The Bones of High Availability

Author: Shaun M. Thomas
Date: September 18th, 2015
Venue: Postgres Open 2015

Your Presenter

Shaun M. Thomas <sthomas@peak6.com>
Database Architect at Peak6 Investments

I apologize in advance.

What is High Availability

$$$$$$$$$$$$$$$$$$$$$$$$$$ $$$$$$$$$$$$$$$$$$$$$$$$$$ $$$$$$$$$$$$$$$$$$$$$$$$$$ $$$$$$$$$$$$$$$$$$$$$$$$$$ $$$$$$$$$$$$$$$$$$$$$$$$$$

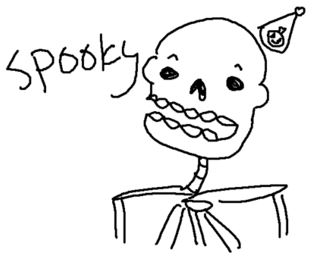
What is High Availability

(But seriously)

- Staying online in the face of adverse conditions.
- The result of a downtime cost/benefit analysis.
- Doing this can get expensive.

What are Bones?

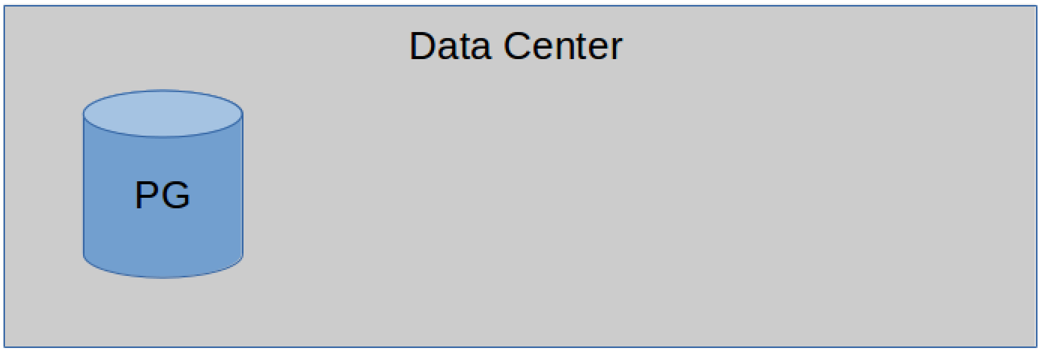
- Fun fact: you contain a spooky skeleton.



- Let's make one for Postgres!

One is the Loneliest Number

Let's start with one node.



Poor Postgres

Just look at him...

Eggs in One Basket

- You've probably seen one of these...
- Good for development.
- Don't use this for anything important.

- Backups happen here, too.
- Crash = SOL
- Story time!

Never, ever depend on a single node for anything, anywhere, ever. If that node ever dies, that's the end of the story, possibly for the entire company.

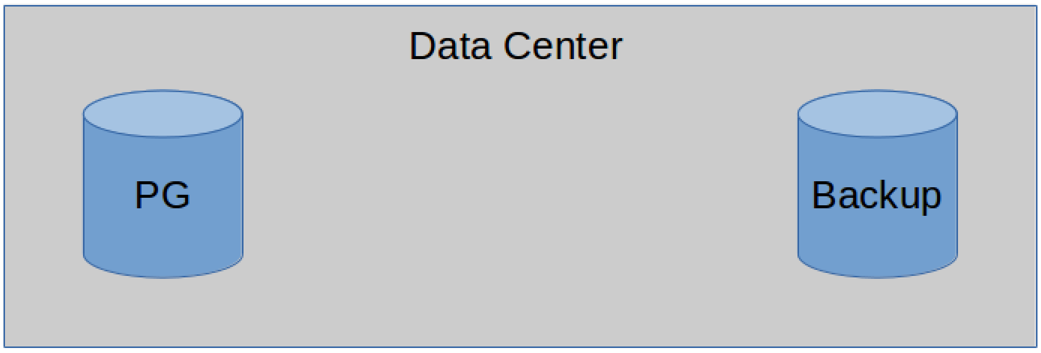
One of my first large Postgres projects in 2005 was when I was newly hired at a small company. They didn't have a Postgres DBA and as a consequence, didn't know the full implications of the `max_fsm_pages` setting. Though that setting has since been deprecated, the default was only 200,000 (at most) back then. That's only enough for 200,000 modified or deleted rows over the entire database between vacuums!

