Barman

Backup and recovery manager for PostgreSQL

Barman Manual

August 22, 2024 (3.11.1)

EnterpriseDB UK Limited

Contents

Barman (Backup and Recovery Manager) is an open-source administration tool for disaster recovery of PostgreSQL servers written in Python. It allows your organisation to perform remote backups of multiple servers in business critical environments to reduce risk and help DBAs during the recovery phase.

Barman is distributed under GNU GPL 3 and maintained by EnterpriseDB, a platinum sponsor of the PostgreSQL project.

> **IMPORTANT:**
> This manual assumes that you are familiar with theoretical disaster recovery concepts, and that you have a grasp of PostgreSQL fundamentals in terms of physical backup and disaster recovery. See section *"Before you start"* below for details.

Introduction

In a perfect world, there would be no need for a backup. However, it is important, especially in business environments, to be prepared for when the *"unexpected"* happens. In a database scenario, the unexpected could take any of the following forms:

- data corruption
- system failure (including hardware failure)
- human error
- natural disaster

In such cases, any ICT manager or DBA should be able to fix the incident and recover the database in the shortest time possible. We normally refer to this discipline as **disaster recovery**, and more broadly *business continuity*.

Within business continuity, it is important to familiarise yourself with two fundamental metrics, as defined by Wikipedia:

- **Recovery Point Objective (RPO)**: *"maximum targeted period in which data might be lost from an IT service due to a major incident"*
- **Recovery Time Objective (RTO)**: *"the targeted duration of time and a service level within which a business process must be restored after a disaster (or disruption) in order to avoid unacceptable consequences associated with a break in business continuity"*

In a few words, RPO represents the maximum amount of data you can afford to lose, while RTO represents the maximum down-time you can afford for your service.

Understandably, we all want **RPO=0** (*"zero data loss"*) and **RTO=0** (*zero down-time*, utopia) - even if it is our grandmothers's recipe website. In reality, a careful cost analysis phase allows you to determine your business continuity requirements.

Fortunately, with an open source stack composed of **Barman** and **PostgreSQL**, you can achieve RPO=0 thanks to synchronous streaming replication. RTO is more the focus of a *High Availability* solution, like **repmgr**. Therefore, by integrating Barman and repmgr, you can dramatically reduce RTO to nearly zero.

Based on our experience at EnterpriseDB, we can confirm that PostgreSQL open source clusters with Barman and repmgr can easily achieve more than 99.99% uptime over a year, if properly configured and monitored.

In any case, it is important for us to emphasise more on cultural aspects related to disaster recovery, rather than the actual tools. Tools without human beings are useless.

Our mission with Barman is to promote a culture of disaster recovery that:

- focuses on backup procedures

- focuses even more on recovery procedures
- relies on education and training on strong theoretical and practical concepts of PostgreSQL's crash recovery, backup, Point-In-Time-Recovery, and replication for your team members
- promotes testing your backups (only a backup that is tested can be considered to be valid), either manually or automatically (be creative with Barman's hook scripts!)
- fosters regular practice of recovery procedures, by all members of your devops team (yes, developers too, not just system administrators and DBAs)
- solicits to regularly scheduled drills and disaster recovery simulations with the team every 3-6 months
- relies on continuous monitoring of PostgreSQL and Barman, and that is able to promptly identify any anomalies

Moreover, do everything you can to prepare yourself and your team for when the disaster happens (yes, *when*), because when it happens:

- It is going to be a Friday evening, most likely right when you are about to leave the office.
- It is going to be when you are on holiday (right in the middle of your cruise around the world) and somebody else has to deal with it.
- It is certainly going to be stressful.
- You will regret not being sure that the last available backup is valid.
- Unless you know how long it approximately takes to recover, every second will seem like forever.

Be prepared, don't be scared.

In 2011, with these goals in mind, 2ndQuadrant started the development of Barman, now one of the most used backup tools for PostgreSQL. Barman is an acronym for "Backup and Recovery Manager".

Currently, Barman works only on Linux and Unix operating systems.

Before you start

Before you start using Barman, it is fundamental that you get familiar with PostgreSQL and the concepts around physical backups, Point-In-Time-Recovery and replication, such as base backups, WAL archiving, etc.

Below you can find a non exhaustive list of resources that we recommend for you to read:

- *PostgreSQL documentation*:
  - SQL Dump[1]
  - File System Level Backup
  - Continuous Archiving and Point-in-Time Recovery (PITR)
  - Reliability and the Write-Ahead Log
- *Book*: PostgreSQL 10 Administration Cookbook

Professional training on these topics is another effective way of learning these concepts. At any time of the year you can find many courses available all over the world, delivered by PostgreSQL companies such as EnterpriseDB.

[1] It is important that you know the difference between logical and physical backup, therefore between `pg_dump` and a tool like Barman.

Design and architecture

Where to install Barman

One of the foundations of Barman is the ability to operate remotely from the database server, via the network.

Theoretically, you could have your Barman server located in a data centre in another part of the world, thousands of miles away from your PostgreSQL server. Realistically, you do not want your Barman server to be too far from your PostgreSQL server, so that both backup and recovery times are kept under control.

Even though there is no *"one size fits all"* way to setup Barman, there are a couple of recommendations that we suggest you abide by, in particular:

- Install Barman on a dedicated server
- Do not share the same storage with your PostgreSQL server
- Integrate Barman with your monitoring infrastructure [2]
- Test everything before you deploy it to production

A reasonable way to start modelling your disaster recovery architecture is to:

- design a couple of possible architectures in respect to PostgreSQL and Barman, such as:
  1. same data centre
  2. different data centre in the same metropolitan area
  3. different data centre
- elaborate the pros and the cons of each hypothesis
- evaluate the single points of failure (SPOF) of your system, with cost-benefit analysis
- make your decision and implement the initial solution

Having said this, a very common setup for Barman is to be installed in the same data centre where your PostgreSQL servers are. In this case, the single point of failure is the data centre. Fortunately, the impact of such a SPOF can be alleviated thanks to two features that Barman provides to increase the number of backup tiers:

1. **geographical redundancy** (introduced in Barman 2.6)
2. **hook scripts**

[2]Integration with Nagios/Icinga is straightforward thanks to the `barman check --nagios` command, one of the most important features of Barman and a true lifesaver.

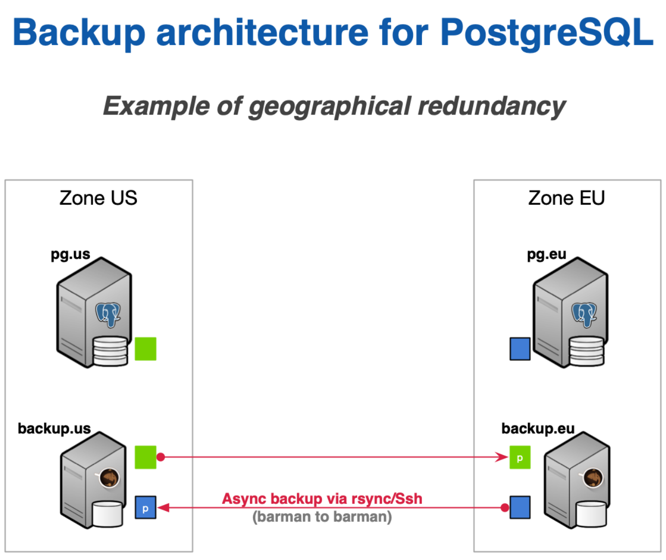
Backup architecture for PostgreSQL
Example of geographical redundancy
Zone US
pg.us
backup.us
Zone EU
pg.eu
backup.eu
p
p
Async backup via rsync/Ssh
(barman to barman)

Figure 1: An example of architecture with geo-redundancy

With *geographical redundancy*, you can rely on a Barman instance that is located in a different data centre/availability zone to synchronise the entire content of the source Barman server. There's more: given that geo-redundancy can be configured in Barman not only at global level, but also at server level, you can create *hybrid installations* of Barman where some servers are directly connected to the local PostgreSQL servers, and others are backing up subsets of different Barman installations (*cross-site backup*). Figure 1 below shows two availability zones (one in Europe and one in the US), each with a primary PostgreSQL server that is backed up in a local Barman installation, and relayed on the other Barman server (defined as *passive*) for multi-tier backup via rsync/SSH. Further information on geo-redundancy is available in the specific section.

Thanks to *hook scripts* instead, backups of Barman can be exported on different media, such as *tape* via `tar`, or locations, like an *S3 bucket* in the Amazon cloud.

Remember that no decision is forever. You can start this way and adapt over time to the solution that suits you best. However, try and keep it simple to start with.

One Barman, many PostgreSQL servers

Another relevant feature that was first introduced by Barman is support for multiple servers. Barman can store backup data coming from multiple PostgreSQL instances, even with different versions, in a centralised way. [3]

[3]The same requirements for PostgreSQL's PITR apply for recovery, as detailed in the section *"Requirements for recovery"*.

As a result, you can model complex disaster recovery architectures, forming a "star schema", where PostgreSQL servers rotate around a central Barman server.

Every architecture makes sense in its own way. Choose the one that resonates with you, and most importantly, the one you trust, based on real experimentation and testing.

From this point forward, for the sake of simplicity, this guide will assume a basic architecture:

- one PostgreSQL instance (with host name `pg`)
- one backup server with Barman (with host name `backup`)

Streaming backup vs rsync/SSH

Barman is able to take backups using either Rsync, which uses SSH as a transport mechanism, or `pg_basebackup`, which uses PostgreSQL's streaming replication protocol.

Choosing one of these two methods is a decision you will need to make, however for general usage we recommend using streaming replication for all currently supported versions of PostgreSQL.

> **IMPORTANT:**
> Because Barman transparently makes use of `pg_basebackup`, features such as parallel backup are currently not available. In this case, bandwidth limitation has some restrictions - compared to the traditional method via `rsync`.

Backup using `rsync`/SSH is recommended in cases where `pg_basebackup` limitations pose an issue for you.

The reason why we recommend streaming backup is that, based on our experience, it is easier to setup than the traditional one. Also, streaming backup allows you to backup a PostgreSQL server on Windows[4], and makes life easier when working with Docker.

The Barman WAL archive

Recovering a PostgreSQL backup relies on replaying transaction logs (also known as *xlog* or WAL files). It is therefore essential that WAL files are stored by Barman alongside the base backups so that they are available at recovery time. This can be achieved using either WAL streaming or standard WAL archiving to copy WALs into Barman's WAL archive.

WAL streaming involves streaming WAL files from the PostgreSQL server with `pg_receivewal` using replication slots. WAL streaming is able to reduce the risk of data loss, bringing RPO down to *near zero* values. It is also possible to add Barman as a synchronous WAL receiver in your PostgreSQL cluster and achieve **zero data loss** (RPO=0).

[4]Backup of a PostgreSQL server on Windows is possible, but it is still experimental because it is not yet part of our continuous integration system. See section *"How to setup a Windows based server"* for details.

Barman also supports standard WAL file archiving which is achieved using PostgreSQL's `archive_command` (either via `rsync`/SSH, or via `barman-wal-archive` from the `barman-cli` package). With this method, WAL files are archived only when PostgreSQL *switches* to a new WAL file. To keep it simple this normally happens every 16MB worth of data changes.

It is *required* that one of WAL streaming or WAL archiving is configured. It is optionally possible to configure both WAL streaming *and* standard WAL archiving - in such cases Barman will automatically de-duplicate incoming WALs. This provides a fallback mechanism so that WALs are still copied to Barman's archive in the event that WAL streaming fails.

For general usage we recommend configuring WAL streaming only.

> **NOTE:** Previous versions of Barman recommended that both WAL archiving *and* WAL streaming were used. This was because PostreSQL versions older than 9.4 did not support replication slots and therefore WAL streaming alone could not guarantee all WALs would be safely stored in Barman's WAL archive. Since all supported versions of PostgreSQL now have replication slots it is sufficient to configure only WAL streaming.

Two typical scenarios for backups

In order to make life easier for you, below we summarise the two most typical scenarios for a given PostgreSQL server in Barman.

Bear in mind that this is a decision that you must make for every single server that you decide to back up with Barman. This means that you can have heterogeneous setups within the same installation.

As mentioned before, we will only worry about the PostgreSQL server (`pg`) and the Barman server (`backup`). However, in real life, your architecture will most likely contain other technologies such as repmgr, pgBouncer, Nagios/Icinga, and so on.

Scenario 1: Backup via streaming protocol

A streaming backup installation is recommended for most use cases - see figure 2 below.

In this scenario, you will need to configure:

1. a standard connection to PostgreSQL, for management, coordination, and monitoring purposes
2. a streaming replication connection that will be used by both `pg_basebackup` (for base backup operations) and `pg_receivewal` (for WAL streaming)

In Barman's terminology this setup is known as **streaming-only** setup as it does not use an SSH connection for backup and archiving operations. This is particularly suitable and extremely practical for Docker environments.

As discussed in "The Barman WAL archive", you can configure WAL archiving via SSH *in addition to* WAL streaming - see figure 3 below.

WAL archiving via SSH requires:

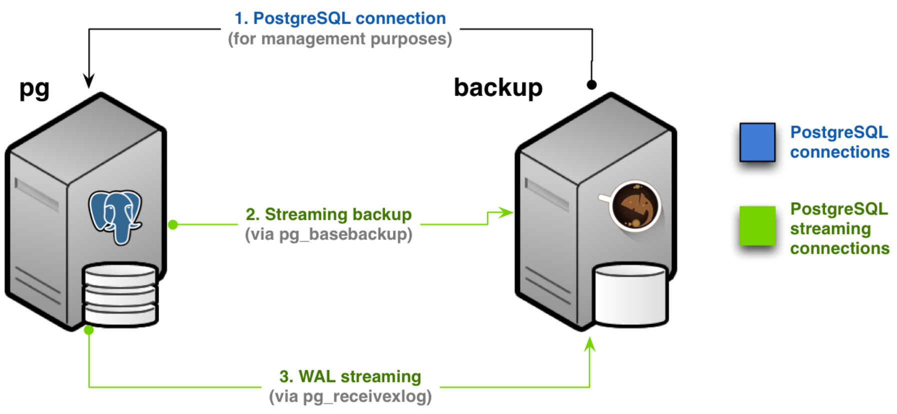

Figure 2: Streaming-only backup (Scenario 1)

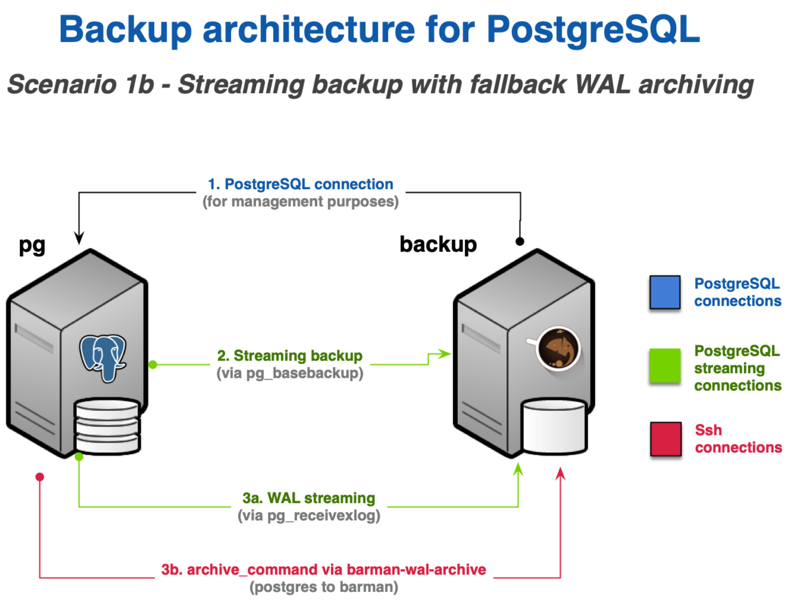
Backup architecture for PostgreSQL
Scenario 1b - Streaming backup with fallback WAL archiving
1. PostgreSQL connection
(for management purposes)
pg
backup
2. Streaming backup
(via pg_basebackup)
3a. WAL streaming
(via pg_receivexlog)
3b. archive_command via barman-wal-archive
(postgres to barman)
PostgreSQL connections
PostgreSQL streaming connections
Ssh connections

Figure 3: Streaming backup with WAL archiving (Scenario 1b)

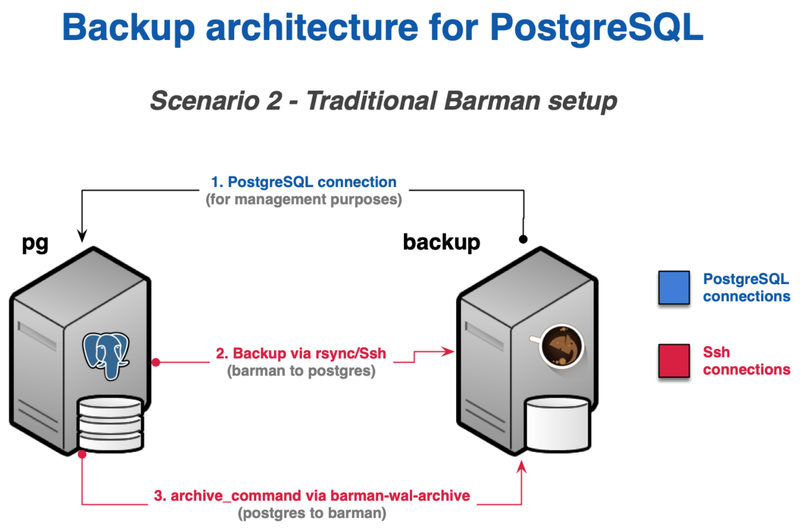
Backup architecture for PostgreSQL
Scenario 2 - Traditional Barman setup
1. PostgreSQL connection
(for management purposes)
pg
backup
2. Backup via rsync/Ssh
(barman to postgres)
3. archive_command via barman-wal-archive
(postgres to barman)
PostgreSQL connections
Ssh connections

