Managing PostgreSQL backup and replication for very large databases

By Flavio Gurgel[1] *(Database Administrator)*

We're the 5th most accessed website in France; to serve well our users we have several monolithic applications as well as microservices, and all of them have to be backed by a reliable database system which is PostgreSQL.

That statement gives us the credentials to say "*Yes, PostgreSQL is web scale*". And being web scale means that we have to handle several terabytes of data reliably, from several database instances.

A good amount of the reliability comes from the high quality PostgreSQL code as well as carefully chosen hardware details and extensive use of replication. Those variables are really important for database availability, but one must never forget that incidents that cause data loss do happen, and even the most carefully designed database infrastructure is not safe from:

- **bad handling**—mistakes on command line happen even for the most experienced ones
- **application bugs**—this happens more than you think
- **bad user input**—you can sanitize this in your code but you'll never think about everything a user can do
- **attacks from the web**—do you think you're 100% safe? Think again
- **losing all servers (master and all standby)**—this is very rare, even rarer if you have geographically separated servers as we do, but it's possible

Replication by itself can't protect you from the points above. Some may use delayed replicas but you can lose the time window to react since it's finite. There's no other way, databases need backups. And the good ones like PostgreSQL are designed for it and give you all the features you need.

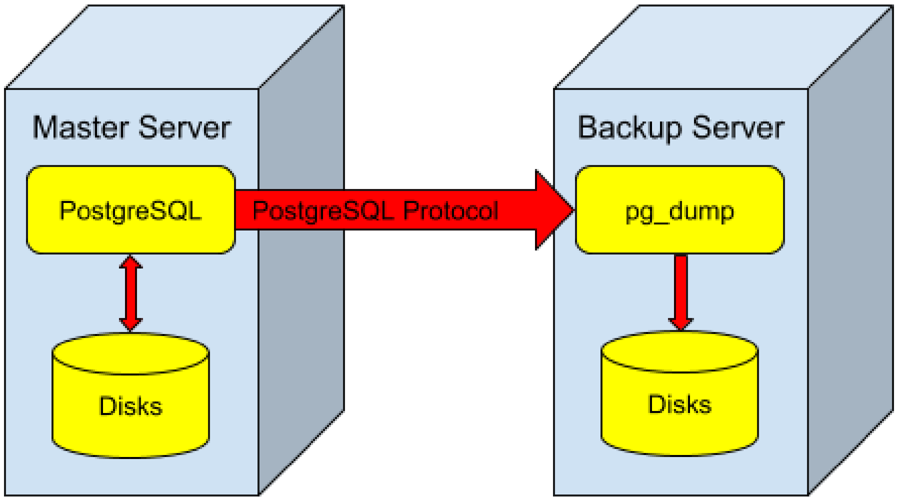
In PostgreSQL, the backup is also the starting point for the physical replication and that's why we're talking about it in this same post.

PostgreSQL backup strategies

There are two ways of doing PostgreSQL backups: **dumps** and **PITR**.

Dumps—the pg_dump and pg_dumpall tools

The *pg_dump* tool is used to dump a single database while the *pg_dumpall* can dump a database cluster—one instance with several databases in the PostgreSQL jargon. The dump is nothing more than a text script written in the SQL language, and can reconstruct a database from scratch.



An example of pg_dump usage—in this case, the pg_dump tool is installed in a dedicated backup server.

pg_dump is also capable of generating custom formatted files which are compressed to save disk space and can be used to parallel restore, which is way faster when you have capable hardware with fast disks and multiple core machines that are common these days. The custom format can be allied to the directory format and you can use parallelism to create the dump, and it's obviously faster than the original single-thread-single-process.

The parallel dump/restore feature is available in all currently supported versions of PostgreSQL, ranging from 9.3 to 10 at the time of this writing.

The custom format can be transformed in plain SQL text script via the *pg_restore* tool.