As a consequence, their database slowly bloated over the course of a year, and was less than a week away from exhausting the disk on its server. An old, unexpandable server that was already several years old. There wasn't enough space anywhere to save the data, and it was the data store for all of their websites. I wrote a script that performed a `VACUUM FULL` on every table in the database, from smallest object to largest, to avoid exhausting remaining disk space.

The process could only run for two hours very late at night, requiring several iterations before the entire system finally reached its optimal size. This required multiple maintenance windows over the course of the week, but the alternative was much worse. If there was even a second server, this could have been avoided.

Two Can Be as Bad as One

Let's move the backups and WAL archives to prevent disaster.



But backups need to be restored...

Keep in mind that this only prevents us from losing the backups. If the database node goes down, we still have to restore the data somewhere and get it running again. This is better than only having the database node, but just barely.

Learn About ABE

No, not *that* Abe!

ABE: Always Backup Elsewhere

1. Backups must exist.
2. Assume the database node can die.

3. Use a shared or dedicated backup server.
4. Goto 1.

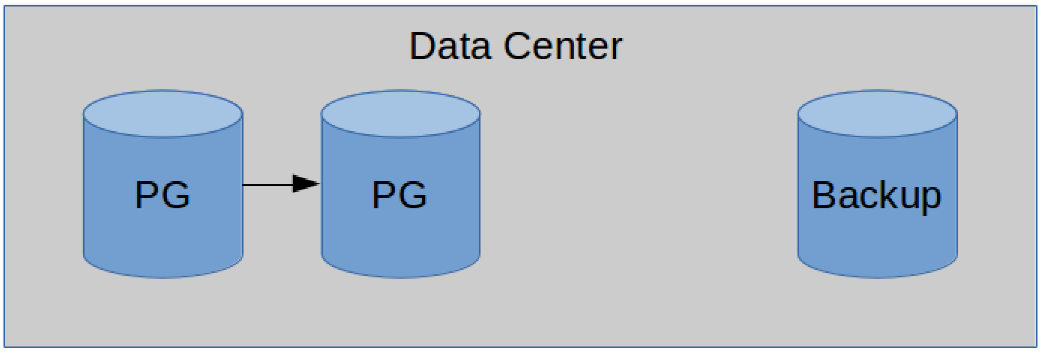
If the machine where the database runs is the only location where backups are held, there are no backups. Local backups are fine for PITR only, in the case the instance fails for some reason.

Since it's very unlikely the database server is the only server in the data center, use one of the other available systems to store Postgres backup files. This can be a shared system where multiple services transmit backup files before it gets captured to tape.

No matter the situation, follow the ABE principle.

Three Cheers for... ?

Staying online!



Three Amigos

- Our first viable HA cluster!
- Many options
  - Built-in replication
  - Slony, Bucardo, Londiste
  - DRBD
  - Pacemaker
  - etcd + HAProxy
- Same or better hardware!

This is the first point where we actually have something we can genuinely call a highly-available database. If one node is ever damaged or in need of maintenance, we can always use the other.

To reach this point, we can use the extremely comprehensive replication included with Postgres, a higher-level trigger-based logical process, or some kind of underlying block-level method. The first of these is very easy to establish, while the second usually requires a bit of experience and is best for select tables.