Figure 4: Scenario 2 - Backup via rsync/SSH

- an additional SSH connection that allows the `postgres` user on the PostgreSQL server to connect as `barman` user on the Barman server
- the `archive_command` in PostgreSQL be configured to ship WAL files to Barman

Scenario 2: Backup via `rsync`/SSH

An `rsync`/SSH backup installation is required for cases where the following features are required:

- file-level incremental backup
- parallel backup
- finer control of bandwidth usage, including on a per-tablespace basis

In this scenario, you will need to configure:

1. a standard connection to PostgreSQL for management, coordination, and monitoring purposes
2. an SSH connection for base backup operations to be used by `rsync` that allows the `barman` user on the Barman server to connect as `postgres` user on the PostgreSQL server
3. an SSH connection for WAL archiving to be used by the `archive_command` in PostgreSQL and that allows the `postgres` user on the PostgreSQL server to connect as `barman` user on the Barman server

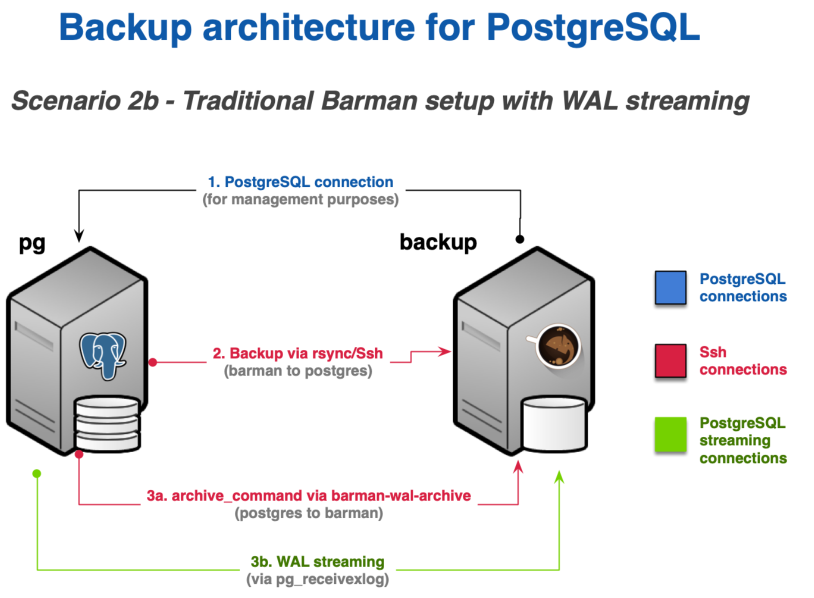
Backup architecture for PostgreSQL
Scenario 2b - Traditional Barman setup with WAL streaming
1. PostgreSQL connection (for management purposes)
pg
backup
2. Backup via rsync/Ssh (barman to postgres)
3a. archive_command via barman-wal-archive (postgres to barman)
3b. WAL streaming (via pg_receivexlog)
PostgreSQL connections
Ssh connections
PostgreSQL streaming connections

Figure 5: Backup via rsync/SSH with WAL streaming (Scenario 2b)

As an alternative to configuring WAL archiving in step 3, you can instead configure WAL streaming as described in Scenario 1. This will use a streaming replication connection instead of `archive_command` and significantly reduce RPO. As with Scenario 1 it is also possible to configure both WAL streaming and WAL archiving as shown in figure 5 below.

System requirements

- Linux/Unix
- Python >= 3.6
- Python modules:
  - argcomplete (optional)
  - psycopg2 >= 2.4.2
  - python-dateutil
  - setuptools
- PostgreSQL >= 10 (next version will require PostgreSQL >= 11)
- rsync >= 3.1.0 (optional)

> **IMPORTANT:** Users of RedHat Enterprise Linux, CentOS and Scientific Linux are required to install the Extra Packages Enterprise Linux (EPEL) repository.

> **NOTE:** Support for Python 2.6 and 3.5 are discontinued. Support for Python 2.7 is limited to Barman 3.4.X version and will receive only bugfixes. It will be discontinued in the near future. Support for Python 3.6 will be discontinued in future releases. Support for PostgreSQL < 10 is discontinued since Barman 3.0.0. Support for PostgreSQL 10 will be discontinued after Barman 3.5.0.

Requirements for backup

The most critical requirement for a Barman server is the amount of disk space available. You are recommended to plan the required disk space based on the size of the cluster, number of WAL files generated per day, frequency of backups, and retention policies.

Barman developers regularly test Barman with XFS and ext4. Like PostgreSQL, Barman does nothing special for NFS. The following points are required for safely using Barman with NFS:

- The `barman_lock_directory` should be on a non-network filesystem.
- Use version 4 of the NFS protocol.
- The file system must be mounted using the hard and synchronous options (`hard,sync`).

Requirements for recovery

Barman allows you to recover a PostgreSQL instance either locally (where Barman resides) or remotely (on a separate server).

Remote recovery is definitely the most common way to restore a PostgreSQL server with Barman.

Either way, the same requirements for PostgreSQL's Log shipping and Point-In-Time-Recovery apply:

- identical hardware architecture
- identical major version of PostgreSQL

In general, it is **highly recommended** to create recovery environments that are as similar as possible, if not identical, to the original server, because they are easier to maintain. For example, we suggest that you use the same operating system, the same PostgreSQL version, the same disk layouts, and so on.

Additionally, dedicated recovery environments for each PostgreSQL server, even on demand, allows you to nurture the disaster recovery culture in your team. You can be prepared for when something unexpected happens by practising recovery operations and becoming familiar with them.

Based on our experience, designated recovery environments reduce the impact of stress in real failure situations, and therefore increase the effectiveness of recovery operations.

Finally, it is important that time is synchronised between the servers, using NTP for example.

Installation

Official packages for Barman are distributed by EnterpriseDB through repositories listed on the Barman downloads page.

These packages use the default python3 version provided by the target operating system. If an alternative python3 version is required then you will need to install Barman from source.

> **IMPORTANT:** The recommended way to install Barman is by using the available packages for your GNU/Linux distribution.

Installation on Red Hat Enterprise Linux (RHEL) and RHEL-based systems using RPM packages

Barman can be installed using RPM packages on RHEL8 and RHEL7 systems and the identical versions of RHEL derivatives AlmaLinux, Oracle Linux, and Rocky Linux. It is required to install the Extra Packages Enterprise Linux (EPEL) repository and the PostgreSQL Global Development Group RPM repository beforehand.

Official RPM packages for Barman are distributed by EnterpriseDB via Yum through the public RPM repository, by following the instructions you find on that website.

Then, as `root` simply type:

```
yum install barman
```

In addition to the Barman packages available in the EDB and PGDG repositories, Barman RPMs published by the Fedora project can be found in EPEL. These RPMs are not maintained by the Barman developers and use a different configuration layout to the packages available in the PGDG and EDB repositories:

- EDB and PGDG packages use `/etc/barman.conf` as the main configuration file and `/etc/barman.d` for additional configuration files.
- The Fedora packages use `/etc/barman/barman.conf` as the main configuration file and `/etc/barman/conf.d` for additional configuration files.

The difference in configuration file layout means that upgrades between the EPEL and non-EPEL Barman packages can break existing Barman installations until configuration files are manually updated. We therefore recommend that you use a single source repository for Barman packages. This can be achieved by adding the following line to the definition of the repositories from which you do not want to obtain Barman packages:

```
exclude=barman* python*-barman
```

Specifically:

- To use only Barman packages from the EDB repositories, add the exclude directive from above to repository definitions in `/etc/yum.repos.d/epel.repo` and `/etc/yum.repos.d/pgdg-*.repo`.
- To use only Barman packages from the PGDG repositories, add the exclude directive from above to repository definitions in `/etc/yum.repos.d/epel.repo` and `/etc/yum.repos.d/enterprisedb*.repo`.
- To use only Barman packages from the EPEL repositories, add the exclude directive from above to repository definitions in `/etc/yum.repos.d/pgdg-*.repo` and `/etc/yum.repos.d/enterprisedb*.repo`.

Installation on Debian/Ubuntu using packages

Barman can be installed on Debian and Ubuntu Linux systems using packages.

It is directly available in the official repository for Debian and Ubuntu, however, these repositories might not contain the latest available version. If you want to have the latest version of Barman, the recommended method is to install both these repositories:

- Public APT repository, directly maintained by Barman developers
- the PostgreSQL Community APT repository, by following instructions in the APT section of the PostgreSQL Wiki

> **NOTE:** Thanks to the direct involvement of Barman developers in the PostgreSQL Community APT repository project, you will always have access to the most updated versions of Barman.

Installing Barman is as easy. As `root` user simply type:

```
apt-get install barman
```

Installation on SLES using packages

Barman can be installed on SLES systems using packages available in the PGDG SLES repositories. Install the necessary repository by following the instructions available on the PGDG site.

Supported SLES version: SLES 15 SP3.

Once the necessary repositories have been installed you can install Barman as the `root` user:

```
zypper install barman
```

Installation from sources

> **WARNING:** Manual installation of Barman from sources should only be performed by expert GNU/Linux users. Installing Barman this way requires system administration activities such as dependencies management, `barman` user creation, configuration of the `barman.conf` file, cron setup for the `barman cron` command, log management, and so on.

Create a system user called `barman` on the `backup` server. As `barman` user, download the sources and uncompress them.

For a system-wide installation, type:

```
barman@backup$ ./setup.py build
# run this command with root privileges or through sudo
barman@backup# ./setup.py install
```

For a local installation, type:

```
barman@backup$ ./setup.py install --user
```

The `barman` application will be installed in your user directory (make sure that your `PATH` environment variable is set properly).

Barman is also available on the Python Package Index (PyPI) and can be installed through `pip`.

PostgreSQL client/server binaries

The following Barman features depend on PostgreSQL binaries:

- Streaming backup with `backup_method = postgres` (requires `pg_basebackup`)
- Streaming WAL archiving with `streaming_archiver = on` (requires `pg_receivewal` or `pg_receivexlog`)
- Verifying backups with `barman verify-backup` (requires `pg_verifybackup`)

Depending on the target OS these binaries are installed with either the PostgreSQL client or server packages:

- On RedHat/CentOS and SLES:
  - The `pg_basebackup` and `pg_receivewal`/`pg_receivexlog` binaries are installed with the PostgreSQL client packages.
  - The `pg_verifybackup` binary is installed with the PostgreSQL server packages.
  - All binaries are installed in `/usr/pgsql-${PG_MAJOR_VERSION}/bin`.

- On Debian/Ubuntu:
  - All binaries are installed with the PostgreSQL client packages.
  - The binaries are installed in `/usr/lib/postgresql/${PG_MAJOR_VERSION}/bin`.

You must ensure that either:

1. The Barman user has the `bin` directory for the appropriate `PG_MAJOR_VERSION` on its path, or:
2. The path_prefix option is set in the Barman configuration for each server and points to the `bin` directory for the appropriate `PG_MAJOR_VERSION`.

The psql program is recommended in addition to the above binaries. While Barman does not use it directly the documentation provides examples of how it can be used to verify PostgreSQL connections are working as intended. The `psql` binary can be found in the PostgreSQL client packages.

Third party PostgreSQL variants

If you are using Barman for the backup and recovery of third-party PostgreSQL variants then you will need to check whether the PGDG client/server binaries described above are compatible with your variant. If they are incompatible then you will need to install compatible alternatives from appropriate packages.

Upgrading Barman

Barman follows the trunk-based development paradigm, and as such there is only one stable version, the latest. After every commit, Barman goes through thousands of automated tests for each supported PostgreSQL version and on each supported Linux distribution.

Also, **every version is back compatible** with previous ones. Therefore, upgrading Barman normally requires a simple update of packages using `yum update` or `apt update`.

There have been, however, the following exceptions in our development history, which required some small changes to the configuration.

Upgrading to Barman 3.0.0

Default backup approach for Rsync backups is now concurrent

Barman will now use concurrent backups if neither `concurrent_backup` nor `exclusive_backup` are specified in `backup_options`. This differs from previous Barman versions where the default was to use exclusive backup.

If you require exclusive backups you will now need to add `exclusive_backup` to `backup_options` in the Barman configuration.

Note that exclusive backups are not supported at all when running against PostgreSQL 15.

Metadata changes

A new field named `compression` will be added to the metadata stored in the `backup.info` file for all backups taken with version 3.0.0. This is used when recovering from backups taken using the built-in compression functionality of `pg_basebackup`.

The presence of this field means that earlier versions of Barman are not able to read backups taken with Barman 3.0.0. This means that if you downgrade from Barman 3.0.0 to an earlier version you will have to either manually remove any backups taken with 3.0.0 or edit the `backup.info` file of each backup to remove the `compression` field.

The same metadata change affects pg-backup-api so if you are using pg-backup-api you will need to update it to version 0.2.0.

Upgrading from Barman 2.10

If you are using `barman-cloud-wal-archive` or `barman-cloud-backup` you need to be aware that from version 2.11 all cloud utilities have been moved into the new `barman-cli-cloud` package. Therefore, you need to ensure that the `barman-cli-cloud` package is properly installed as part of the upgrade to the latest version. If you are not using the above tools, you can upgrade to the latest version as usual.

Upgrading from Barman 2.X (prior to 2.8)

Before upgrading from a version of Barman 2.7 or older users of `rsync` backup method on a primary server should explicitly set `backup_options` to either `concurrent_backup` (recommended for PostgreSQL 9.6 or higher) or `exclusive_backup` (current default), otherwise Barman emits a warning every time it runs.

Upgrading from Barman 1.X

If your Barman installation is 1.X, you need to explicitly configure the archiving strategy. Before, the file based archiver, controlled by `archiver`, was enabled by default.

Before you upgrade your Barman installation to the latest version, make sure you add the following line either globally or for any server that requires it:

```
archiver = on
```

Additionally, for a few releases, Barman will transparently set `archiver = on` with any server that has not explicitly set an archiving strategy and emit a warning.

Configuration

There are three types of configuration files in Barman:

- **global/general configuration**
- **server configuration**
- **model configuration**

The main configuration file (set to `/etc/barman.conf` by default) contains general options such as main directory, system user, log file, and so on.

Server configuration files, one for each server to be backed up by Barman, are located in the `/etc/barman.d` directory and must have a `.conf` suffix.

Similarly, model configuration files are located in the `/etc/barman.d` directory and must have a `.conf` suffix.

> *NOTE*: models define a set of configuration overrides which can be applied on top of the configuration of Barman servers that are part of the same cluster as the model, through the barman config-switch command.

> **IMPORTANT**: For historical reasons, you can still have one single configuration file containing both global as well as server and model options. However, for maintenance reasons, this approach is deprecated.

Configuration files in Barman follow the *INI* format.

Configuration files accept distinct types of parameters:

- string
- enum
- integer
- boolean, `on/true/1` are accepted as well are `off/false/0`.

None of them requires to be quoted.

> *NOTE*: some `enum` allows `off` but not `false`.

Options scope

Every configuration option has a *scope*:

- global
- server
- model
- global/server: server options that can be generally set at global level

Global options are allowed in the *general section*, which is identified in the INI file by the `[barman]` label:

```
[barman]
; ... global and global/server options go here
```

Server options can only be specified in a *server section*, which is identified by a line in the configuration file, in square brackets (`[` and `]`). The server section represents the ID of that server in Barman. The following example specifies a section for the server named `pg`, which belongs to the `my-cluster` cluster:

```
[pg]
cluster=my-cluster
; Configuration options for the
; server named 'pg' go here
```

Model options can only be specified in a *model section*, which is identified the same way as a *server section*. There can be no conflicts among the identifier of *server sections* and *model sections*. The following example specifies a section for the model named `pg:switchover`, which belongs to the `my-cluster` cluster:

```
[pg:switchover]
cluster=my-cluster
model=true
; Configuration options for the model named 'pg:switchover', which belongs to
; the server which is configured with the option 'cluster=pg', go here
```

There are two reserved words that cannot be used neither as server names nor as model names in Barman:

- `barman`: identifier of the global section
- `all`: a handy shortcut that allows you to execute some commands on every server managed by Barman in sequence

Barman implements the **convention over configuration** design paradigm, which attempts to reduce the number of options that you are required to configure without losing flexibility. Therefore, some server options can be defined at global level and overridden at server level, allowing users to specify a generic behavior and refine it for one or more servers. These options have a global/server scope.