Other than safely backing up databases, dumps are useful in many other ways:

- **they're compact**—only data and object creation statements are present. Indexes take a lot of disk space and are reduced to a simple line in a dump
- **they can be compressed**—*gzip* is used under the hood and you can choose the compression level
- **they're portable**—originally, dumps were the only way to upgrade from a previous PostgreSQL major version to another. It still is a possibility but we have a faster way by using *pg_upgrade*, we'll talk about it in another post
- **they're portable (2)**—the file format, even the custom one, is platform agnostic and you can restore in any operating system supported by PostgreSQL (Linux, Windows, MacOS, BSDs, etc) or if PostgreSQL was compiled with different options like page size, wal segment sizes (this will be flexible in future versions anyway) or heap partition size
- **they're portable (3)**—with some customization and by using the INSERT mode, one can transform a PostgreSQL dump to be readable by another database system

- **they're flexible**—we can dump a cluster, a single database, a single table (or a list of tables), a schema—and also restore selectively when using the custom formats and the pg_restore tool
- they can be taken with normal database operations—no need to stop database traffic to create dumps
- **they're consistent**—PostgreSQL uses its MVCC and snapshot features so the dump is a frozen photography of the database at the time it started, with all constraints respected
- **they test your database**—to create a dump, PostgreSQL will scan all the tables with the corresponding toast tables (objects larger than 2kB in the default compile time settings) so if you have errors on data pages, PostgreSQL will let you know and you can take actions
- **they test your database (2)**—in case of inconsistencies of constraints on the origin database (happens rarely and are caused by PostgreSQL bugs, but has happened before) PostgreSQL will let you know when you try to restore the dump

Dumps are restricted in some ways too:

- creating a dump can put a lot of pressure on the system—especially if we use aggressive parallelism and we get out of CPU cycles and some queries may suffer or you have slow disks and the whole system will slow down to the point of timing out stuff up in your stack
- restore is usually slower than other strategies (as we'll see below)
- dumps are a consistent snapshot of the database and… that's it. The database is constantly evolving and if we need to restore several hours after the latest dump, we are going to lose all those hours of data changes

To the first point, you may say "*I create my dumps after hours during the night*" or "*weekends*". When you're a 24/7 website like **leboncoin**, you know that there's no "*after hours*" or "*weekend*" because all those hours are open to business.

To the second, well, restoring a database is rare so let's get along with it. And than you'll discover that the pressure to get online again can be high as hell. Every downtime second counts and you may discover it the worst way.

To the third, maybe you can retype all lost data. If it's your case, know that the other 99% of database administrators just can't.

Evidently, for some databases, dumps are just fine. One example could be a mostly static BI database that is fed once a day and may be rebuilt or updated from the latest data from the transactional system, where an outage of some hours can be "acceptable" since it won't impact the final customer immediately.

Some databases can even stay up with no backup at all, like temporary schemas used for intermediary calculations that are stored elsewhere.

But if you read this post to this point, you may be interested in

PITR—Point In Time Recovery

This is another backup feature included with PostgreSQL that is very powerful.

PITR is the capability of a database to be restored at **any point in time** with the limitation of when you created your oldest base backup.

PITR is a physical backup strategy, that is, we won't have a script or text file that represents our database, here we're going to physically copy the PostgreSQL database files from the live disk (or disks) to another one.

But, how is PostgreSQL capable to restore at a specific point in time? Actually this is quite simple because of the *ACID* characteristics of PostgreSQL, *D* is for "Durability" which means that once written to the database, a datum **has to** be written on disk. For PostgreSQL, that means that we write data to a sequential type of log file that we call a WAL—*Write Ahead Log.*

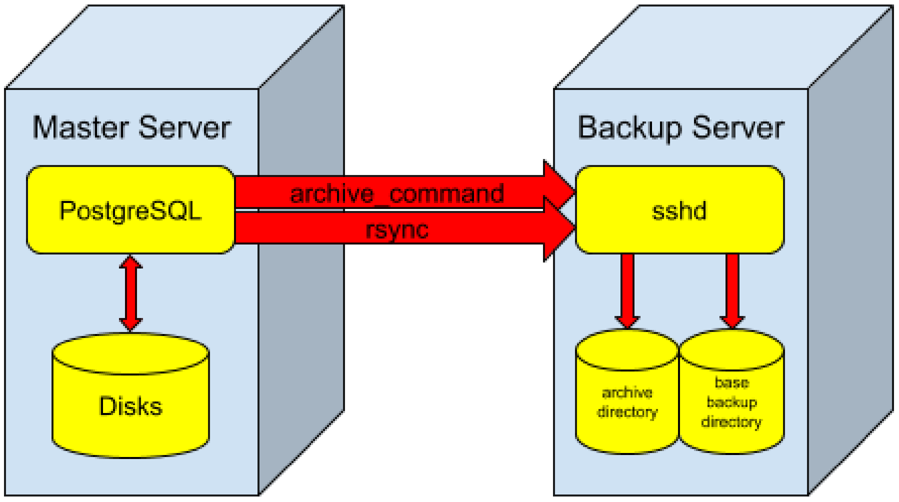
The sequential nature of *WAL* writing means that PostgreSQL writes data in an ordered way in those *WAL* files, and we have supplied mechanisms to pile up these files in another disk that we call **archive.**