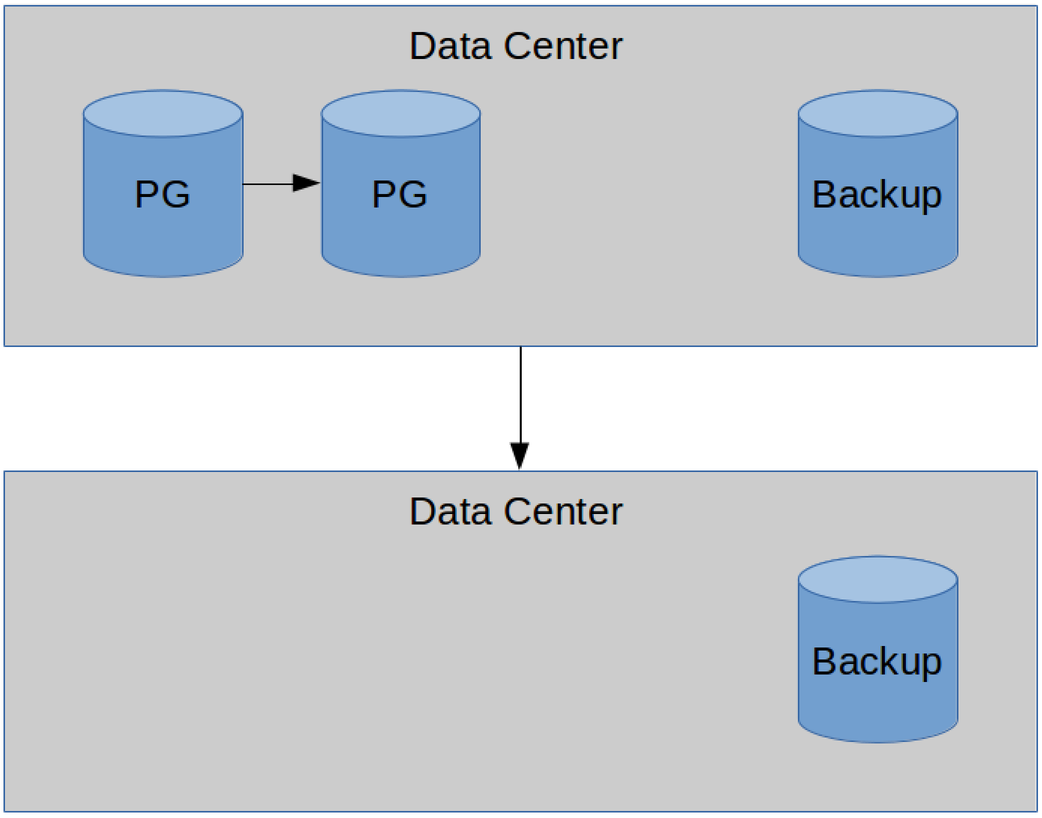
The third option is where things get more complicated, but it also gives us the capability to circumvent Postgres entirely. If we use something like DRBD, the two nodes involved in the cluster will be byte-for-byte identical at all times, with much less transactional overhead and associated latency.

Regardless of the synchronization method we use between these two servers, we'll want some way to automate switching between them. One popular method relies on a Pacemaker stack. Others might prefer

This is the bare minimum for a HA cluster: two nodes for the active database, and one node for backups. And keep in mind that the secondary server has to be *at least* the same specification as the primary, or we risk problems after failing over. If it's a better server, we can use it for in-place upgrades, assuming the primary gets a similar bump in capabilities after the maintenance is over.