For a list of all the available configurations and their scope, please refer to section 5 of the 'man' page.

```
man 5 barman
```

Examples of configuration

The following is a basic example of main configuration file:

```
[barman]
barman_user = barman
configuration_files_directory = /etc/barman.d
barman_home = /var/lib/barman
log_file = /var/log/barman/barman.log
log_level = INFO
compression = gzip
```

The example below, on the other hand, is a server configuration file that uses streaming backup:

```
[streaming-pg]
description =  "Example of PostgreSQL Database (Streaming-Only)"
conninfo = host=pg user=barman dbname=postgres
streaming_conninfo = host=pg user=streaming_barman
backup_method = postgres
streaming_archiver = on
slot_name = barman
```

The following example defines a configuration model with a set of overrides that can be applied to the server which cluster is `streaming-pg`:

```
[streaming-pg:switchover]
cluster=streaming-pg
model=true
conninfo = host=pg-2 user=barman dbname=postgres
streaming_conninfo = host=pg-2 user=streaming_barman
```

The following code shows a basic example of traditional backup using `rsync`/SSH:

```
[ssh-pg]
description =  "Example of PostgreSQL Database (via Ssh)"
ssh_command = ssh postgres@pg
conninfo = host=pg user=barman dbname=postgres
backup_method = rsync
parallel_jobs = 1
reuse_backup = link
archiver = on
```

For more detailed information, please refer to the distributed `barman.conf` file, as well as the `ssh-server.conf-template` and `streaming-server.conf-template` template files.

Setup of a new server in Barman

As mentioned in the *"Design and architecture"* section, we will use the following conventions:

- `pg` as server ID and host name where PostgreSQL is installed
- `backup` as host name where Barman is located
- `barman` as the user running Barman on the `backup` server (identified by the parameter `barman_user` in the configuration)
- `postgres` as the user running PostgreSQL on the `pg` server

> **IMPORTANT:** a server in Barman must refer to the same PostgreSQL instance for the whole backup and recoverability history (i.e. the same system identifier). **This means that if you perform an upgrade of the instance (using for example `pg_upgrade`, you must not reuse the same server definition in Barman, rather use another one as they have nothing in common.**

Preliminary steps

This section contains some preliminary steps that you need to undertake before setting up your PostgreSQL server in Barman.

> **IMPORTANT:** Before you proceed, it is important that you have made your decision in terms of WAL archiving and backup strategies, as outlined in the *"Design and architecture"* section. In particular, you should decide which WAL archiving methods to use, as well as the backup method.

PostgreSQL connection

You need to make sure that the `backup` server can connect to the PostgreSQL server on `pg` as superuser or, that the correct set of privileges are granted to the user that connects to the database.

You can create a specific superuser in PostgreSQL, named `barman`, as follows:

```
postgres@pg$ createuser -s -P barman
```

Or create a normal user with the required set of privileges as follows:

```
postgres@pg$ createuser -P barman
```

```
GRANT EXECUTE ON FUNCTION pg_backup_start(text, boolean) to barman;
GRANT EXECUTE ON FUNCTION pg_backup_stop(boolean) to barman;
GRANT EXECUTE ON FUNCTION pg_switch_wal() to barman;
GRANT EXECUTE ON FUNCTION pg_create_restore_point(text) to barman;

GRANT pg_read_all_settings TO barman;
GRANT pg_read_all_stats TO barman;
```

In the case of using PostgreSQL version 14 or a prior version, the functions `pg_backup_start` and `pg_backup_stop` had different names and different signatures. You will therefore need to replace the first two lines in the above block with:

```
GRANT EXECUTE ON FUNCTION pg_start_backup(text, boolean, boolean) to barman;
GRANT EXECUTE ON FUNCTION pg_stop_backup() to barman;
GRANT EXECUTE ON FUNCTION pg_stop_backup(boolean, boolean) to barman;
```

It is worth noting that with PostgreSQL version 13 and below without a real superuser, the `--force` option of the `barman switch-wal` command will not work.
If you are running PostgreSQL version 15 or above, you can grant the `pg_checkpoint` role, so you can use this feature without a superuser:

```
GRANT pg_checkpoint TO barman;
```

> **IMPORTANT:** The above `createuser` command will prompt for a password, which you are then advised to add to the `~barman/.pgpass` file on the `backup` server. For further information, please refer to "The Password File" section in the PostgreSQL Documentation.

This connection is required by Barman in order to coordinate its activities with the server, as well as for monitoring purposes.

You can choose your favourite client authentication method among those offered by PostgreSQL. More information can be found in the "Client Authentication" section of the PostgreSQL Documentation.

Run the following command as the barman user on the `backup` host in order to verify that the `backup` host can connect to PostgreSQL on the `pg` host:

```
barman@backup$ psql -c 'SELECT version()' -U barman -h pg postgres
```

Write down the above information (user name, host name and database name) and keep it for later. You will need it with in the `conninfo` option for your server configuration, like in this example:

```
[pg]
; ...
conninfo = host=pg user=barman dbname=postgres application_name=myapp
```

> **NOTE:** `application_name` is optional.

PostgreSQL WAL archiving and replication

Before you proceed, you need to properly configure PostgreSQL on `pg` to accept streaming replication connections from the Barman server. Please read the following sections in the PostgreSQL documentation:

- Role attributes
- The pg_hba.conf file
- Setting up standby servers using streaming replication

One configuration parameter that is crucially important is the `wal_level` parameter. This parameter must be configured to ensure that all the useful information necessary for a backup to be coherent are included in the transaction log file.

```
wal_level = 'replica'|'logical'
```

Restart the PostgreSQL server for the configuration to be refreshed.

PostgreSQL streaming connection

If you plan to use WAL streaming or streaming backup, you need to setup a streaming connection. We recommend creating a specific user in PostgreSQL, named `streaming_barman`, as follows:

```
postgres@pg$ createuser -P --replication streaming_barman
```

> **IMPORTANT:** The above command will prompt for a password, which you are then advised to add to the `~barman/.pgpass` file on the `backup` server. For further information, please refer to "The Password File" section in the PostgreSQL Documentation.

You can manually verify that the streaming connection works through the following command:

```
barman@backup$ psql -U streaming_barman -h pg \
  -c "IDENTIFY_SYSTEM" \
  replication=1
```

If the connection is working you should see a response containing the system identifier, current timeline ID and current WAL flush location, for example:

```
      systemid       | timeline |  xlogpos   | dbname
---------------------+----------+------------+--------
 7139870358166741016 |        1 | 1/330000D8 |
(1 row)
```

> **IMPORTANT:** Please make sure you are able to connect via streaming replication before going any further.

You also need to configure the `max_wal_senders` parameter in the PostgreSQL configuration file. The number of WAL senders depends on the PostgreSQL architecture you have implemented. In this example, we are setting it to `2`:

```
max_wal_senders = 2
```

This option represents the maximum number of concurrent streaming connections that the server will be allowed to manage.

Another important parameter is `max_replication_slots`, which represents the maximum number of replication slots [5] that the server will be allowed to manage. This parameter is needed if you are planning to use the streaming connection to receive WAL files over the streaming connection:

```
max_replication_slots = 2
```

The values proposed for `max_replication_slots` and `max_wal_senders` must be considered as examples, and the values you will use in your actual setup must be chosen after a careful evaluation of the architecture. Please consult the PostgreSQL documentation for guidelines and clarifications.

SSH connections

SSH is a protocol and a set of tools that allows you to open a remote shell to a remote server and copy files between the server and the local system. You can find more documentation about SSH usage in the article "SSH Essentials" by Digital Ocean.

SSH key exchange is a very common practice that is used to implement secure passwordless connections between users on different machines, and it's needed to use `rsync` for WAL archiving and for backups.

> **NOTE:** This procedure is not needed if you plan to use the streaming connection only to archive transaction logs and backup your PostgreSQL server.

SSH configuration of postgres user

Unless you have done it before, you need to create an SSH key for the PostgreSQL user. Log in as `postgres`, in the `pg` host and type:

```
postgres@pg$ ssh-keygen -t rsa
```

As this key must be used to connect from hosts without providing a password, no passphrase should be entered during the key pair creation.

[5] Replication slots have been introduced in PostgreSQL 9.4. See section *"WAL Streaming / Replication slots"* for details.

SSH configuration of barman user

As in the previous paragraph, you need to create an SSH key for the Barman user. Log in as `barman` in the `backup` host and type:

```
barman@backup$ ssh-keygen -t rsa
```

For the same reason, no passphrase should be entered.

From PostgreSQL to Barman

The SSH connection from the PostgreSQL server to the backup server is needed to correctly archive WAL files using the `archive_command` setting.

To successfully connect from the PostgreSQL server to the backup server, the PostgreSQL public key has to be configured into the authorized keys of the backup server for the `barman` user.

The public key to be authorized is stored inside the `postgres` user home directory in a file named `.ssh/id_rsa.pub`, and its content should be included in a file named `.ssh/authorized_keys` inside the home directory of the `barman` user in the backup server. If the `authorized_keys` file doesn't exist, create it using `600` as permissions.

The following command should succeed without any output if the SSH key pair exchange has been completed successfully:

```
postgres@pg$ ssh barman@backup -C true
```

The value of the `archive_command` configuration parameter will be discussed in the *"WAL archiving via archive_command section"*.

From Barman to PostgreSQL

The SSH connection between the backup server and the PostgreSQL server is used for the traditional backup over rsync. Just as with the connection from the PostgreSQL server to the backup server, we should authorize the public key of the backup server in the PostgreSQL server for the `postgres` user.

The content of the file `.ssh/id_rsa.pub` in the `barman` server should be put in the file named `.ssh/authorized_keys` in the PostgreSQL server. The permissions of that file should be `600`.

The following command should succeed without any output if the key pair exchange has been completed successfully.

```
barman@backup$ ssh postgres@pg -C true
```

The server configuration file

Create a new file, called `pg.conf`, in `/etc/barman.d` directory, with the following content:

```
[pg]
description =  "Our main PostgreSQL server"
conninfo = host=pg user=barman dbname=postgres
backup_method = postgres
# backup_method = rsync
```

The `conninfo` option is set accordingly to the section *"Preliminary steps: PostgreSQL connection"*.

The meaning of the `backup_method` option will be covered in the backup section of this guide.

If you plan to use the streaming connection for WAL archiving or to create a backup of your server, you also need a `streaming_conninfo` parameter in your server configuration file:

```
streaming_conninfo = host=pg user=streaming_barman dbname=postgres
```

This value must be chosen accordingly as described in the section *"Preliminary steps: PostgreSQL connection"*.

WAL streaming

Barman can reduce the Recovery Point Objective (RPO) by allowing users to add continuous WAL streaming from a PostgreSQL server, on top of the standard `archive_command` strategy.

Barman relies on `pg_receivewal`, it exploits the native streaming replication protocol and continuously receives transaction logs from a PostgreSQL server (master or standby). Prior to PostgreSQL 10, `pg_receivewal` was named `pg_receivexlog`.

> **IMPORTANT:** Barman requires that `pg_receivewal` is installed on the same server. It is recommended to install the latest available version of `pg_receivewal`, as it is back compatible. Otherwise, users can install multiple versions of `pg_receivewal` on the Barman server and properly point to the specific version for a server, using the `path_prefix` option in the configuration file.

In order to enable streaming of transaction logs, you need to:

1. setup a streaming connection as previously described
2. set the `streaming_archiver` option to `on`

The `cron` command, if the aforementioned requirements are met, transparently manages log streaming through the execution of the `receive-wal` command. This is the recommended scenario.

However, users can manually execute the `receive-wal` command:

```
barman receive-wal <server_name>
```

> **NOTE:** The `receive-wal` command is a foreground process.

Transaction logs are streamed directly in the directory specified by the `streaming_wals_directory` configuration option and are then archived by the `archive-wal` command.

Unless otherwise specified in the `streaming_archiver_name` parameter, Barman will set `application_name` of the WAL streamer process to `barman_receive_wal`, allowing you to monitor its status in the `pg_stat_replication` system view of the PostgreSQL server.

Replication slots

Replication slots are an automated way to ensure that the PostgreSQL server will not remove WAL files until they were received by all archivers. Barman uses this mechanism to receive the transaction logs from PostgreSQL.

You can find more information about replication slots in the PostgreSQL manual.

You can even base your backup architecture on streaming connection only. This scenario is useful to configure Docker-based PostgreSQL servers and even to work with PostgreSQL servers running on Windows.

> **IMPORTANT:** At this moment, the Windows support is still experimental, as it is not yet part of our continuous integration system.

How to configure the WAL streaming

First, the PostgreSQL server must be configured to stream the transaction log files to the Barman server.

To configure the streaming connection from Barman to the PostgreSQL server you need to enable the `streaming_archiver`, as already said, including this line in the server configuration file:

```
streaming_archiver = on
```

If you plan to use replication slots (recommended), another essential option for the setup of the streaming-based transaction log archiving is the `slot_name` option:

```
slot_name = barman
```

This option defines the name of the replication slot that will be used by Barman. It is mandatory if you want to use replication slots.

When you configure the replication slot name, you can manually create a replication slot for Barman with this command:

```
barman@backup$ barman receive-wal --create-slot pg
Creating physical replication slot 'barman' on server 'pg'
Replication slot 'barman' created
```

Starting with Barman 2.10, you can configure Barman to automatically create the replication slot by setting:

```
create_slot = auto
```

Streaming WALs and backups from different hosts (Barman 3.10.0 and later)

Barman uses the connection info defined in `streaming_conninfo` when creating `pg_receivewal` processes to stream WAL segments and uses `conninfo` when checking the status of replication slots. Because `conninfo` and `streaming_conninfo` are also used when taking backups this default configuration forces Barman to stream WALs and take backups from the same host.

If an alternative configuration is required, such as backups being sourced from a standby with WALs being streamed from the primary, then this can be achieved using the following options:

- `wal_streaming_conninfo`: A connection string which Barman will use instead of `streaming_conninfo` when receiving WAL segments via the streaming replication protocol and when checking the status of the replication slot used for receiving WALs.
- `wal_conninfo`: An optional connection string specifically for monitoring WAL streaming status and performing related checks. If set, Barman will use this instead of `wal_streaming_conninfo` when checking the status of the replication slot.

The following restrictions apply and are enforced by Barman during checks:

- Connections defined by `wal_streaming_conninfo` and `wal_conninfo` must reach a PostgreSQL instance which belongs to the same cluster reached by the `streaming_conninfo` and `conninfo` connections.
- The `wal_streaming_conninfo` connection string must be able to create streaming replication connections.
- Either `wal_streaming_conninfo` *or* `wal_conninfo` (if it is set) must have sufficient permissions to read settings and check replication slot status. The required permissions are one of:
  - The `pg_monitor` role.
  - Both the `pg_read_all_settings` and `pg_read_all_stats` roles.
  - The `superuser` role.

  IMPORTANT: While it is possible to stream WALs from *any* PostgreSQL instance in a cluster there is a risk that WAL segments can be lost when streaming WALs from a standby, if such a standby is unable to keep up with its own upstream source. For this reason it is *strongly recommended* that WALs are always streamed directly from the primary.

Limitations of partial WAL files with recovery

The standard behaviour of `pg_receivewal` is to write transactional information in a file with `.partial` suffix after the WAL segment name.

Barman expects a partial file to be in the `streaming_wals_directory` of a server. When completed, `pg_receivewal` removes the `.partial` suffix and opens the following one, delivering the file to the `archive-wal` command of Barman for permanent storage and compression.

In case of a sudden and unrecoverable failure of the master PostgreSQL server, the `.partial` file that has been streamed to Barman contains very important information that the standard archiver (through PostgreSQL's `archive_command`) has not been able to deliver to Barman.

As of Barman 2.10, the `get-wal` command is able to return the content of the current `.partial` WAL file through the `--partial/-P` option. This is particularly useful in the case of recovery, both full or to a point in time. Therefore, in case you run a `recover` command with `get-wal` enabled, and without `--standby-mode`, Barman will automatically add the `-P` option to `barman-wal-restore` (which will then relay that to the remote `get-wal` command) in the `restore_command` recovery option.

`get-wal` will also search in the `incoming` directory, in case a WAL file has already been shipped to Barman, but not yet archived.

WAL archiving via `archive_command`

The `archive_command` is the traditional method to archive WAL files.

The value of this PostgreSQL configuration parameter must be a shell command to be executed by the PostgreSQL server to copy the WAL files to the Barman incoming directory.

This can be done in two ways, both requiring a SSH connection:

- via `barman-wal-archive` utility (from Barman 2.6)
- via rsync/SSH (common approach before Barman 2.6)

See sections below for more details.

> **IMPORTANT:** Read the "Concurrent Backup and backup from a standby" section for more detailed information on how Barman supports this feature.

WAL archiving via `barman-wal-archive`

From Barman 2.6, the **recommended way** to safely and reliably archive WAL files to Barman via `archive_command` is to use the `barman-wal-archive` command contained in the `barman-cli` package, distributed via EnterpriseDB public repositories and available under GNU GPL 3 licence. `barman-cli` must be installed on each PostgreSQL server that is part of the Barman cluster.

Using barman-wal-archive instead of rsync/SSH reduces the risk of data corruption of the shipped WAL file on the Barman server. When using rsync/SSH as archive_command a WAL file, there is no mechanism that guarantees that the content of the file is flushed and fsync-ed to disk on destination.

For this reason, we have developed the barman-wal-archive utility that natively communicates with Barman's put-wal command (introduced in 2.6), which is responsible to receive the file, fsync its content and place it in the proper incoming directory for that server. Therefore, barman-wal-archive reduces the risk of copying a WAL file in the wrong location/directory in Barman, as the only parameter to be used in the archive_command is the server's ID.

For more information on the barman-wal-archive command, type man barman-wal-archive on the PostgreSQL server.

You can check that barman-wal-archive can connect to the Barman server, and that the required PostgreSQL server is configured in Barman to accept incoming WAL files with the following command:

```
barman-wal-archive --test backup pg DUMMY
```

Where backup is the host where Barman is installed, pg is the name of the PostgreSQL server as configured in Barman and DUMMY is a placeholder (barman-wal-archive requires an argument for the WAL file name, which is ignored).

