster, you don't necessarily have to have Kubernetes (there's a docker-compose example) or Docker at all. Just then all this post will be useful not as a CPM solution (copy-paste-modify) but as an installation manual with snippets.

Primary and standby instead of master and slave

So why did colleagues from Postgresql refuse to use terms "master" and "slave"? Hmm, I could be wrong but there was a rumor that it's because of non-Political Correctness, we all know that slavery is the worst thing that ever happened. Well, right.

The first thing you need to do is start the primary server, followed by the first standby layer, then the second standby layer — all according to the task at hand. Hence we get a simple procedure for the start of a typical Postgresql server in the primary/standby mode with the configuration to enable Streaming Replication.

What you should pay attention to in the configuration file

```
wal_level = hot_standby
max_wal_senders = 5
wal_keep_segments = 5001
hot_standby = on
```

All parameters have short description in comments, but all you need to

know is that with this configuration we enable access to read WAL logs for clients and allow execute queries during recovery. You can find interesting article about it on Postgresql Wiki.

As soon as the primary server of the cluster starts up, we move on to start the standby one, which “knows” where its primary is located.

My goal here has boiled down to the assembly of a universal Docker Image, which would work differently depending on the mode, namely the following way:

For primary:

- Configures Repmgr (more about it later)
- Creates a database and a user for the application
- Creates a database and a user for monitoring and replication support
- Updates the config (postgresql.conf) and provides access to external users(pg_hba.conf)
- Runs the Postgresql service in the background
- Registers as master in Repmgr
- Starts repmgrd — a Repmgr demon for monitoring the replication (about it later, too)

For standby:

- Clones the primary server via Repmgr (by the way, with all configs as it simply copies the $PGDATAdirectory)
- Configures Repmgr
- Runs the Postgresql service in the background. After cloning, the standby server identifies itself as a standby and starts to obediently follow the primary server
- Registers as a slave in Repmgr
- Starts repmgrd

Sequence is of utmost importance when performing all these operations, thats why dockerize project helps a lot here.

The difference between the first and the second standby layers is that ANY server from the first layer can act as master for a second layer server. Do not forget that the second layer should start after the first with a time delay.

Split-brain and selection of a new leader in the cluster

Split brain — a situation in which different segments of a cluster can create/elect a new master and think that the problem is solved.

This is one but not the only problem that Repmgr helped me to solve.

Essentially, this is a manager that can:

- **Clone master** (in Repmgr terms) and automatically configure a newborn slave
- Help **recover** the cluster if master dies
- **Elect a new master** and reconfigure all slave services to follow the new leader, automatically or manually
- **Remove nodes from a cluster**
- **Monitor the health of a cluster**
- **Execute commands** on events in a cluster

What comes to the rescue in our case is repmgrd, which gets started by the primary process to monitor the integrity of the cluster. In situations when you have lost access to master, Repmgr attempts to parse the current structure of the cluster and determine which will be the next

master. Certainly, Repmgr is intelligent enough to elect the only correct master and avoid creating a Split Brain situation.

Pgpool-II — swiming pool of connections

The last component of the system is Pgpool. As I stated in the section about bad decisions, the service still does its job:

- **Load-balances** between all nodes in a cluster
- **Stores connect descriptors** to optimize database access speed
- In this Streaming Replication case, PgPool-II automatically locates master and uses it for write requests

As a conclusion, I have created quite a simple Docker Image, which at startup configures itself to work with a set of nodes and users who will be able to be authorized through MD5 authorization via Pgpool (this bit happened to be a bit more complicated than I thought).

Far too often you need to get rid of a single point of failure to suffice, and in our situation this point is a pgpool service that is proxying all requests and thus can become the weakest chain in the path to data access. Fortunately, k8s comes handy allowing us to create as many replications of the service as needed.

Sadly that this Kubernetes example is missing the part about creating replications, but if you are aware of how Replication Controller and/or Deployment works, it'll be no trouble to pull the configuration off.

Result

This article is not a retelling of the scripts to solve the problem, rather, it is a description of the structure of the solution. The former means that to better understand and optimize the solution, you will have to read the code, at least README.md on github, which is step-by-step and meticulously explains how to start up the cluster for docker-compose and Kubernetes. In addition, for those who will appreciate what I wrote and decide to move beyond the proposed solution, I am willing to lend a virtual helping hand.

- Sources and documentation on Github
- Docker Image for cluster-ready Postgresql from the example
- Docker Image for Pgpool with configuration through ENV variables

Documentation and used material

- Streaming replication in postgres
- Repmgr
- Pgpool2
- Kubernetes

PS:

I hope you have found the article helpful. Have a positive summer

everyone! Good luck and good mood, colleagues ;)

PPS:

The original article was written in Russian, so thanks to Sergey Rodin (Technical Writer, Lazada)and Tom Wickings (Technical Project Manager, Infrastructure Department, Lazada) for help to translate it!

Cloud Computing Kubernetes Postgres Database Big Data