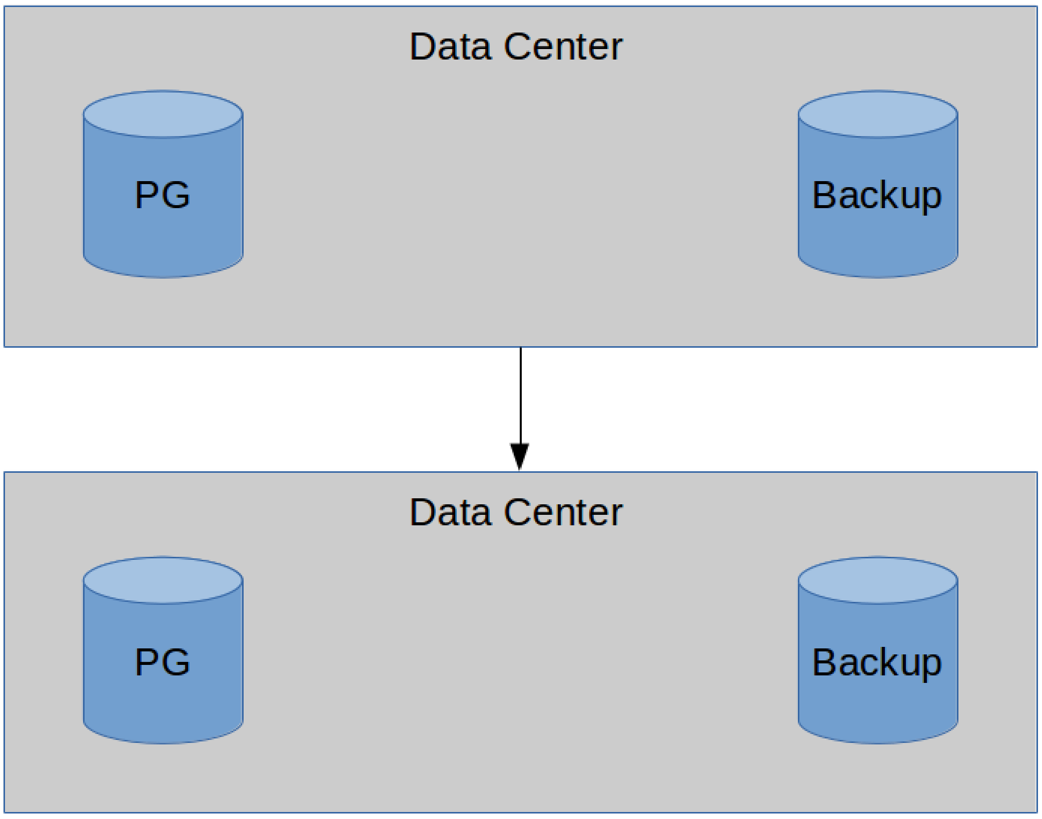
One for the Road

Backups... are safer elsewhere.



Two for the Road?

Offsite replicas are also nice.



But why?

Moving backups and replicas off-site protects the database from catastrophic data loss in the event the data center itself suffers some kind of damage or becomes unreachable. Here are a few fun examples!

Backhoes!

The #1 predator of The Internet!

It's an inside joke, but there's a kernel of truth here. Backups should be available from more than one location in case the connection to one gets... interrupted.

Floods!

Was your data center in Iowa?

I know *my* servers always run better when immersed in the Mississippi river.

To be serious for a moment, this can't be overstated. In the case of a natural disaster such as a flood, servers and data at a single location might be a total loss. If data is located externally, we can eventually bring the system up again.

Without this, a site outage may become permanent.

Shit Happens

Remember Murphy's Law.

This really should set the tone for the rest of this talk. You can't be taken by surprise if you're always expecting something to fail.