If everything is configured correctly you should see the following output:

```
Ready to accept WAL files for the server pg
```

Since it uses SSH to communicate with the Barman server, SSH key authentication is required for the postgres user to login as barman on the backup server. If a port other than the SSH default of 22 should be used then the --port option can be added to specify the port that should be used for the SSH connection.

Edit the postgresql.conf file of the PostgreSQL instance on the pg database, activate the archive mode and set archive_command to use barman-wal-archive:

```
archive_mode = on
wal_level = 'replica'
archive_command = 'barman-wal-archive backup pg %p'
```

Then restart the PostgreSQL server.

WAL archiving via rsync/SSH

You can retrieve the incoming WALs directory using the show-servers Barman command and looking for the incoming_wals_directory value:

```
barman@backup$ barman show-servers pg |grep incoming_wals_directory
        incoming_wals_directory: /var/lib/barman/pg/incoming
```

Edit the `postgresql.conf` file of the PostgreSQL instance on the `pg` database and activate the archive mode:

```
archive_mode = on
wal_level = 'replica'
archive_command = 'rsync -a %p barman@backup:INCOMING_WALS_DIRECTORY/%f'
```

Make sure you change the `INCOMING_WALS_DIRECTORY` placeholder with the value returned by the `barman show-servers pg` command above.

Restart the PostgreSQL server.

In some cases, you might want to add stricter checks to the `archive_command` process. For example, some users have suggested the following one:

```
archive_command = 'test $(/bin/hostname --fqdn) = HOSTNAME \
  && rsync -a %p barman@backup:INCOMING_WALS_DIRECTORY/%f'
```

Where the `HOSTNAME` placeholder should be replaced with the value returned by `hostname --fqdn`. This *trick* is a safeguard in case the server is cloned and avoids receiving WAL files from recovered PostgreSQL instances.

Verification of WAL archiving configuration

In order to test that continuous archiving is on and properly working, you need to check both the PostgreSQL server and the backup server. In particular, you need to check that WAL files are correctly collected in the destination directory.

For this purpose and to facilitate the verification of the WAL archiving process, the `switch-wal` command has been developed:

```
barman@backup$ barman switch-wal --force --archive pg
```

The above command will force PostgreSQL to switch WAL file and trigger the archiving process in Barman. Barman will wait for one file to arrive within 30 seconds (you can change the timeout through the `--archive-timeout` option). If no WAL file is received, an error is returned.

You can verify if the WAL archiving has been correctly configured using the `barman check` command.

Streaming backup

Barman can backup a PostgreSQL server using the streaming connection, relying on `pg_basebackup`. Since version 3.11, Barman also supports block-level incremental backups using the streaming connection, for more information consult the *"Features in detail"* section.

> **IMPORTANT:** Barman requires that `pg_basebackup` is installed in the same server. It is recommended to install the last available version of `pg_basebackup`, as it is backwards compatible. You can even install multiple versions of `pg_basebackup` on the Barman server and properly point to the specific version for a server, using the `path_prefix` option in the configuration file.

To successfully backup your server with the streaming connection, you need to use `postgres` as your backup method:

```
backup_method = postgres
```

> **IMPORTANT:** You will not be able to start a backup if WAL is not being correctly archived to Barman, either through the `archiver` or the `streaming_archiver`

To check if the server configuration is valid you can use the `barman check` command:

```
barman@backup$ barman check pg
```

To start a backup you can use the `barman backup` command:

```
barman@backup$ barman backup pg
```

Backup with `rsync`/SSH

The backup over `rsync` was the only method for backups in Barman before version 2.0, and before 3.11 it was the only method that supported incremental backups. Current Barman supports file-level as well as block-level incremental backups. Backups using `rsync` implements the file-level backup feature. Please consult the *"Features in detail"* section for more information.

To take a backup using `rsync` you need to put these parameters inside the Barman server configuration file:

```
backup_method = rsync
ssh_command = ssh postgres@pg
```

The `backup_method` option activates the `rsync` backup method, and the `ssh_command` option is needed to correctly create an SSH connection from the Barman server to the PostgreSQL server.

> **IMPORTANT:** You will not be able to start a backup if WAL is not being correctly archived to Barman, either through the `archiver` or the `streaming_archiver`

To check if the server configuration is valid you can use the `barman check` command:

```
barman@backup$ barman check pg
```

To take a backup use the `barman backup` command:

```
barman@backup$ barman backup pg
```

> **NOTE:** Starting with Barman 3.11.0, Barman uses a keep-alive mechanism when taking rsync-based backups. It keeps sending a simple `SELECT 1` query over the libpq connection where Barman runs `pg_backup_start/pg_backup_stop` low-level API functions, and it's in place to reduce the probability of a firewall or a router dropping that connection as it can be idle for a long time while the base backup is being copied. You can control the interval of the hearbeats, or even disable the mechanism, through the `keepalive_interval` configuration option.

Backup with cloud snapshots

Barman is able to create backups of PostgreSQL servers deployed within certain cloud environments by taking snapshots of storage volumes. When configured in this manner the physical backups of PostgreSQL files are volume snapshots stored in the cloud while Barman acts as a storage server for WALs and the backup catalog. These backups can then be managed by Barman just like traditional backups taken with the `rsync` or `postgres` backup methods even though the backup data itself is stored in the cloud.

It is also possible to create snapshot backups without a Barman server using the barman-cloud-backup command directly on a suitable PostgreSQL server.

Prerequisites for cloud snapshots

In order to use the snapshot backup method with Barman, deployments must meet the following prerequisites:

- PostgreSQL must be deployed on a compute instance within a supported cloud provider.
- PostgreSQL must be configured such that all critical data, such as PGDATA and any tablespace data, is stored on storage volumes which support snapshots.
- The `findmnt` command must be available on the PostgreSQL host.

> **IMPORTANT:** Any configuration files stored outside of PGDATA will not be included in the snapshots. The management of such files must be carried out using another mechanism such as a configuration management system.

Google Cloud Platform snapshot prerequisites

The google-cloud-compute and grpcio libraries must be available to the Python distribution used by Barman. These libraries are an optional dependency and are not installed as standard by any of the Barman packages. They can be installed as follows using `pip`:

```
pip3 install grpcio google-cloud-compute
```

> **NOTE:** The minimum version of Python required by the google-cloud-compute library is 3.7. GCP snapshots cannot be used with earlier versions of Python.

The following additional prerequisites apply to snapshot backups on Google Cloud Platform:

- All disks included in the snapshot backup must be zonal persistent disks. Regional persistent disks are not currently supported.
- A service account with the required set of permissions must be available to Barman. This can be achieved by attaching such an account to the compute instance running Barman (recommended) or by using the `GOOGLE_APPLICATION_CREDENTIALS` environment variable to point to a credentials file.

The required permissions are:

- `compute.disks.createSnapshot`
- `compute.disks.get`
- `compute.globalOperations.get`
- `compute.instances.get`
- `compute.snapshots.create`
- `compute.snapshots.delete`
- `compute.snapshots.list`

Azure snapshot prerequisites

The azure-mgmt-compute and azure-identity libraries must be available to the Python distribution used by Barman.

These libraries are an optional dependency and are not installed as standard by any of the Barman packages. They can be installed as follows using `pip`:

```
pip3 install azure-mgmt-compute azure-identity
```

> **NOTE:** The minimum version of Python required by the azure-mgmt-compute library is 3.7. Azure snapshots cannot be used with earlier versions of Python.

The following additional prerequisites apply to snapshot backups on Azure:

- All disks included in the snapshot backup must be managed disks which are attached to the VM instance as data disks.
- Barman must be able to use a credential obtained either using managed identity or CLI login and this must grant access to Azure with the required set of permissions.

The following permissions are required:

- `Microsoft.Compute/disks/read`
- `Microsoft.Compute/virtualMachines/read`
- `Microsoft.Compute/snapshots/read`
- `Microsoft.Compute/snapshots/write`
- `Microsoft.Compute/snapshots/delete`

AWS snapshot prerequisites

The boto3 library must be available to the Python distribution used by Barman.

This library is an optional dependency and not installed as standard by any of the Barman packages. It can be installed as follows using `pip`:

```
pip3 install boto3
```

The following additional prerequisites apply to snapshot backups on AWS:

- All disks included in the snapshot backup must be non-root EBS volumes and must be attached to the same VM instance.
- NVMe volumes are not currently supported.

The following permissions are required:

- `ec2:CreateSnapshot`
- `ec2:CreateTags`
- `ec2:DeleteSnapshot`
- `ec2:DescribeSnapshots`
- `ec2:DescribeInstances`
- `ec2:DescribeVolumes`

Configuration for snapshot backups

To configure Barman for backup via cloud snapshots, set the `backup_method` parameter to `snapshot` and set `snapshot_provider` to a supported cloud provider:

```
backup_method = snapshot
snapshot_provider = gcp
```

Currently Google Cloud Platform (`gcp`), Microsoft Azure (`azure`) and AWS (`aws`) are supported.

The following parameters must be set regardless of cloud provider:

```
snapshot_instance = INSTANCE_NAME
snapshot_disks = DISK_NAME,DISK2_NAME,...
```

Where `snapshot_instance` is set to the name of the VM or compute instance where the storage volumes are attached and `snapshot_disks` is a comma-separated list of the disks which should be included in the backup.

> **IMPORTANT:** You must ensure that `snapshot_disks` includes every disk which stores data required by PostgreSQL. Any data which is not stored on a storage volume listed in `snapshot_disks` will not be included in the backup and therefore will not be available at recovery time.

Configuration for Google Cloud Platform snapshots

The following additional parameters must be set when using GCP:

```
gcp_project = GCP_PROJECT_ID
gcp_zone = ZONE
```

`gcp_project` should be set to the ID of the GCP project which owns the instance and storage volumes defined by `snapshot_instance` and `snapshot_disks`. `gcp_zone` should be set to the availability zone in which the instance is located.

Configuration for Azure snapshots

The following additional parameters must be set when using Azure:

```
azure_subscription_id = AZURE_SUBSCRIPTION_ID
azure_resource_group = AZURE_RESOURCE_GROUP
```

`azure_subscription_id` should be set to the ID of the Azure subscription ID which owns the instance and storage volumes defined by `snapshot_instance` and `snapshot_disks`. `azure_resource_group` should be set to the resource group to which the instance and disks belong.

Configuration for AWS snapshots

When specifying `snapshot_instance` or `snapshot_disks`, Barman will accept either the instance/volume ID which was assigned to the resource by AWS *or* a name. If a name is used then Barman will query AWS to find resources with a matching `Name` tag. If zero or multiple matching resources are found then Barman will exit with an error.

The following optional parameters can be set when using AWS:

```
aws_region = AWS_REGION
aws_profile = AWS_PROFILE_NAME
aws_await_snapshots_timeout = TIMEOUT_IN_SECONDS
```

If `aws_profile` is used it should be set to the name of a section in the AWS credentials file. If `aws_profile` is not used then the default profile will be used. If no credentials file exists then credentials will be sourced from the environment.

If `aws_region` is specified it will override any region that may be defined in the AWS profile.

If `aws_await_snapshots_timeout` is not set, the default of `3600` seconds will be used.

Taking a snapshot backup

Once the configuration options are set and appropriate credentials are available to Barman, backups can be taken using the barman backup command.

Barman will validate the configuration parameters for snapshot backups during the `barman check` command and also when starting a backup.

Note that the following arguments / config variables are unavailable when using `backup_method = snapshot`:

Command argument	Config variable
N/A	`backup_compression`
`--bwlimit`	`bandwidth_limit`
`--jobs`	`parallel_jobs`
N/A	`network_compression`
`--reuse-backup`	`reuse_backup`

For a more in-depth discussion of snapshot backups, including considerations around management and recovery of snapshot backups, see the cloud snapshots section in feature details.

How to setup a Windows based server

You can backup a PostgreSQL server running on Windows using the streaming connection for both WAL archiving and for backups.

> **IMPORTANT:** This feature is still experimental because it is not yet part of our continuous integration system.

Follow every step discussed previously for a streaming connection setup.

> **WARNING:**: At this moment, `pg_basebackup` interoperability from Windows to Linux is still experimental. If you are having issues taking a backup from a Windows server and your PostgreSQL locale is not in English, a possible workaround for the issue is instructing your PostgreSQL to emit messages in English. You can do this by putting the following parameter in your `postgresql.conf` file:
>
> ```
> lc_messages = 'English'
> ```
>
> This has been reported to fix the issue.

You can backup your server as usual.

Remote recovery is not supported for Windows servers, so you must recover your cluster locally in the Barman server and then copy all the files on a Windows server or use a folder shared between the PostgreSQL server and the Barman server.

Additionally, make sure that the system user chosen to run PostgreSQL has the permission needed to access the restored data. Basically, it must have full control over the PostgreSQL data directory.

General commands

Barman has many commands and, for the sake of exposition, we can organize them by scope.

The scope of the **general commands** is the entire Barman server, that can backup many PostgreSQL servers. **Server commands**, instead, act only on a specified server. **Backup commands** work on a backup, which is taken from a certain server.

The following list includes the general commands.

`cron`

`barman` doesn't include a long-running daemon or service file (there's nothing to `systemctl start`, `service start`, etc.). Instead, the `barman cron` subcommand is provided to perform `barman`'s background "steady-state" backup operations.

You can perform maintenance operations, on both WAL files and backups, using the `cron` command:

```
barman cron
```

> **NOTE:** This command should be executed in a *cron script*. Our recommendation is to schedule `barman cron` to run every minute. If you installed Barman using the rpm or debian package, a cron entry running on every minute will be created for you.

`barman cron` executes WAL archiving operations concurrently on a server basis, and this also enforces retention policies on those servers that have:

- `retention_policy` not empty and valid;
- `retention_policy_mode` set to `auto`.

The `cron` command ensures that WAL streaming is started for those servers that have requested it, by transparently executing the `receive-wal` command.

In order to stop the operations started by the `cron` command, comment out the cron entry and execute:

```
barman receive-wal --stop SERVER_NAME
```

You might want to check `barman list-servers` to make sure you get all of your servers.

> **NOTE:** `barman cron` runs background maintenance tasks only and is not responsible for running scheduled backups. Any regularly scheduled backup jobs you require must be scheduled separately, for example in another cron entry which runs `barman backup all`.

`diagnose`

The `diagnose` command creates a JSON report useful for diagnostic and support purposes. This report contains information for all configured servers.

> **NOTE:** From Barman `3.10.0` onwards you can optionally specify the `--show-config-source` argument to the command. In that case, for each configuration option of Barman and of the Barman servers, the output will include not only the configuration value, but also the configuration file which provides the effective value.

> **IMPORTANT:** Even if the diagnose is written in JSON and that format is thought to be machine readable, its structure is not to be considered part of the interface. Format can change between different Barman versions.

`list-servers`

You can display the list of active servers that have been configured for your backup system with:

```
barman list-servers
```

A machine readable output can be obtained with the `--minimal` option:

```
barman list-servers --minimal
```

Server commands

As we said in the previous section, server commands work directly on a PostgreSQL server or on its area in Barman, and are useful to check its status, perform maintenance operations, take backups, and manage the WAL archive.

`archive-wal`

The `archive-wal` command execute maintenance operations on WAL files for a given server. This operations include processing of the WAL files received from the streaming connection or from the `archive_command` or both.

> **IMPORTANT:** The `archive-wal` command, even if it can be directly invoked, is designed to be started from the `cron` general command.

`backup`

The `backup` command takes a full backup (*base backup*) of the given servers. It has several options that let you override the corresponding configuration parameter for the new backup. For more information, consult the manual page.

You can perform a full backup for a given server with:

```
barman backup <server_name>
```

> **TIP:** You can use `barman backup all` to sequentially backup all your configured servers.

> **TIP:** You can use `barman backup <server_1_name> <server_2_name>` to sequentially backup both `<server_1_name>` and `<server_2_name>` servers.

For information on how to take incremental backups in Barman, please check the incremental backup section.

Barman 2.10 introduces the `-w`/`--wait` option for the `backup` command. When set, Barman temporarily saves the state of the backup to `WAITING_FOR_WALS`, then waits for all the required WAL files to be archived before setting the state to `DONE` and proceeding with post-backup hook scripts. If the `--wait-timeout` option is provided, Barman will stop waiting for WAL files after the specified number of seconds, and the state will remain in `WAITING_FOR_WALS`.The `cron` command will continue to check that missing WAL files are archived, then label the backup as `DONE`.

`check`

You can check the connection to a given server and the configuration coherence with the `check` command:

```
barman check <server_name>
```

> **TIP:** You can use `barman check all` to check all your configured servers.
>
> **IMPORTANT:** The `check` command is probably the most critical feature that Barman implements. We recommend to integrate it with your alerting and monitoring infrastructure. The `--nagios` option allows you to easily create a plugin for Nagios/Icinga.

`config-update`

The `config-update` command is used to create or update configuration of servers and models in Barman

The syntax for running `config-update` command is:

```
barman config-update <json_changes>
```

`json_changes` should be a JSON string containing an array of documents. Each document must contain the following key:

- `scope`: either `server` or `model`, depending on if you want to create or update a Barman server or a Barman model;

They must also contain either of the following keys, depending on value of `scope`:

- `server_name`: if `scope` is `server`, you should fill this key with the Barman server name;
- `model_name`: if `scope` is `model`, you should fill this key with the Barman model name.