PostgreSQL offers two ways of doing this:

- **by using the *archive_command***—one of PostgreSQL's configurations, where we put a chosen command or script to send *WAL* files to another disk with *cp*, another system with *scp* or *rsync*, to the cloud like with *aws-cli* or a mix of whatever we want with a home made script. Think of it as a "push", or better, "shipping" *WAL* files somewhere.
- **by using *pg_receivewal***—a client tool supplied by PostgreSQL that is capable to connect via streaming to the database server and receive *WAL* data as it is created by the database server, and save locally to a set of *WAL* files as above, but in a "pull" fashion or "consuming" from the database server.

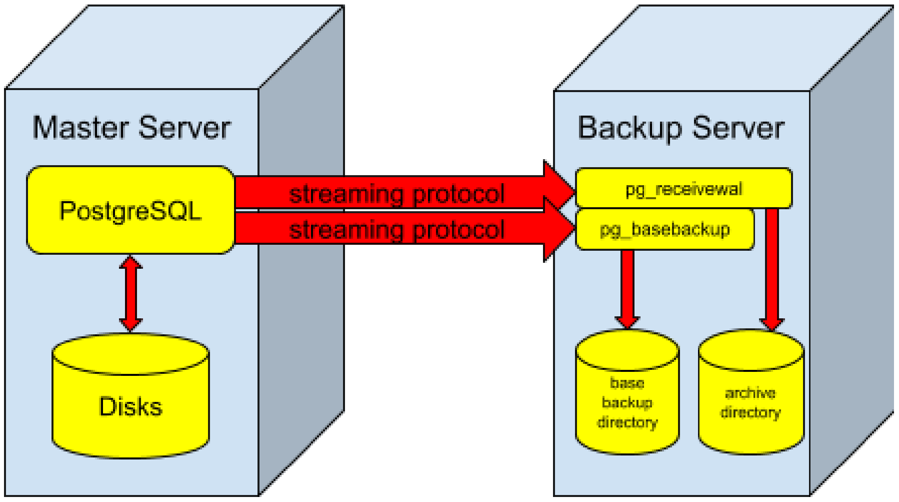
But a sequence of *WAL* files is nothing without a starting point. That's why we have to make what we call a basebackup. In PostgreSQL, we can do that with two different ways:

- Tell the database server that we're going to start a backup -> copy all the contents of the database server directory, including eventual tablespaces, by using *rsync*, *scp*, tar or whatever method we want -> tell the database server when we are done



PITR Strategy—In this case we use an archive_command to continuously archive, and rsync for the base backup—the master server "pushes" data to the ssh daemon on the backup server.

- Use the *pg_basebackup* tool—it connects to the database server using the same type of streaming connection used to pull *WAL* data, but in this case it will copy the contents of the database server directory and tablespaces automatically, and save it to a local directory.



PITR Strategy—Using PostgreSQL native tools to "pull" data from the master server

Needless to say that by using *pg_basebackup* and *pg_receivewal* tools our life is made much easier than dealing with scripts, commands and network stuff. These tools take care of it for us and are reliable, because they were written to PostgreSQL by PostgreSQL developers.

But does all of that work with the live database without issues ? Yes it does. But how can we copy "hot" database binary files and be consistent at the end, since those files are being constantly changed? Well, that's why we need a minimum set of *WAL* files, from the beginning to the end of the copy, when restoring, we'll need to replay at least the *WAL* generated in between. They say to PostgreSQL how to reconstruct the database, from the basebackup to a consistent state by replaying all the write operations made.

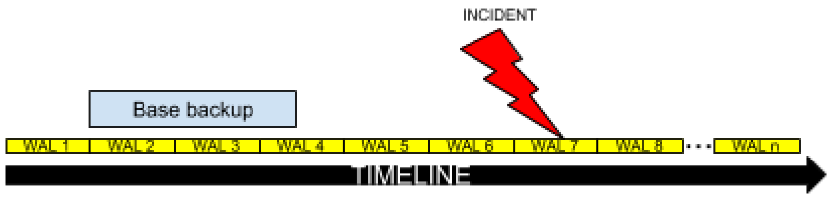
What are the strengths of the *PITR* strategy?