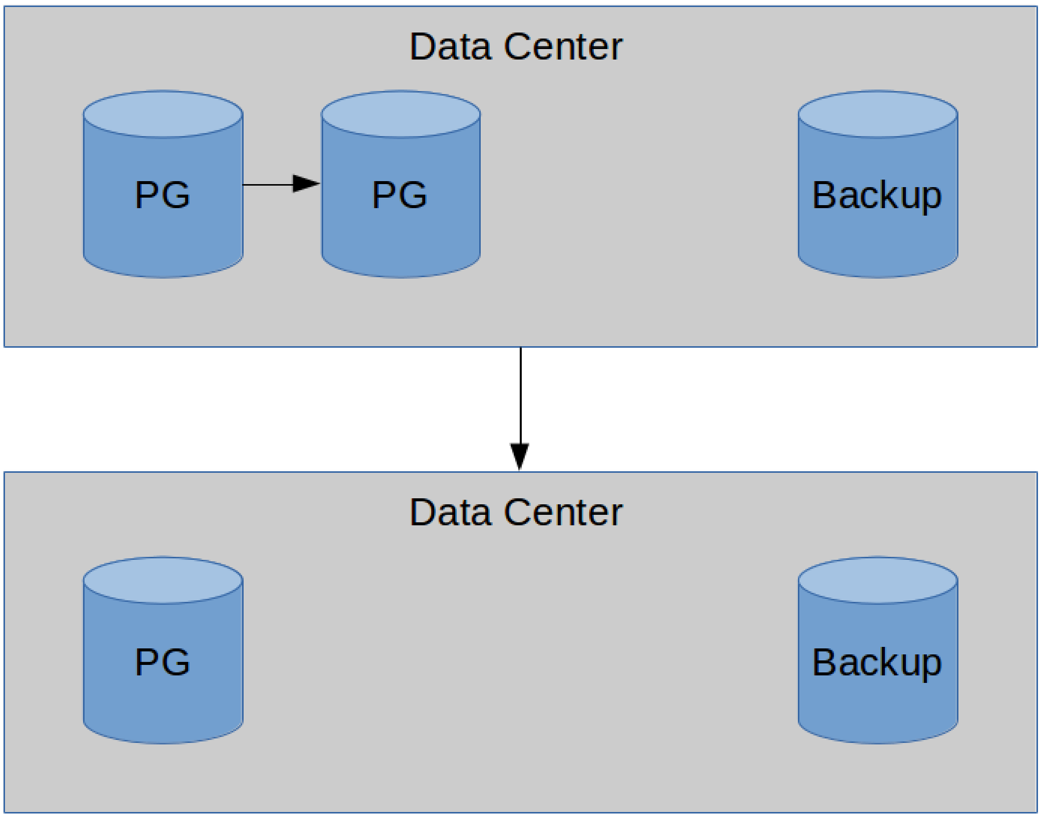
Also, that guy is *way* too happy about ruining our day.

What's Missing

... No.

Five Golden ... Servers!

Keep Postgres running everywhere!



This is your foundation.

Having an offsite backup makes it possible to restore that backup to another server, and make that server an active member of our availability architecture. Depending on the utilization of that other data center, this may or may not be a lot more expensive than keeping a simple backup server at that location. But if we want to stay up in the case of a data center outage, it's a requirement.

True Disaster Recovery

- Have an off-site Postgres server.
  - Preferably the same hardware
  - Warm/hot standby
  - Lots of data for WAL stream
- Things are starting to get expensive.
- And complicated...
  - Promote
  - DNS
  - Etc.

Earlier, we needed three servers to claim we were highly available. Now, we need at least four to have a Disaster Recovery setup. Now, things are starting to get serious. Simply using another data center can be expensive, but now we're also allocating a running---but mostly idle---server. In many cases, this extra server can be a cut-down version of our main setup. If we're running in our alternative location, we probably have more to worry about than a few slow queries.

But it's still an expense to consider. Getting the data there can also be a problem, depending on how much volume comes through our primary server. Using warm or hot standby, we will still have WAL traffic to contend with. If we're depending on WAL files to catch up in the case of a long connection interruption, that's traffic for *both* streams. So now we have bandwidth costs to go along with the server space and power delivery.

Beyond that, we have to direct traffic to the new server in case of emergency. That means reconfiguring and restarting applications, connection pools, name resolution, or whatever is necessary to activate the remote systems. In a perfect world, all of this is automated and well tested, but we all know the reality of that scenario.

Also, five servers is a lot.