Besides these, you should fill each document with one or more Barman configuration options along with the desired values for them.

This is an example for updating the Barman server `my_server` with `archiver=on` and `streaming_archiver=off`:

```
barman config-update \
  ‘[{“scope”: “server”, “server_name”: “my_server”, “archiver”: “on”, “streaming_archiver”: “o
```

> *NOTE*: `barman config-update` command writes the configuration options to a file named `.barman.auto.conf`, which is created under the `barman_home`. That configuration file takes higher precedence and overrides values coming from the Barman global configuration file (typically `/etc/barman.conf`) and from included files as per `configuration_files_directory` (typically files in `/etc/barman.d`). Keep that in mind if you later, for any reason, decide to manually change configuration options in those files..

`config-switch`

The `config-switch` command is used to apply a set of configuration overrides defined through a model to a Barman server. The final configuration of the Barman server is composed of the configuration of the server plus the overrides applied by the selected model. Models are particularly useful for clustered environments, so you can create different configuration models which can be used in response to failover events, for example.

The syntax for applying a model through `config-switch` command is:

```
barman config-switch <server_name> <model_name>
```

> *NOTE*: the command will only succeed if `<model_name>` exists and belongs to the same `cluster` as `<server_name>`.

> *NOTE*: there can be at most one model active at a time. If you run the command twice with different models, only the overrides defined for the last one apply.

The syntax for unapplying an existing active model for a server is:

```
barman config-switch <server_name> --reset
```

It will take care of unapplying the overrides that were previously in place by some active model.

> *NOTE*: this command can also be useful for recovering from a specific situation: when you have a server with an active model which was previously configured but which no longer exists in your configuration.

`generate-manifest`

This command is useful when backup is created remotely and pg_basebackup is not involved and `backup_manifest` file does not exist in backup. It will generate `backup_manifest` file from backup_id using backup in barman server. If the file already exist, generation command will abort.

Command example:

```
barman generate-manifest <server_name> <backup_id>
```

Either backup_id backup id shortcuts can be used.

This command can also be used as post_backup hook script as follows:

```
post_backup_script=barman generate-manifest ${BARMAN_SERVER} ${BARMAN_BACKUP_ID}
```

`get-wal`

Barman allows users to request any *xlog* file from its WAL archive through the `get-wal` command:

```
barman get-wal [-o OUTPUT_DIRECTORY][-j|-x] <server_name> <wal_id>
```

If the requested WAL file is found in the server archive, the uncompressed content will be returned to `STDOUT`, unless otherwise specified.

The following options are available for the `get-wal` command:

- `-o` allows users to specify a destination directory where Barman will deposit the requested WAL file
- `-j` will compress the output using `bzip2` algorithm
- `-x` will compress the output using `gzip` algorithm
- `-p SIZE` peeks from the archive up to WAL files, starting from the requested file

It is possible to use `get-wal` during a recovery operation, transforming the Barman server into a *WAL hub* for your servers. This can be automatically achieved by adding the `get-wal` value to the `recovery_options` global/server configuration option:

```
recovery_options = 'get-wal'
```

`recovery_options` is a global/server option that accepts a list of comma separated values. If the keyword `get-wal` is present during a recovery operation, Barman will prepare the recovery configuration by setting the `restore_command` so that `barman get-wal` is used to fetch the required WAL files. Similarly, one can use the `--get-wal` option for the `recover` command at run-time.

If `get-wal` is set in `recovery_options` but not required during a recovery operation then the `--no-get-wal` option can be used with the `recover` command to disable the `get-wal` recovery option.

This is an example of a `restore_command` for a local recovery:

```
restore_command = 'sudo -u barman barman get-wal SERVER %f > %p'
```

Please note that the `get-wal` command should always be invoked as `barman` user, and that it requires the correct permission to read the WAL files from the catalog. This is the reason why we are using `sudo -u barman` in the example.

Setting `recovery_options` to `get-wal` for a remote recovery will instead generate a `restore_command` using the `barman-wal-restore` script. `barman-wal-restore` is a more resilient shell script which manages SSH connection errors.

This script has many useful options such as the automatic compression and decompression of the WAL files and the *peek* feature, which allows you to retrieve the next WAL files while PostgreSQL is applying one of them. It is an excellent way to optimise the bandwidth usage between PostgreSQL and Barman.

`barman-wal-restore` is available in the `barman-cli` package.

This is an example of a `restore_command` for a remote recovery:

```
restore_command = 'barman-wal-restore -U barman backup SERVER %f %p'
```

Since it uses SSH to communicate with the Barman server, SSH key authentication is required for the `postgres` user to login as `barman` on the backup server. If a port other than the SSH default of 22 should be used then the `--port` option can be added to specify the port that should be used for the SSH connection.

You can check that `barman-wal-restore` can connect to the Barman server, and that the required PostgreSQL server is configured in Barman to send WAL files with the following command:

```
barman-wal-restore --test backup pg DUMMY DUMMY
```

Where `backup` is the host where Barman is installed, `pg` is the name of the PostgreSQL server as configured in Barman and DUMMY is a placeholder (`barman-wal-restore` requires two argument for the WAL file name and destination directory, which are ignored).

If everything is configured correctly you should see the following output:

```
Ready to retrieve WAL files from the server pg
```

For more information on the `barman-wal-restore` command, type `man barman-wal-restore` on the PostgreSQL server.

`list-backups`

You can list the catalog of available backups for a given server with:

```
barman list-backups <server_name>
```

> **TIP:** You can request a full list of the backups of all servers using `all` as the server name.

To get a machine-readable output you can use the `--minimal` option, and to get the output in JSON format you can use the `--format=json` option.

rebuild-xlogdb

At any time, you can regenerate the content of the WAL archive for a specific server (or every server, using the `all` shortcut). The WAL archive is contained in the `xlog.db` file and every server managed by Barman has its own copy.

The `xlog.db` file can be rebuilt with the `rebuild-xlogdb` command. This will scan all the archived WAL files and regenerate the metadata for the archive.

For example:

```
barman rebuild-xlogdb <server_name>
```

receive-wal

This command manages the `receive-wal` process, which uses the streaming protocol to receive WAL files from the PostgreSQL streaming connection.

receive-wal process management

If the command is run without options, a `receive-wal` process will be started. This command is based on the `pg_receivewal` PostgreSQL command.

```
barman receive-wal <server_name>
```

> **NOTE:** The `receive-wal` command is a foreground process.

If the command is run with the `--stop` option, the currently running `receive-wal` process will be stopped.

The `receive-wal` process uses a status file to track last written record of the transaction log. When the status file needs to be cleaned, the `--reset` option can be used.

> **IMPORTANT:** If you are not using replication slots, you rely on the value of `wal_keep_segments` (or `wal_keep_size` from PostgreSQL version 13.0 onwards). Be aware that under high peaks of workload on the database, the `receive-wal` process might fall behind and go out of sync. As a precautionary measure, Barman currently requires that users manually execute the command with the `--reset` option, to avoid making wrong assumptions.

Replication slot management

The `receive-wal` process is also useful to create or drop the replication slot needed by Barman for its WAL archiving procedure.

With the `--create-slot` option, the replication slot named after the `slot_name` configuration option will be created on the PostgreSQL server.

With the `--drop-slot`, the previous replication slot will be deleted.

`replication-status`

The `replication-status` command reports the status of any streaming client currently attached to the PostgreSQL server, including the `receive-wal` process of your Barman server (if configured).

You can execute the command as follows:

```
barman replication-status <server_name>
```

> **TIP:** You can request a full status report of the replica for all your servers using `all` as the server name.

To have a machine-readable output you can use the `--minimal` option.

`show-servers`

You can show the configuration parameters for a given server with:

```
barman show-servers <server_name>
```

> **TIP:** you can request a full configuration report using `all` as the server name.

`status`

The `status` command shows live information and status of a PostgreSQL server or of all servers if you use `all` as server name.

```
barman status <server_name>
```

`switch-wal`

This command makes the PostgreSQL server switch to another transaction log file (WAL), allowing the current log file to be closed, received and then archived.

```
barman switch-wal <server_name>
```

If there has been no transaction activity since the last transaction log file switch, the switch needs to be forced using the `--force` option.

The `--archive` option requests Barman to trigger WAL archiving after the xlog switch. By default, a 30 seconds timeout is enforced (this can be changed with `--archive-timeout`). If no WAL file is received, an error is returned.

> **NOTE:** In Barman 2.1 and 2.2 this command was called `switch-xlog`. It has been renamed for naming consistency with PostgreSQL 10 and higher.

`verify-backup`

The `verify-backup` command uses backup_manifest file from backup and runs `pg_verifybackup` against it.

```
barman verify-backup <server_name> <backup_id>
```

This command will call `pg_verifybackup <path_to_backup_manifest> -n` (available on PG>=13) `pg_verifybackup` Must be installed on backup server. For rsync backups, it can be used with `generate-manifest` command.

Either backup_id backup id shortcuts can be used.

Backup commands

Backup commands are those that works directly on backups already existing in Barman's backup catalog.

> **NOTE:** Remember a backup ID can be retrieved with `barman list-backups <server_name>`

Backup ID shortcuts

Barman allows you to use special keywords to identify a specific backup:

- `last/latest`: identifies the newest backup in the catalog
- `first/oldest`: identifies the oldest backup in the catalog
- `last-failed`: identifies the newest failed backup in the catalog

Using those keywords with Barman commands allows you to execute actions without knowing the exact ID of a backup for a server. For example we can issue:

```
barman delete <server_name> oldest
```

to remove the oldest backup available in the catalog and reclaim disk space.

Additionally, if backup was taken with the `--name <friendly_name>` option, you can use the friendly name in place of the backup ID to refer to that specific backup.

`check-backup`

Starting with version 2.5, you can check that all required WAL files for the consistency of a full backup have been correctly archived by `barman` with the `check-backup` command:

```
barman check-backup <server_name> <backup_id>
```

> **IMPORTANT:** This command is automatically invoked by `cron` and at the end of a `backup` operation. This means that, under normal circumstances, you should never need to execute it.

In case one or more WAL files from the start to the end of the backup have not been archived yet, `barman` will label the backup as `WAITING_FOR_WALS`. The `cron` command will continue to check that missing WAL files are archived, then label the backup as `DONE`.

In case the first required WAL file is missing at the end of the backup, such backup will be marked as `FAILED`. It is therefore important that you verify that WAL archiving (whether via streaming or `archive_command`) is properly working before executing a backup operation - especially when backing up from a standby server.

`delete`

You can delete a given backup with:

```
barman delete <server_name> <backup_id>
```

The `delete` command accepts any shortcut to identify backups.

> **IMPORTANT:** If the specified backup has dependent block-level incremental backups, those backups and all their dependents will also be deleted during this operation as they would effectively become unusable for recovery with a missing parent in its chain.

`keep`

If you have a backup which you wish to keep beyond the retention policy of the server then you can make it an archival backup with:

```
barman keep <server_name> <backup_id> [--target TARGET, --status, --release]
```

> **NOTE:** To ensure the integrity of your backup system, block-level incremental backups cannot use the keep annotation in Barman. This restriction is due to the way block-level incremental backups depend on each other. Using the keep annotation on such backups could result in orphaned backups, which means that certain backups might exist without their necessary parent backups.
>
> In simpler terms, if you were allowed to apply the keep annotation to a block-level incremental backup, there would be a risk that parts of the backup chain would be retained without their required predecessors. This situation could create backups that would be no longer be useful or complete, as they would be missing the essential parent backups needed to restore them properly.

Possible values for `TARGET` are:

- `full`: The backup can always be used to recover to the latest point in time. To achieve this, Barman will retain all WALs needed to ensure consistency of the backup and all subsequent WALs.
- `standalone`: The backup can only be used to recover the server to its state at the time the backup was taken. Barman will only retain the WALs needed to ensure consistency of the backup.

If the `--status` option is provided then Barman will report the archival status of the backup. This will either be the recovery target of `full` or `standalone` for archival backups or `nokeep` for backups which have not been flagged as archival.

If the `--release` option is provided then Barman will release the keep flag from this backup. This will remove its archival status and make it available for deletion, either directly or by retention policy.

Once a backup has been flagged as an archival backup, the behaviour of Barman will change as follows:

- Attempts to delete that backup by ID using `barman delete` will fail.
- Retention policies will never consider that backup as `OBSOLETE` and therefore `barman cron` will never delete that backup.
- The WALs required by that backup will be retained forever. If the specified recovery target is `full` then *all* subsequent WALs will also be retained.

This can be reverted by removing the keep flag with `barman keep <server_name> <backup_id> --release`.

> **WARNING:** Once a `standalone` archival backup is not required by the retention policy of a server `barman cron` will remove the WALs between that backup and the begin_wal value of the next most recent backup. This means that while it is safe to change the target from `full` to `standalone`, it is *not* safe to change the target from `standalone` to `full` because there is no guarantee the necessary WALs for a recovery to the latest point in time will still be available.

`list-files`

You can list the files (base backup and required WAL files) for a given backup with:

```
barman list-files [--target TARGET_TYPE] <server_name> <backup_id>
```

With the `--target TARGET_TYPE` option, it is possible to choose the content of the list for a given backup.

Possible values for `TARGET_TYPE` are:

- `data`: lists the data files
- `standalone`: lists the base backup files, including required WAL files
- `wal`: lists all WAL files from the beginning of the base backup to the start of the following one (or until the end of the log)
- `full`: same as `data` + `wal`

The default value for `TARGET_TYPE` is `standalone`.

> **IMPORTANT:** The `list-files` command facilitates interaction with external tools, and can therefore be extremely useful to integrate Barman into your archiving procedures.

recover

The `recover` command is used to recover a whole server after a backup is executed using the `backup` command.

This is achieved issuing a command like the following:

```
barman@backup$ barman recover <server_name> <backup_id> /path/to/recover/dir
```

> **IMPORTANT:** Do not issue a `recover` command using a target data directory where a PostgreSQL instance is running. In that case, remember to stop it before issuing the recovery. This applies also to tablespace directories.

At the end of the execution of the recovery, the selected backup is recovered locally and the destination path contains a data directory ready to be used to start a PostgreSQL instance.

> **IMPORTANT:** Running this command as user `barman`, it will become the database superuser.

The specific ID of a backup can be retrieved using the list-backups command.

> **IMPORTANT:** Barman does not currently keep track of symbolic links inside PGDATA (except for tablespaces inside pg_tblspc). We encourage system administrators to keep track of symbolic links and to add them to the disaster recovery plans/procedures in case they need to be restored in their original location.

The recovery command has several options that modify the command behavior.

Remote recovery

Add the `--remote-ssh-command <COMMAND>` option to the invocation of the recovery command. Doing this will allow Barman to execute the copy on a remote server, using the provided command to connect to the remote host.

> **NOTE:** It is advisable to use the `postgres` user to perform the recovery on the remote host.

> **IMPORTANT:** Do not issue a `recover` command using a target data directory where a PostgreSQL instance is running. In that case, remember to stop it before issuing the recovery. This applies also to tablespace directories.

Known limitations of the remote recovery are:

- Barman requires at least 4GB of free space in the system temporary directory unless the `get-wal` command is specified in the `recovery_option` parameter in the Barman configuration.
- The SSH connection between Barman and the remote host **must** use the public key exchange authentication method
- The remote user **must** be able to create the directory structure of the backup in the destination directory.
- There must be enough free space on the remote server to contain the base backup and the WAL files needed for recovery.

Tablespace remapping

Barman is able to automatically remap one or more tablespaces using the recover command with the --tablespace option. The option accepts a pair of values as arguments using the `NAME:DIRECTORY` format:

- `NAME` is the identifier of the tablespace
- `DIRECTORY` is the new destination path for the tablespace

If the destination directory does not exists, Barman will try to create it (assuming you have the required permissions).

Point in time recovery

Barman wraps PostgreSQL's Point-in-Time Recovery (PITR), allowing you to specify a recovery target, either as a timestamp, as a restore label, or as a transaction ID.

> **IMPORTANT:** The earliest PITR for a given backup is the end of the base backup itself. If you want to recover at any point in time between the start and the end of a backup, you must use the previous backup. From Barman 2.3 you can exit recovery when consistency is reached by using `--target-immediate` option.

The recovery target can be specified using one of the following mutually exclusive options:

- `--target-time TARGET_TIME`: to specify a timestamp
- `--target-xid TARGET_XID`: to specify a transaction ID
- `--target-lsn TARGET_LSN`: to specify a Log Sequence Number (LSN) - requires PostgreSQL 10 or higher
- `--target-name TARGET_NAME`: to specify a named restore point previously created with the pg_create_restore_point(name) function
- `--target-immediate`: recovery ends when a consistent state is reached (that is the end of the base backup process)

> **IMPORTANT:** Recovery target via *time*, *XID* and LSN **must be** subsequent to the end of the backup. If you want to recover to a point in time between the start and the end of a backup, you must recover from the previous backup in the catalogue.

> **IMPORTANT:** If no timezone is specified when using `--target-time`, the timezone of the Barman host will be used.

You can use the `--exclusive` option to specify whether to stop immediately before or immediately after the recovery target.

Barman allows you to specify a target timeline for recovery using the `--target-tli` option. This can be set to a numeric timeline ID or one of the special values `latest` (to recover to the most recent timeline in the WAL archive) and `current` (to recover to the timeline which was current when the backup was taken). If this option is omitted then PostgreSQL versions 12 and above will recover to the `latest` timeline and PostgreSQL versions below 12 will recover to the `current` timeline. You can find more details about timelines in the PostgreSQL documentation as mentioned in the *"Before you start"* section.