- Copying data will be as fast as your disks and network—we can calculate all that and have margins for the system to keep on working during backup as nothing else is going on
- Restore will use the same metrics so we can predict how much time we'll need to recover in case of disaster
- Restore is usually faster than with dumps—we don't need to recalculate indexes and constraints that are needed when restoring dumps, we just copy files.
- We can restore the fastest way—just replay enough *WAL* to have a consistent database
- We can restore to the max—replay all *WAL* that we have archived, so we'll restore as close as possible to the incident and lose a minimum of data
- We can restore to a specific point in time—hence the name of the strategy, we can give PostgreSQL a timestamp where to stop replaying *WAL*. Let's say that someone has mistyped a command and dropped a database. We take the moment that happened and say to PostgreSQL to stop a second before—voilà! minimum loss of data and all your database is back.
- Other than that, we can use previous marks that we can create by saying their names to the server. For example, just before a risky operation, we can "mark" the start of that operation with a name and, in case of problems, we can restore until that mark.
- As with dumps, it can be done (actually it has to be done) with a live database system, but it puts less stress on the system since it only needs disk reads and some network bandwidth; all the other processing can be made on a remote system.

The *PITR* strategy also has some restrictions compared to dumps:

- A basebackup is as big as the live database on disk, because it includes all the PostgreSQL catalog, indexes and the natural bloat of database files.
- Compression can be used after copying but is not very effective because it can be very slow to create and even slower to restore if we chose a high compression format like *bzip2*.
- They are not portable at all—we can only restore the database in the same PostgreSQL major version, on the same operating system and architecture (a Linux 64 bit origin won't restore on a Linux 32 bit and even less on Windows, MacOS or any BSD). PostgreSQL binaries have to be compiled with the exact same options.
- There's no granularity—we can only backup the whole database cluster (instance) with all its databases and, with them, all its tables and objects.
- To be consistent, we need to *WAL* replay at least to a minimum to achieve consistency. Just copying the basebackup won't work as it's not consistent because the database was live and the copy is not atomic

- There's no guarantee that the database was in good shape because the copy is physical—if something is broken like incomplete pages, malformed indexes or broken relations between tables, it will be copied as is without warning. And will be restored as is too. Maybe the *WAL* replay will hang but it's rare.



The PITR Strategy in details—the file WAL 1 is useless because it arrived before base backup and can be deleted. For a minimum restore we'll need WAL 2 to 4. To recover just before the incident, we'll need WAL 2 to 7. For a full restore we can use all WAL present.

Ok, which strategy to choose from?

The answer is—it depends.

If you really don't need *PITR* characteristics like you can have some data loss during the day as the example in the dumps item above, you may be fine with just dumps.

In all the other cases, the recommended is to have **both** strategies configured. Most strengths of one strategy are the drawbacks of the other and vice-versa, so we'll be better having all possibilities at hand when in trouble.

Great, understood, but PITR seems a bit complicated

It's not too complicated when we understand how it works but there are some tools that can be of help:

- *barman*[2]—this is used at **leboncoin**. It handles *PITR* very well and organizes multiple server backups. You set up backup policies and it can manage old backups and all the associated *WAL* files. It can work with *rsync* + log shipping (useful if you have very old PostgreSQL versions before 9.4) or *pg_basebackup* and *pg_receivewal*, our choice at **leboncoin**.
- *pgbackrest*[3]—similar to barman in many ways, under the hood it uses *rsync* and log shipping only, but the tool is evolving fast and can have new features as you read this.

- One can obviously set up a server with cron jobs to handle *pg_basebackup* and some sort of daemonizing *pg_receivewal* as well as handling retention with scripts, but be careful—it's too easy to have backups that just plain don't work. And you may only discover it when it's already too late.

Wait for the second part

In the second part, we'll talk about:

- restore tests and why is it important
- how each strategy is set up at **leboncoin**
- elapsed time to backup and to restore depending on database size and type
- basebackup intervals
- dump intervals
- discussing the required retention
- restore cases and techniques

Stay tuned!

Links

1. https://github.com/flaviogurgel
2. https://www.pgbarman.org/
3. https://pgbackrest.org/