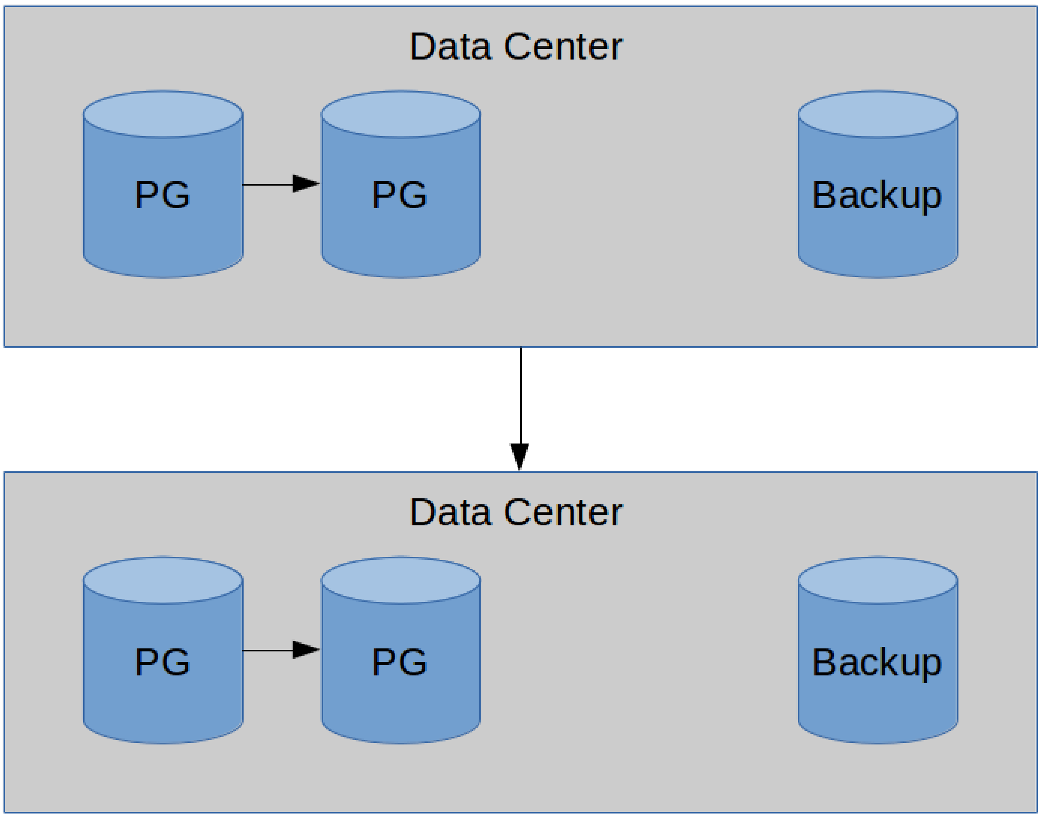
Fifth Wheel

Remember this guy?

Now that we have a fifth server that's acting as emergency outage insurance, we have a similar situation as we started with: one very sad server.

Failure is Always an Option

How much do we depend on our DR setup?



Sometimes the answer is "a lot."

Cast a Hex

- If we expect to need DR long-term.
    - Moving the primary data center.
    - Persistent primary outage.
- Retain old hardware for ICE uses.

When do we need that sixth server? This is getting a bit ridiculous, isn't it? We already had one mostly idle server in the DR environment, and now we have two? Why?

It could be seen as a stretch, but it's not unknown to move servers from one location to another. While doing so, however long that process takes, we need a secondary set of servers that can fill in at the same capacity as the servers we're moving. In the same basic configuration. Otherwise, we run the risk of a hardware failure in the DR environment while we're operating on limited resources. And there's always the risk of natural disaster. If we're on the East coast, and a hurricane eliminates our primary data center, we could be running on DR systems for months. How avidly do we want our website to remain online?

Beyond that, it's possible the DR hardware is a hand-me-down from previous upgrades of the primary servers. In that case, retaining the original pair for DR use is simply a matter of convenience. Whatever the reason, think through potential failure scenarios, and have a working checklist of elements necessary to weather outages of various description. If one of those situations calls for another server in the DR environment, it's better to have one than need one.

All's Well that Ends Well

Hooray!

We can dream, anyway.