Barman 2.4 introduces support for `--target-action` option, accepting the following values:

- `shutdown`: once recovery target is reached, PostgreSQL is shut down
- `pause`: once recovery target is reached, PostgreSQL is started in pause state, allowing users to inspect the instance
- `promote`: once recovery target is reached, PostgreSQL will exit recovery and is promoted as a master

> **IMPORTANT:** By default, no target action is defined (for back compatibility). The `--target-action` option requires a Point In Time Recovery target to be specified.

For more detailed information on the above settings, please consult the PostgreSQL documentation on recovery target settings.

Barman 2.4 also adds the `--standby-mode` option for the `recover` command which, if specified, properly configures the recovered instance as a standby by creating a `standby.signal` file (from PostgreSQL versions lower than 12), or by adding `standby_mode = on` to the generated recovery configuration.

Further information on Postgresql *standby mode* is available in the official documentation:

- For Postgres 11 and lower versions in the standby section of PostgreSQL documentation.
- For PostgreSQL 12 and greater versions in the replication section of PostgreSQL documentation.

> **IMPORTANT** When `--standby-mode` is used during recovery is necessary for the user to modify the configuration of the recovered instance, allowing the recovered server to connect to the primary once the WAL files replication from Barman is successfully completed. If the recovered instance version is 11 or lower this is achieved by adding the `primary_conninfo` parameter to the `recovery.conf` file. If the recovered instance version is 12 or greater, the `primary_conninfo` parameter needs to be added to the `postgresql.conf` file.

Fetching WALs from the Barman server

The `barman recover` command can optionally configure PostgreSQL to fetch WALs from Barman during recovery. This is enabled by setting the `recovery_options` global/server configuration option to `'get-wal'` as described in the get-wal section. If `recovery_options` is not set or is empty then Barman will instead copy the WALs required for recovery while executing the `barman recover` command.

The `--get-wal` and `--no-get-wal` options can be used to override the behaviour defined by `recovery_options`. Use `--get-wal` with `barman recover` to enable the fetching of WALs from the Barman server, alternatively use `--no-get-wal` to disable it.

> **IMPORTANT:** When recovering with `--no-get-wal` in conjunction with any of these targets [`--target-xid`, `--target-name`, `--target-time`], Barman will copy the whole WAL archive from the Barman host to the recovery host. By doing that, and assuming that all the WALs required for reaching the configured target were already archived into Barman, we guarantee that at least these WALs will be made available to Postgres. This happens because currently there is no reliable and/or performant way of determining in Barman which WALs are needed by Postgres to reach those kinds of recovery targets.

Recovering compressed backups

If a backup has been compressed using the `backup_compression` option then `barman recover` is able to uncompress the backup on recovery. This is a multi-step process:

1. The compressed backup files are copied to a staging directory on the local or remote server using Rsync.
2. The compressed files are uncompressed to the target directory.
3. Config files which need special handling by Barman are copied from the recovery destination, analysed or edited as required, and copied back to the recovery destination using Rsync.
4. The staging directory for the backup is removed.

Because barman does not know anything about the environment in which it will be deployed it relies on the `recovery_staging_path` option in order to choose a suitable location for the staging directory.

If you are using the `backup_compression` option you *must* therefore either set `recovery_staging_path` in the global/server config *or* use the `--recovery-staging-path` option with the `barman recover` command. If you do neither of these things and attempt to recover a compressed backup then Barman will fail rather than try to guess a suitable location.

Recovering block-level incremental backups

If a backup is a block-level incremental, `barman recover` is able to combine the chain of backups on recovery through `pg_combinebackup`. A chain of backups is the tree branch that goes from the full backup to the one requested for the recovery. This is a multi-step process:

1. The chain of backups is combined into a new synthetic backup. A folder named with the ID of the incremental backup being recovered is created inside a given staging directory on the local server using `pg_combinebackup`. For any type of recover (local or remote), the synthetic backup is created locally in the barman server.
2. If it's a remote recover, the content is copied to the final destination using Rsync. Otherwise, when it's a local recover, the content is just moved to the final destination.
3. The folder named with the ID of the incremental backup being recovered, which was created inside the provided staging directory, is removed at the end of the recovery process.

When recovering from a block-level incremental backup, you *must* therefore either set `local_staging_path` in the global/server config *or* use the `--local-staging-path` option with the `barman recover` command. If you do neither of these things and attempt to recover such backup then Barman fails rather than trying to guess a suitable location.

> **IMPORTANT:** If any of the backups in the chain were taken with checksums disabled, but the final backup was taken with checksums enabled, the resulting directory may contain pages with invalid checksums. Follow up the `limitations` section in pg_basebackup documentation.

`show-backup`

You can retrieve all the available information for a particular backup of a given server with:

```
barman show-backup <server_name> <backup_id>
```

The `show-backup` command accepts any shortcut to identify backups.

Features in detail

In this section we present several Barman features and discuss their applicability and the configuration required to use them.

This list is not exhaustive, as many scenarios can be created working on the Barman configuration. Nevertheless, it is useful to discuss common patterns.

Backup features

Incremental backup

Incremental backup is a type of backup which uses an already existing backup as reference for copying only necessary data changes from the PostgreSQL server. It must not be confused with differential backup, which is implemented by *WAL continuous archiving*.

The main goals of incremental backups in Barman are:

- Reduce the time taken for the full backup process
- Reduce the disk space occupied by several periodic backups (**data deduplication**)

Barman currently supports **file-level incremental backups** (using `rsync`) as well as **block-level incremental backups** (using `pg_basebackup`).

> **NOTE:** Incremental backups of different backup types are currently not compatible i.e. a block-level incremental backup can not be taken upon an `rsync` backup and a file-level incremental backup can not be taken upon a streaming backup (taken with `pg_basebackup`).

File-level incremental backups

This feature heavily relies on `rsync` and hard links, which must therefore be supported by both the underlying operating system and the file system where the backup data resides.

The main concept is that a subsequent base backup will share those files that have not changed since the previous backup, leading to relevant savings in disk usage. This is particularly true of VLDB contexts and of those databases containing a high percentage of *read-only historical tables*.

Rsync incremental backups can be enabled through a global/server option called `reuse_backup`, that transparently manages the `barman backup` command. It accepts three values:

- `off`: standard full backup (default)
- `link`: file-level incremental backup, by reusing the last backup for a server and creating a hard link of the unchanged files (for backup space and time reduction)

- `copy`: file-level incremental backup, by reusing the last backup for a server and creating a copy of the unchanged files (just for backup time reduction)

The most common scenario is to set `reuse_backup` to `link`, as follows:

```
reuse_backup = link
```

Setting this at global level will automatically enable incremental backup for all your servers.

As a final note, users can override the setting of the `reuse_backup` option through the `--reuse-backup` runtime option for the `barman backup` command. Similarly, the runtime option accepts three values: `off`, `link` and `copy`. For example, you can run a one-off incremental backup as follows:

```
barman backup --reuse-backup=link <server_name>
```

> **NOTE:** Unlike Postgres block-level incremental backups, Rsync file-level incremental backups are independent on their own, meaning that a backup that reused a previous backup for deduplication is not compromised in any way if the parent backup is deleted. As mentioned, deduplication in Rsync backups is implemented with the use of hard links, so when a previously reused backup is deleted, files shared with other backups will still remain on disk, only being removed when the last backup using those files is also deleted. It also means that there's no need to take full backups when a previous full backup is deleted, each backup will still have all files and can be reused without any concerns. Along the same lines, there is no need to ever take a full backup with rsync by using `reuse_backup = off`. If it is the first backup being taken with `reuse_backup = link`, in essence it behaves like `off` because there are no existing files to create hard-links on.

> **IMPORTANT:** The `reuse_backup` option must be used along with `rsync` or `local-rsync` as backup method.

Block-level incremental backups

Since version 3.11, Barman introduces support for block-level incremental backups, leveraging the native incremental backup support introduced in PostgreSQL 17.

With block-level incremental backups, deduplication occurs at the data block level (pages in PostgreSQL). This means that only those pages modified since the last backup will need to be stored, making it a more efficient option, especially for large databases with spread write patterns. In PostgreSQL, this feature is implemented with the use of WAL Summarization, therefore `summarize_wal` must be enabled on your database server in order to use it.

You can perform block-level incremental backups in Barman using the `--incremental` option when running a backup command. It accepts a backup id or backup ID shortcut as argument, which references a previous backup (full or incremental) in the catalog to be used as a parent for deduplication. In addition, you can also use `last-full` or `latest-full` to reference the latest eligible full-backup in the catalog.

```
barman backup --incremental <backup_id> <server_name>
```

To be able to perform a block-level incremental backup in Barman you must:

- Have PostgreSQL 17 or later.
- Have `summarize_wal` enabled.
- Have `postgres` as your backup method.

> **NOTE:** Compressed backups are not **yet** eligible for block-level incremental backups in Barman.

> **IMPORTANT:** If you decide to enable `data_checksums` between block-level incremental backups, it is adivised to take a new full-backup as divergent checkum configurations can potentially lead to issues during recovery.

Limiting bandwidth usage

It is possible to limit the usage of I/O bandwidth through the `bandwidth_limit` option (global/per server), by specifying the maximum number of kilobytes per second. By default it is set to 0, meaning no limit.

> **IMPORTANT:** the `bandwidth_limit` option is supported with the `postgres` backup method, but the `tablespace_bandwidth_limit` option is available only if you use `rsync`.

In case you have several tablespaces and you prefer to limit the I/O workload of your backup procedures on one or more tablespaces, you can use the `tablespace_bandwidth_limit` option (global/per server):

```
tablespace_bandwidth_limit = tbname:bwlimit[, tbname:bwlimit, ...]
```

The option accepts a comma separated list of pairs made up of the tablespace name and the bandwidth limit (in kilobytes per second).

When backing up a server, Barman will try and locate any existing tablespace in the above option. If found, the specified bandwidth limit will be enforced. If not, the default bandwidth limit for that server will be applied.

Network Compression

It is possible to reduce the size of transferred data using compression. It can be enabled using the `network_compression` option (global/per server):

> **IMPORTANT:** the `network_compression` option is not available with the `postgres` backup method.

```
network_compression = true|false
```

Setting this option to `true` will enable data compression during network transfers (for both backup and recovery). By default it is set to `false`.

Backup Compression

Barman can use the compression features of pg_basebackup in order to compress the backup data during the backup process. This can be enabled using the `backup_compression` config option (global/per server):

> **IMPORTANT:** the `backup_compression` and other options discussed in this section are not available with the `rsync` or `local-rsync` backup methods. Only with `postgres` backup method.

Compression algorithms

Setting this option will cause pg_basebackup to compress the backup using the specified compression algorithm. Currently, supported algorithm in Barman are: `gzip`, `lz4`,`zstd` and `none`. `none` compression algorithm will create an uncompressed archive.

```
backup_compression = gzip|lz4|zstd|none
```

Barman requires the CLI utility for the selected compression algorithm to be available on both the Barman server *and* the PostgreSQL server. The CLI utility is used to extract the backup label from the compressed backup and to decompress the backup on the PostgreSQL server during recovery. These can be installed through system packages named `gzip`, `lz4` and `zstd` on Debian, Ubuntu, RedHat, CentOS and SLES systems.

> **Note:** On Ubuntu 18.04 (bionic) the `lz4` utility is available in the `liblz4-tool` pacakge.

> **Note:** `zstd` version must be 1.4.4 or higher. The system packages for `zstd` on Debian 10 (buster), Ubuntu 18.04 (bionic) and SLES 12 install an earlier version - `backup_compression = zstd` will not work with these packages.

> **Note:** `lz4` and `zstd` are only available with PostgreSQL version 15 or higher.

> **IMPORTANT:** If you are using `backup_compression` you must also set `recovery_staging_path` so that `barman recover` is able to recover the compressed backups. See the Recovering compressed backups section for more information.

Compression workers

This optional parameter allows compression using multiple threads to increase compression speed (default being 0).

```
backup_compression_workers = 2
```

> **Note:** This option is only available with `zstd` compression.

> **Note:** `zstd` version must be 1.5.0 or higher. Or 1.4.4 or higher compiled with multithreading option.

Compression level

The compression level can be specified using the `backup_compression_level` option. This should be set to an integer value supported by the compression algorithm specified in `backup_compression`. If not defined, compression algorithm default value will be used.

`none` compression only supports `backup_compression_level=0`.

> **Note:** `backup_compression_level` available and default values depends on the compression algorithm used. Please check the compression algorithm documentation for more details.

> **Note:** On PostgreSQL version prior to 15, `gzip` support `backup_compression_level=0`. It results using default compression level

Compression location

When using Barman with PostgreSQL version 15 or higher it is possible to specify for compression to happen on the server (i.e. PostgreSQL will compress the backup) or on the client (i.e. pg_basebackup will compress the backup). This can be achieved using the `backup_compression_location` option:

> **IMPORTANT:** the `backup_compression_location` option is only available when running against PostgreSQL 15 or later.

```
backup_compression_location = server|client
```

Using `backup_compression_location = server` should reduce the network bandwidth required by the backup at the cost of moving the compression work onto the PostgreSQL server.

When `backup_compression_location` is set to `server` then an additional option, `backup_compression_format`, can be set to `plain` in order to have pg_basebackup uncompress the data before writing it to disk:

Compression format

```
backup_compression_format = plain|tar
```

If `backup_compression_format` is unset or has the value `tar` then the backup will be written to disk as compressed tarballs. A description of both the `plain` and `tar` formats can be found in the pg_basebackup documentation.

> **IMPORTANT:** Barman uses external tools to manage compressed backups. Depending on the `backup_compression` and `backup_compression_format` You may need to install one or more tools on the Postgres server and the Barman server. The following table will help you choose according to your configuration.

backup_compression	backup_compression_format	Postgres server	Barman server
gzip	plain	**tar**	None
gzip	tar	**tar**	**tar**
lz4	plain	**tar, lz4**	None
lz4	tar	**tar, lz4**	**tar, lz4**
zstd	plain	**tar, zstd**	None
zstd	tar	**tar, zstd**	**tar, zstd**
none	tar	**tar**	**tar**

Concurrent backup

Normally, during backup operations, Barman uses PostgreSQL native functions `pg_start_backup` and `pg_stop_backup` for *concurrent backup*.[6] This is the recommended way of taking backups for PostgreSQL 9.6 and above (though note the functions have been renamed to `pg_backup_start` and `pg_backup_stop` in the PostgreSQL 15 beta).

As well as being the recommended backup approach, concurrent backup also allows the following architecture scenario with Barman: **backup from a standby server**, using `rsync`.

By default, `backup_options` is set to `concurrent_backup`. If exclusive backup is required for PostgreSQL servers older than version 15 then users should set `backup_options` to `exclusive_backup`.

When `backup_options` is set to `concurrent_backup`, Barman activates the *concurrent backup mode* for a server and follows these two simple rules:

- `ssh_command` must point to the destination Postgres server

[6] Concurrent backup is a technology that uses the *streaming replication protocol* (for example, using a tool like `pg_basebackup`).

- `conninfo` must point to a database on the destination Postgres database.

> **IMPORTANT:** In case of a concurrent backup, currently Barman cannot determine whether the closing WAL file of a full backup has actually been shipped - opposite of an exclusive backup where PostgreSQL itself makes sure that the WAL file is correctly archived. Be aware that the full backup cannot be considered consistent until that WAL file has been received and archived by Barman. Barman 2.5 introduces a new state, called `WAITING_FOR_WALS`, which is managed by the `check-backup` command (part of the ordinary maintenance job performed by the `cron` command). From Barman 2.10, you can use the `--wait` option with `barman backup` command.

Concurrent backup of a standby

If backing up a standby then the following configuration options should point to the standby server:

- `conninfo`
- `streaming_conninfo` (when using `backup_method = postgres` or `streaming_archiver = on`)
- `ssh_command` (when using `backup_method = rsync`)

The following config option should point to the primary server:

- `primary_conninfo`

Barman will use `primary_conninfo` to switch to a new WAL on the primary so that the concurrent backup against the standby can complete without having to wait for a WAL switch to occur naturally.

> **NOTE:** It is especially important that `primary_conninfo` is set if the standby is to be backed up when there is little or no write traffic on the primary.

As of Barman 3.8.0, If `primary_conninfo` is set, is possible to add for a server a `primary_checkpoint_timeout` option, which is the maximum time (in seconds) for Barman to wait for a new WAL file to be produced before forcing the execution of a checkpoint on the primary. The `primary_checkpoint_timeout` option should be set to an amount of seconds greater of the value of the `archive_timeout` option set on the primary server.

If `primary_conninfo` is not set then the backup will still run however it will wait at the stop backup stage until the current WAL segment on the primary is newer than the latest WAL required by the backup.

Barman currently requires that WAL files and backup data come from the same PostgreSQL server. In the case that the standby is promoted to primary the backups and WALs will continue to be valid however you may wish to update the Barman configuration so that it uses the new standby for taking backups and receiving WALs.

WALs can be obtained from the standby using either WAL streaming or WAL archiving. To use WAL streaming follow the instructions in the WAL streaming section.

To use WAL archiving from the standby follow the instructions in the WAL archiving via archive_command section *and additionally* set `archive_mode = always` in the PostgreSQL config on the standby server.