You're Kidding Me

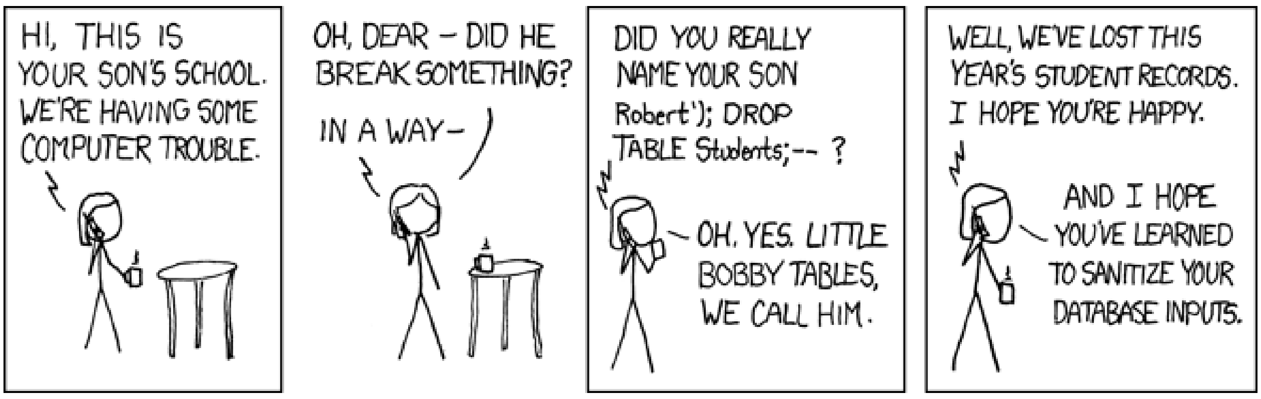
Hi!

I hate you!

Sigh.

Everything is Synchronized

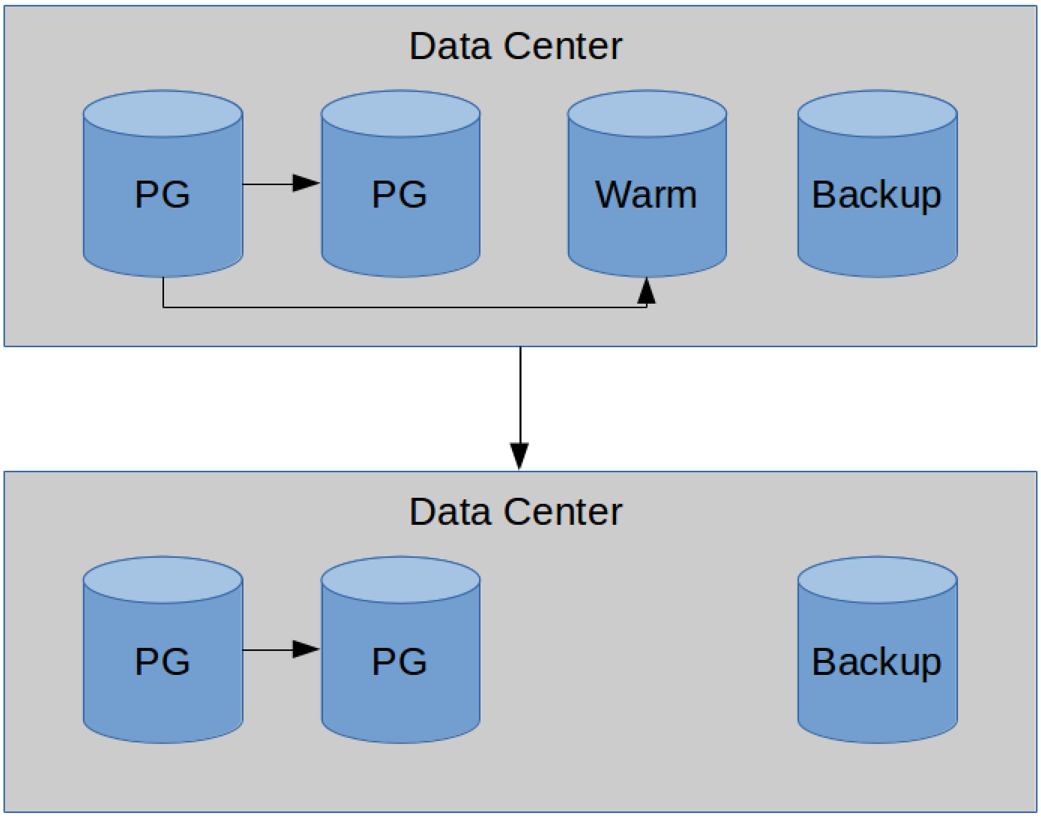
Think about that for a second.


HI, THIS IS YOUR SON'S SCHOOL. WE'RE HAVING SOME COMPUTER TROUBLE.
OH, DEAR – DID HE BREAK SOMETHING?
IN A WAY–
DID YOU REALLY NAME YOUR SON Robert'); DROP TABLE Students;-- ?
OH. YES. LITTLE BOBBY TABLES, WE CALL HIM.
WELL, WE'VE LOST THIS YEAR'S STUDENT RECORDS. I HOPE YOU'RE HAPPY.
AND I HOPE YOU'VE LEARNED TO SANITIZE YOUR DATABASE INPUTS.

We have four servers that all have the same data within a few milliseconds of each other. What could go wrong?!

When the Worst Happens

Have a delayed replica!



Restoring backups is slow. If bad data, a corrupt table, or an unintended *DELETE* statement hit the main server, it'll be faithfully replicated everywhere. We **need** an answer for that.

Have a Backup Plan

- recovery_min_apply_delay (9.4+)
- Goes well with (9.1+):
    - pg_xlog_replay_pause()
    - pg_xlog_replay_resume()
- Previously, this was fairly convoluted.

The *recovery_min_apply_delay* setting allows us to insert a delay into the standard Postgres replication stream that prevents immediate replay. This gives us the ability to do quite a few things:

- Obtain data that was accidentally deleted.
- Preserve data that was corrupted.
- Switch to PITR and roll forward to just before hardware corruption.

Really, the list goes on. In any scenario where we need untainted data for any reason, a time-delayed server can fulfill that requirement. Provided, of course, the chosen delay is sufficient for emergency escalation and response times. An hour or two is usually enough, but always consider how quickly automated and manual checks take to recognize various issues.

Once a problem is recognized, the immediate reaction should be to pause replication on this delayed server with *pg_xlog_replay_pause*. From there, the server can be treated as any normal standby system without the risk of it being affected by whatever calamity befell the other systems. This gives us time to react, and in emergencies, that's a rare commodity indeed.

Before the *recovery_min_apply_delay* setting, the only alternative was to keep a server separated from the main replication systems, and rely on WAL-file transmission. It's pretty easy to script a *find* command that maintains a specific time interval for relevant WAL files, but this is a lot more reliable.

And Be Paranoid

- New Postgres 9.5 feature:
  - archive_mode = always

Depending on how paranoid you are, there's also the option of saving WAL archives on every server involved in a replication stream. This is especially helpful because it removes *archive_command* from the chain of custody when preserving WAL files remotely. We no longer have to worry if it's doing the job properly.

If you weren't at PGCon this year, we had a very sobering conversation about this.

Or Suffer The Consequences

Imagine this:

- Everything in the 6-server stack!
- Hardware upgrade! Yay!
- DRBD bug causes subtle data corruption.
- For an hour.
- Corruption is faithfully consumed by secondary.
- And DR.
- And the DR secondary.
- Oh...

Use the Tools Provided

- Data checksums
  - Only available during initdb.
  - Needs dump/restore for existing data.
- pg_rewind (9.5+)
  - Replica primary switching is now built-in.

More Information

- High Availability, Load Balancing, and Replication
- Replication
- recovery.conf
- DRBD
- Slony-I
- Bucardo

- Londiste
- Pacemaker
- repmgr
- etcd
- HAProxy
- recovery_min_apply_delay
- --data-checksums

Questions