> **NOTE:** With PostgreSQL 10 and earlier Barman cannot handle WAL streaming and WAL archiving being enabled at the same time on a standby. You must therefore disable WAL archiving if using WAL streaming and vice versa. This is because it is possible for WALs produced by PostgreSQL 10 and earlier to be logically equivalent but differ at the binary level, causing Barman to fail to detect that two WALs are identical.

Immediate checkpoint

Before starting a backup, Barman requests a checkpoint, which generates additional workload. Normally that checkpoint is throttled according to the settings for workload control on the PostgreSQL server, which means that the backup could be delayed.

This default behaviour can be changed through the `immediate_checkpoint` configuration global/server option (set to `false` by default).

If `immediate_checkpoint` is set to `true`, PostgreSQL will not try to limit the workload, and the checkpoint will happen at maximum speed, starting the backup as soon as possible.

At any time, you can override the configuration option behaviour, by issuing `barman backup` with any of these two options:

- `--immediate-checkpoint`, which forces an immediate checkpoint;
- `--no-immediate-checkpoint`, which forces to wait for the checkpoint to happen.

Local backup

> **DISCLAIMER:** This feature is not recommended for production usage, as Barman and PostgreSQL reside on the same server and are part of the same single point of failure. Some EnterpriseDB customers have requested to add support for local backup to Barman to be used under specific circumstances and, most importantly, under the 24/7 production service delivered by the company. Using this feature currently requires installation from sources, or to customise the environment for the `postgres` user in terms of permissions as well as logging and cron configurations.

Under special circumstances, Barman can be installed on the same server where the PostgreSQL instance resides, with backup data stored on a separate volume from PGDATA and, where applicable, tablespaces. Usually, these volumes reside on network storage appliances, with filesystems like NFS.

This architecture is not endorsed by EnterpriseDB. For an enhanced business continuity experience of PostgreSQL, with better results in terms of RPO and RTO, EnterpriseDB still recommends the shared nothing architecture with a remote installation of Barman, capable of acting like a witness server for replication and monitoring purposes.

The only requirement for local backup is that Barman runs with the same user as the PostgreSQL server, which is normally `postgres`. Given that the Community packages by default install Barman under the `barman` user, this use case requires manual installation procedures that include:

- cron configurations
- log configurations, including logrotate

In order to use local backup for a given server in Barman, you need to set `backup_method` to `local-rsync`. The feature is essentially identical to its `rsync` equivalent, which relies on SSH instead and operates remotely. With `local-rsync` file system copy is performed issuing `rsync` commands locally (for this reason it is required that Barman runs with the same user as PostgreSQL).

An excerpt of configuration for local backup for a server named `local-pg13` is:

```
[local-pg13]
description = "Local PostgreSQL 13"
backup_method = local-rsync
...
```

Archiving features

WAL compression

The `barman cron` command will compress WAL files if the `compression` option is set in the configuration file. This option allows five values:

- `bzip2`: for Bzip2 compression (requires the `bzip2` utility)
- `gzip`: for Gzip compression (requires the `gzip` utility)
- `pybzip2`: for Bzip2 compression (uses Python's internal compression module)
- `pygzip`: for Gzip compression (uses Python's internal compression module)
- `pigz`: for Pigz compression (requires the `pigz` utility)
- `custom`: for custom compression, which requires you to set the following options as well: - `custom_compression_filter`: a compression filter - `custom_decompression_filter`: a decompression filter - `custom_compression_magic`: a hex string to identify a custom compressed wal file

  NOTE: All methods but `pybzip2` and `pygzip` require `barman archive-wal` to fork a new process.

Synchronous WAL streaming

Barman can also reduce the Recovery Point Objective to zero, by collecting the transaction WAL files like a synchronous standby server would.

To configure such a scenario, the Barman server must be configured to archive WALs via the streaming connection, and the `receive-wal` process should figure as a synchronous standby of the PostgreSQL server.

First of all, you need to retrieve the application name of the Barman `receive-wal` process with the `show-servers` command:

```
barman@backup$ barman show-servers pg|grep streaming_archiver_name
 streaming_archiver_name: barman_receive_wal
```

Then the application name should be added to the `postgresql.conf` file as a synchronous standby:

```
synchronous_standby_names = 'barman_receive_wal'
```

> **IMPORTANT:** this is only an example of configuration, to show you that Barman is eligible to be a synchronous standby node. We are not suggesting to use ONLY Barman. You can read *"Synchronous Replication"* from the PostgreSQL documentation for further information on this topic.

The PostgreSQL server needs to be restarted for the configuration to be reloaded.

If the server has been configured correctly, the `replication-status` command should show the `receive_wal` process as a synchronous streaming client:

```
[root@backup ~]# barman replication-status pg
Status of streaming clients for server 'pg':
  Current xlog location on master: 0/9000098
  Number of streaming clients: 1

  1. #1 Sync WAL streamer
     Application name: barman_receive_wal
     Sync stage      : 3/3 Remote write
     Communication   : TCP/IP
     IP Address      : 139.59.135.32 / Port: 58262 / Host: -
     User name       : streaming_barman
     Current state   : streaming (sync)
     Replication slot: barman
     WAL sender PID  : 2501
     Started at      : 2016-09-16 10:33:01.725883+00:00
     Sent location   : 0/9000098 (diff: 0 B)
     Write location  : 0/9000098 (diff: 0 B)
     Flush location  : 0/9000098 (diff: 0 B)
```

Catalog management features

Minimum redundancy safety

You can define the minimum number of periodic backups for a PostgreSQL server, using the global/per server configuration option called `minimum_redundancy`, by default set to 0.

By setting this value to any number greater than 0, Barman makes sure that at any time you will have at least that number of backups in a server catalog.

This will protect you from accidental `barman delete` operations.

> **IMPORTANT:** Make sure that your retention policy settings do not collide with minimum redundancy requirements. Regularly check Barman's log for messages on this topic.

Retention policies

Barman supports **retention policies** for backups.

A backup retention policy is a user-defined policy that determines how long backups and related archive logs (Write Ahead Log segments) need to be retained for recovery procedures.

Based on the user's request, Barman retains the periodic backups required to satisfy the current retention policy and any archived WAL files required for the complete recovery of those backups.

Barman users can define a retention policy in terms of **backup redundancy** (how many periodic backups) or a **recovery window** (how long).

Retention policy based on redundancy In a redundancy based retention policy, the user determines how many periodic backups to keep. A redundancy-based retention policy is contrasted with retention policies that use a recovery window.

Retention policy based on recovery window A recovery window is one type of Barman backup retention policy, in which the DBA specifies a period of time and Barman ensures retention of backups and/or archived WAL files required for point-in-time recovery to any time during the recovery window. The interval always ends with the current time and extends back in time for the number of days specified by the user. For example, if the retention policy is set for a recovery window of seven days, and the current time is 9:30 AM on Friday, Barman retains the backups required to allow point-in-time recovery back to 9:30 AM on the previous Friday.

> **IMPORTANT:** Block-level incremental backups are not considered during the retention policy processing. This is because this kind of incremental backups depends on all of its parent backups, up to the full backup which generates the chain, in order to be recoverable. To maintain the consistency of backup chains, only full backups are taken into account when applying retention policies.
>
> How It Works

When the retention policy is applied, Barman ignores block-level incremental backups and focuses only on the status of the full backups.

If the full backup is marked as KEEP:FULL, KEEP:STANDALONE, or VALID, the status VALID is marked to all dependent block-level incremental backups. If the full backup is marked as OBSOLETE, then all block-level incremental backups that depend on it will also be marked as OBSOLETE and removed.

Scope

Retention policies can be defined for:

- **PostgreSQL periodic base backups**: through the `retention_policy` configuration option
- **Archive logs**, for Point-In-Time-Recovery: through the `wal_retention_policy` configuration option

IMPORTANT: In a temporal dimension, archive logs must be included in the time window of periodic backups.

There are two typical use cases here: full or partial point-in-time recovery.

Full point in time recovery scenario: Base backups and archive logs share the same retention policy, allowing you to recover at any point in time from the first available backup.

Partial point in time recovery scenario: Base backup retention policy is wider than that of archive logs, for example allowing users to keep full, weekly backups of the last 6 months, but archive logs for the last 4 weeks (granting to recover at any point in time starting from the last 4 periodic weekly backups).

IMPORTANT: Currently, Barman implements only the **full point in time recovery** scenario, by constraining the `wal_retention_policy` option to `main`.

How they work

Retention policies in Barman can be:

- **automated**: enforced by `barman cron`
- **manual**: Barman simply reports obsolete backups and allows you to delete them

IMPORTANT: Currently Barman does not implement manual enforcement. This feature will be available in future versions.

Configuration and syntax

Retention policies can be defined through the following configuration options:

- `retention_policy`: for base backup retention
- `wal_retention_policy`: for archive logs retention
- `retention_policy_mode`: can only be set to `auto` (retention policies are automatically enforced by the `barman cron` command)

These configuration options can be defined both at a global level and a server level, allowing users maximum flexibility on a multi-server environment.

Syntax for `retention_policy`

The general syntax for a base backup retention policy through `retention_policy` is the following:

```
retention_policy = {REDUNDANCY value | RECOVERY WINDOW OF value {DAYS | WEEKS | MONTHS}}
```

Where:

- syntax is case insensitive
- `value` is an integer and is > 0
- in case of **redundancy retention policy**: - `value` must be greater than or equal to the server minimum redundancy level (if that value is not assigned, a warning is generated) - the first valid backup is the value-th backup in a reverse ordered time series
- in case of **recovery window policy**: - the point of recoverability is: current time - window - the first valid backup is the first available backup before the point of recoverability; its value in a reverse ordered time series must be greater than or equal to the server minimum redundancy level (if it is not assigned to that value and a warning is generated)

By default, `retention_policy` is empty (no retention enforced).

Syntax for `wal_retention_policy`

Currently, the only allowed value for `wal_retention_policy` is the special value `main`, that maps the retention policy of archive logs to that of base backups.

Hook scripts

Barman allows a database administrator to run hook scripts on these two events:

- before and after a backup

- before and after the deletion of a backup
- before and after a WAL file is archived
- before and after a WAL file is deleted

There are two types of hook scripts that Barman can manage:

- standard hook scripts
- retry hook scripts

The only difference between these two types of hook scripts is that Barman executes a standard hook script only once, without checking its return code, whereas a retry hook script may be executed more than once, depending on its return code.

Specifically, when executing a retry hook script, Barman checks the return code and retries indefinitely until the script returns either `SUCCESS` (with standard return code `0`), or `ABORT_CONTINUE` (return code `62`), or `ABORT_STOP` (return code `63`). Barman treats any other return code as a transient failure to be retried. Users are given more power: a hook script can control its workflow by specifying whether a failure is transient. Also, in case of a 'pre' hook script, by returning `ABORT_STOP`, users can request Barman to interrupt the main operation with a failure.

Hook scripts are executed in the following order:

1. The standard 'pre' hook script (if present)
2. The retry 'pre' hook script (if present)
3. The actual event (i.e. backup operation, or WAL archiving), if retry 'pre' hook script was not aborted with `ABORT_STOP`
4. The retry 'post' hook script (if present)
5. The standard 'post' hook script (if present)

The output generated by any hook script is written in the log file of Barman.

> **NOTE:** Currently, `ABORT_STOP` is ignored by retry 'post' hook scripts. In these cases, apart from logging an additional warning, `ABORT_STOP` will behave like `ABORT_CONTINUE`.

Backup scripts

These scripts can be configured with the following global configuration options (which can be overridden on a per server basis):

- `pre_backup_script`: *hook script* executed *before* a base backup, only once, with no check on the exit code
- `pre_backup_retry_script`: *retry hook script* executed *before* a base backup, repeatedly until success or abort

- `post_backup_retry_script`: *retry hook script* executed *after* a base backup, repeatedly until success or abort
- `post_backup_script`: *hook script* executed *after* a base backup, only once, with no check on the exit code

The script definition is passed to a shell and can return any exit code. Only in case of a *retry* script, Barman checks the return code (see the hook script section).

The shell environment will contain the following variables:

- `BARMAN_BACKUP_DIR`: backup destination directory
- `BARMAN_BACKUP_ID`: ID of the backup
- `BARMAN_CONFIGURATION`: configuration file used by Barman
- `BARMAN_ERROR`: error message, if any (only for the `post` phase)
- `BARMAN_PHASE`: phase of the script, either `pre` or `post`
- `BARMAN_PREVIOUS_ID`: ID of the previous backup (if present)
- `BARMAN_RETRY`: `1` if it is a retry script, `0` if not
- `BARMAN_SERVER`: name of the server
- `BARMAN_STATUS`: status of the backup
- `BARMAN_VERSION`: version of Barman

Backup delete scripts

Version **2.4** introduces pre and post backup delete scripts.

As previous scripts, backup delete scripts can be configured within global configuration options, and it is possible to override them on a per server basis:

- `pre_delete_script`: *hook script* launched *before* the deletion of a backup, only once, with no check on the exit code
- `pre_delete_retry_script`: *retry hook script* executed *before* the deletion of a backup, repeatedly until success or abort
- `post_delete_retry_script`: *retry hook script* executed *after* the deletion of a backup, repeatedly until success or abort
- `post_delete_script`: *hook script* launched *after* the deletion of a backup, only once, with no check on the exit code

The script is executed through a shell and can return any exit code. Only in case of a *retry* script, Barman checks the return code (see the upper section).

Delete scripts uses the same environmental variables of a backup script, plus:

- `BARMAN_NEXT_ID`: ID of the next backup (if present)

WAL archive scripts

Similar to backup scripts, archive scripts can be configured with global configuration options (which can be overridden on a per server basis):

- `pre_archive_script`: *hook script* executed *before* a WAL file is archived by maintenance (usually `barman cron`), only once, with no check on the exit code
- `pre_archive_retry_script`: *retry hook script* executed *before* a WAL file is archived by maintenance (usually `barman cron`), repeatedly until it is successful or aborted
- `post_archive_retry_script`: *retry hook script* executed *after* a WAL file is archived by maintenance, repeatedly until it is successful or aborted
- `post_archive_script`: *hook script* executed *after* a WAL file is archived by maintenance, only once, with no check on the exit code

The script is executed through a shell and can return any exit code. Only in case of a *retry* script, Barman checks the return code (see the upper section).

Archive scripts share with backup scripts some environmental variables:

- `BARMAN_CONFIGURATION`: configuration file used by Barman
- `BARMAN_ERROR`: error message, if any (only for the `post` phase)
- `BARMAN_PHASE`: phase of the script, either `pre` or `post`
- `BARMAN_SERVER`: name of the server

Following variables are specific to archive scripts:

- `BARMAN_SEGMENT`: name of the WAL file
- `BARMAN_FILE`: full path of the WAL file
- `BARMAN_SIZE`: size of the WAL file
- `BARMAN_TIMESTAMP`: WAL file timestamp
- `BARMAN_COMPRESSION`: type of compression used for the WAL file

WAL delete scripts

Version **2.4** introduces pre and post WAL delete scripts.

Similarly to the other hook scripts, wal delete scripts can be configured with global configuration options, and is possible to override them on a per server basis:

- `pre_wal_delete_script`: *hook script* executed *before* the deletion of a WAL file
- `pre_wal_delete_retry_script`: *retry hook script* executed *before* the deletion of a WAL file, repeatedly until it is successful or aborted
- `post_wal_delete_retry_script`: *retry hook script* executed *after* the deletion of a WAL file, repeatedly until it is successful or aborted

- `post_wal_delete_script`: *hook script* executed *after* the deletion of a WAL file

The script is executed through a shell and can return any exit code. Only in case of a *retry* script, Barman checks the return code (see the upper section).

WAL delete scripts use the same environmental variables as WAL archive scripts.

Recovery scripts

Version **2.4** introduces pre and post recovery scripts.

As previous scripts, recovery scripts can be configured within global configuration options, and is possible to override them on a per server basis:

- `pre_recovery_script`: *hook script* launched *before* the recovery of a backup, only once, with no check on the exit code
- `pre_recovery_retry_script`: *retry hook script* executed *before* the recovery of a backup, repeatedly until success or abort
- `post_recovery_retry_script`: *retry hook script* executed *after* the recovery of a backup, repeatedly until success or abort
- `post_recovery_script`: *hook script* launched *after* the recovery of a backup, only once, with no check on the exit code

The script is executed through a shell and can return any exit code. Only in case of a *retry* script, Barman checks the return code (see the upper section).

Recovery scripts uses the same environmental variables of a backup script, plus:

- `BARMAN_DESTINATION_DIRECTORY`: the directory where the new instance is recovered
- `BARMAN_TABLESPACES`: tablespace relocation map (JSON, if present)
- `BARMAN_REMOTE_COMMAND`: secure shell command used by the recovery (if present)
- `BARMAN_RECOVER_OPTIONS`: recovery additional options (JSON, if present)

Customization

Lock file directory

Barman allows you to specify a directory for lock files through the `barman_lock_directory` global option.

Lock files are used to coordinate concurrent work at global and server level (for example, cron operations, backup operations, access to the WAL archive, and so on.).

By default (for backward compatibility reasons), `barman_lock_directory` is set to `barman_home`.

TIP: Users are encouraged to use a directory in a volatile partition, such as the one dedicated to run-time variable data (e.g. `/var/run/barman`).

Binary paths

As of version 1.6.0, Barman allows users to specify one or more directories where Barman looks for executable files, using the global/server option `path_prefix`.

If a `path_prefix` is provided, it must contain a list of one or more directories separated by colon. Barman will search inside these directories first, then in those specified by the `PATH` environment variable.

By default the `path_prefix` option is empty.

Integration with cluster management systems

Barman has been designed for integration with standby servers (with streaming replication or traditional file based log shipping) and high availability tools like repmgr.

From an architectural point of view, PostgreSQL must be configured to archive WAL files directly to the Barman server. Barman, thanks to the `get-wal` framework, can also be used as a WAL hub. For this purpose, you can use the `barman-wal-restore` script, part of the `barman-cli` package, with all your standby servers.

The `replication-status` command allows you to get information about any streaming client attached to the managed server, in particular hot standby servers and WAL streamers.

Configuration Models

Configuration models define a set of overrides for configuration options. These overrides can be applied to Barman servers which are part of the same cluster as the config models. They can be useful when handling clustered environments, so you can change the configuration of a Barman server in response to failover events, for example.

As an example, let's say you have a PostgreSQL cluster with the following nodes:

- `pg-node-1`: primary
- `pg-node-2`: standby
- `pg-node-3`: standby

Assume you are backing up from the primary node, and have a configuration which includes the following options:

```
[my-barman-server]
cluster = my-cluster
conninfo = host=pg-node-1 user=barman database=postgres
```

```
streaming_conninfo = host=pg-node-1 user=streaming_barman
; other options...
```

You could, for example, have a configuration model for that cluster as follows:

```
[my-barman-server:backup-from-pg-node-2]
cluster = my-cluster
model = true
conninfo = host=pg-node-2 user=barman database=postgres
streaming_conninfo = host=pg-node-2 user=streaming_barman
```

Which could be applied upon a failover from `pg-node-1` to `pg-node-2` with the following command, so you start backing up from the new primary node:

```
barman config-switch my-barman-server my-barman-server:backup-from-pg-node-2
```

That will override the cluster configuration options with the values defined in the selected model.

> *NOTE*: not all options are configurable through models. Please refer to section 5 of the 'man' page to check settings which scope applies to models.

> *NOTE*: you might be interested in checking pg-backup-api, which can start a REST API and listen for remote requests for executing `barman` commands, including `barman config-switch`.

Parallel jobs

By default, Barman uses only one worker for file copy during both backup and recover operations. Starting from version 2.2, it is possible to customize the number of workers that will perform file copy. In this case, the files to be copied will be equally distributed among all parallel workers.

It can be configured in global and server scopes, adding these in the corresponding configuration file:

```
parallel_jobs = n
```

where `n` is the desired number of parallel workers to be used in file copy operations. The default value is 1.

In any case, users can override this value at run-time when executing `backup` or `recover` commands. For example, you can use 4 parallel workers as follows:

```
barman backup --jobs 4 server1
```

Or, alternatively:

```
barman backup --j 4 server1
```

Please note that this parallel jobs feature is only available for servers configured through `rsync`/SSH. For servers configured through streaming protocol, Barman will rely on `pg_basebackup` which is currently limited to only one worker.

Parallel jobs and sshd MaxStartups

Barman limits the rate at which parallel Rsync jobs are started in order to avoid exceeding the maximum number of concurrent unauthenticated connections allowed by the SSH server. This maximum is defined by the sshd parameter `MaxStartups` - if more than `MaxStartups` connections have been created but not yet authenticated then the SSH server may drop some or all of the connections resulting in a failed backup or recovery.

The default value of sshd `MaxStartups` on most platforms is 10. Barman therefore starts parallel jobs in batches of 10 and does not start more than one batch of jobs within a one second time period. This yields an effective rate limit of 10 jobs per second.

This limit can be changed using the following two configuration options:

- `parallel_jobs_start_batch_size`: The maximum number of parallel jobs to start in a single batch.
- `parallel_jobs_start_batch_period`: The time period in seconds over which a single batch of jobs will be started.

For example, to ensure no more than five new Rsync jobs will be created within any two second time period:

```
parallel_jobs_start_batch_size = 5
parallel_jobs_start_batch_period = 2
```

The configuration options can be overridden using the following arguments with both `barman backup` and `barman recover` commands:

- `--jobs-start-batch-size`
- `--jobs-start-batch-period`

Geographical redundancy

It is possible to set up **cascading backup architectures** with Barman, where the source of a backup server is a Barman installation rather than a PostgreSQL server.

This feature allows users to transparently keep *geographically distributed* copies of PostgreSQL backups.

In Barman jargon, a backup server that is connected to a Barman installation rather than a PostgreSQL server is defined **passive node**. A passive node is configured through the `primary_ssh_command` option, available both at global (for a full replica of a primary Barman installation) and server level (for mixed scenarios, having both *direct* and *passive* servers).

Sync information

The `barman sync-info` command is used to collect information regarding the current status of a Barman server that is useful for synchronisation purposes. The available syntax is the following:

```
barman sync-info [--primary] <server_name> [<last_wal> [<last_position>]]
```

The command returns a JSON object containing:

- A map with all the backups having status `DONE` for that server
- A list with all the archived WAL files
- The configuration for the server
- The last read position (in the *xlog database file*)
- the name of the last read WAL file

The JSON response contains all the required information for the synchronisation between the `master` and a `passive` node.

If `--primary` is specified, the command is executed on the defined primary node, rather than locally.

Configuration

Configuring a server as `passive node` is a quick operation. Simply add to the server configuration the following option:

```
primary_ssh_command = ssh barman@primary_barman
```

This option specifies the SSH connection parameters to the primary server, identifying the source of the backup data for the passive server.

If you are invoking barman with the `-c/--config` option and you want to use the same option when the passive node invokes barman on the primary node then add the following option:

```
forward_config_path = true
```

Node synchronisation

When a node is marked as `passive` it is treated in a special way by Barman:

- it is excluded from standard maintenance operations
- direct operations to PostgreSQL are forbidden, including `barman backup`

Synchronisation between a passive server and its primary is automatically managed by `barman cron` which will transparently invoke:

1. `barman sync-info --primary`, in order to collect synchronisation information
2. `barman sync-backup`, in order to create a local copy of every backup that is available on the primary node
3. `barman sync-wals`, in order to copy locally all the WAL files available on the primary node

Manual synchronisation

Although `barman cron` automatically manages passive/primary node synchronisation, it is possible to manually trigger synchronisation of a backup through:

```
barman sync-backup <server_name> <backup_id>
```

Launching `sync-backup` barman will use the primary_ssh_command to connect to the master server, then if the backup is present on the remote machine, will begin to copy all the files using rsync. Only one single synchronisation process per backup is allowed.

WAL files also can be synchronised, through:

```
barman sync-wals <server_name>
```

Cloud snapshot backups

Snapshot backups are backups which consist of one or more snapshots of cloud storage volumes.

A snapshot backup can be taken for a suitable PostgreSQL server using either of the following commands:

- `barman backup` with the required configuration operations for snapshots if a Barman server is being used to store WALs and backup metadata.
- `barman-cloud-backup` with the required command line arguments if there is no Barman server and instead a cloud object store is being used for WALs and backup metadata.

Snapshot backup details

The high level process for taking a snapshot backup is as follows:

1. Barman carries out a series of pre-flight checks to validate the snapshot options, instance and disks.
2. Barman starts a backup using the PostgreSQL backup API.
3. The cloud provider API is used to trigger a snapshot for each specified disk. Barman will wait until the snapshot has reached the required state for guaranteeing application consistency before moving on to the next disk.
4. Additional provider-specific data, such as the device name for each disk, is saved to the backup metadata.
5. The mount point and mount options for each disk are saved in the backup metadata.
6. Barman stops the backup using the PostgreSQL backup API.

The cloud provider API calls are made on the node where the backup command runs; this will be either the Barman server (when `barman backup` is used) or the PostgreSQL server (when `barman-cloud-backup` is used).

The following pre-flight checks are carried out before each backup and also when `barman check` runs against a server configured for snapshot backups:

- The compute instance specified by `snapshot_instance` and any provider-specific arguments exists.
- The disks specified by `snapshot_disks` exist.
- The disks specified by `snapshot_disks` are attached to `snapshot_instance`.
- The disks specified by `snapshot_disks` are mounted on `snapshot_instance`.

Recovering from a snapshot backup

Barman will not currently perform a fully automated recovery from snapshot backups. This is because recovery from snapshots requires the provision and management of new infrastructure which is something better handled by dedicated infrastructure-as-code solutions such as Terraform.

However, the `barman recover` command can still be used to validate the snapshot recovery instance, carry out post-recovery tasks such as checking the PostgreSQL configuration for unsafe options and set any required PITR options. It will also copy the backup_label file into place (since the backup label is not stored in any of the volume snapshots) and copy across any required WALs (unless the `--get-wal` recovery option is used, in which case it will configure the PostgreSQL `restore_command` to fetch the WALs).

If restoring a backup made with `barman-cloud-backup` then the more limited barman-cloud-restore command should be used instead of `barman recover`.

Recovery from a snapshot backup consists of the following steps:

1. Provision a new disk for each snapshot taken during the backup.
2. Provision a compute instance where each disk provisioned in step 1 is attached and mounted according to the backup metadata.
3. Use the barman recover or barman-cloud-restore command to validate and finalize the recovery.

Steps 1 and 2 are best handled by an existing infrastructure-as-code system however it is also possible to carry these steps out manually or using a custom script.

The following resources may be helpful when carrying out these steps:

- An example recovery script for GCP.
- An example runbook for Azure.

The above resources make assumptions about the backup/recovery environment and should not be considered suitable for production use without further customization.

Once the recovery instance is provisioned and disks cloned from the backup snapshots are attached and mounted, run `barman recover` with the following additional arguments:

- `--remote-ssh-command`: The ssh command required to log in to the recovery instance.
- `--snapshot-recovery-instance`: The name of the recovery instance as required by the cloud provider.
- Any additional arguments specific to the snapshot provider.

For example:

```
barman recover SERVER_NAME BACKUP_ID REMOTE_RECOVERY_DIRECTORY \
    --remote-ssh-command 'ssh USER@HOST' \
    --snapshot-recovery-instance INSTANCE_NAME
```

Barman will automatically detect that the backup is a snapshot backup and check that the attached disks were cloned from the snapshots for that backup. Barman will then prepare PostgreSQL for recovery by copying the backup label and WALs into place and setting any required recovery options in the PostgreSQL configuration.

The following additional `barman recover` arguments are available with the `gcp` provider:

- `--gcp-zone`: The name of the availability zone in which the recovery instance is located. If not provided then Barman will use the value of `gcp_zone` set in the server config.

The following additional `barman recover` arguments are available with the `azure` provider:

- `--azure-resource-group`: The resource group to which the recovery instance belongs. If not provided then Barman will use the value of `azure_resource_group` set in the server config.

The following additional `barman recover` arguments are available with the `aws` provider:

- `--aws-region`: The AWS region in which the recovery instance is located. If not provided then Barman will use the value of `aws_region` set in the server config.

Note the following `barman recover` arguments / config variables are unavailable when recovering snapshot backups:

Command argument	Config variable .
`--bwlimit`	`bandwidth_limit`
`--jobs`	`parallel_jobs`
`--recovery-staging-path`	`recovery_staging_path`
`--tablespace`	N/A

Backup metadata for snapshot backups

Whether the recovery disks and instance are provisioned via infrastructure-as-code, ad-hoc automation or manually, it will be necessary to query Barman to find the snapshots required for a given backup. This can be achieved using barman show-backup which will provide details for each snapshot in the backup. For example:

```
$ barman show-backup primary 20230123T131430
Backup 20230123T131430:
  Server Name            : primary
  System Id              : 7190784995399903779
  Status                 : DONE
  PostgreSQL Version     : 140006
  PGDATA directory       : /opt/postgres/data

  Server information:
    Checksums            : on

  Snapshot information:
    provider             : gcp
    project              : project_id

    device_name          : pgdata
    snapshot_name        : barman-av-ubuntu20-primary-pgdata-20230123t131430
    snapshot_project     : project_id
    Mount point          : /opt/postgres
    Mount options        : rw,noatime
    device_name          : tbs1
    snapshot_name        : barman-av-ubuntu20-primary-tbs1-20230123t131430
    snapshot_project     : project_id
    Mount point          : /opt/postgres/tablespaces/tbs1
    Mount options        : rw,noatime
```

The the `--format=json` option can be used when integrating with external tooling, e.g.:

```
$ barman --format=json show-backup primary 20230123T131430
...
"snapshots_info": {
  "provider": "gcp",
  "provider_info": {
    "project": "project_id"
  },
  "snapshots": [
    {
      "mount": {
        "mount_options": "rw,noatime",
        "mount_point": "/opt/postgres"
      },
      "provider": {
        "device_name": "pgdata",
        "snapshot_name": "barman-av-ubuntu20-primary-pgdata-20230123t131430",
        "snapshot_project": "project_id"
      }
    },
    {
      "mount": {
        "mount_options": "rw,noatime",
        "mount_point": "/opt/postgres/tablespaces/tbs1"
      },
      "provider": {
        "device_name": "tbs1",
        "snapshot_name": "barman-av-ubuntu20-primary-tbs1-20230123t131430",
        "snapshot_project": "project_id",
      }
    }
  ]
}
...
```

For backups taken with `barman-cloud-backup` there is an analogous barman-cloud-backup-show command which can be used along with `barman-cloud-backup-list` to query the backup metadata in the cloud object store.

The metadata available in `snapshots_info/provider_info` and `snapshots_info/snapshots/*/provider` varies by cloud provider as explained in the following sections.

GCP provider-specific metadata

The following fields are available in `snapshots_info/provider_info`:

- `project`: The GCP project ID of the project which owns the resources involved in backup and recovery.

The following fields are available in `snapshots_info/snapshots/*/provider`:

- `device_name`: The short device name with which the source disk for the snapshot was attached to the backup VM at the time of the backup.
- `snapshot_name`: The name of the snapshot.
- `snapshot_project`: The GCP project ID which owns the snapshot.

Azure provider-specific metadata

The following fields are available in `snapshots_info/provider_info`:

- `subscription_id`: The Azure subscription ID which owns the resources involved in backup and recovery.
- `resource_group`: The Azure resource group to which the resources involved in the backup belong.

The following fields are available in `snapshots_info/snapshots/*/provider`:

- `location`: The Azure location of the disk from which the snapshot was taken.
- `lun`: The LUN identifying the disk from which the snapshot was taken at the time of the backup.
- `snapshot_name`: The name of the snapshot.

AWS provider-specific metadata

The following fields are available in `snapshots_info/provider_info`:

- `account_id`: The ID of the AWS account which owns the resources used to make the backup.
- `region`: The AWS region in which the resources involved in backup are located.

The following fields are available in `snapshots_info/snapshots/*/provider`:

- `device_name`: The device to which the source disk was mapped on the backup VM at the time of the backup.
- `snapshot_id`: The ID of the snapshot as assigned by AWS.
- `snapshot_name`: The name of the snapshot.

Barman client utilities (`barman-cli`)

Formerly a separate open-source project, `barman-cli` has been merged into Barman's core since version 2.8, and is distributed as an RPM/Debian package. `barman-cli` contains a set of recommended client utilities to be installed alongside the PostgreSQL server:

- `barman-wal-archive`: archiving script to be used as `archive_command` as described in the "WAL archiving via `barman-wal-archive`" section;
- `barman-wal-restore`: WAL restore script to be used as part of the `restore_command` recovery option on standby and recovery servers, as described in the "`get-wal`" section above;

For more detailed information, please refer to the specific man pages or the `--help` option.

Installation

Barman client utilities are normally installed where PostgreSQL is installed. Our recommendation is to install the `barman-cli` package on every PostgreSQL server, being that primary or standby.

Please refer to the main "Installation" section to install the repositories.

To install the package on RedHat/CentOS system, as `root` type:

```
yum install barman-cli
```

On Debian/Ubuntu, as `root` user type:

```
apt-get install barman-cli
```

Barman client utilities for the Cloud (`barman-cli-cloud`)

Barman client utilities have been extended to support object storage integration and enhance disaster recovery capabilities of your PostgreSQL databases by relaying WAL files and backups to a supported cloud provider.

Supported cloud providers are:

- AWS S3 (or any S3 compatible object store)
- Azure Blob Storage
- Google Cloud Storage (Rest API)

These utilities are distributed in the `barman-cli-cloud` RPM/Debian package, and can be installed alongside the PostgreSQL server:

- `barman-cloud-wal-archive`: archiving script to be used as `archive_command` to directly ship WAL files to cloud storage, bypassing the Barman server; alternatively, as a hook script for WAL archiving (`pre_archive_retry_script`);
- `barman-cloud-wal-restore`: script to be used as `restore_command` to fetch WAL files from cloud storage, bypassing the Barman server, and store them directly in the PostgreSQL standby;
- `barman-cloud-backup`: backup script to be used to take a local backup directly on the PostgreSQL server and to ship it to a supported cloud provider, bypassing the Barman server; alternatively, as a hook script for copying barman backups to the cloud (`post_backup_retry_script`)
- `barman-cloud-backup-delete`: script to be used to delete one or more backups taken with `barman-cloud-backup` from cloud storage and remove associated WALs;
- `barman-cloud-backup-keep`: script to be used to flag backups in cloud storage as archival backups - such backups will be kept forever regardless of any retention policies applied;
- `barman-cloud-backup-list`: script to be used to list the content of Barman backups taken with `barman-cloud-backup` from cloud storage;
- `barman-cloud-backup-show`: script to be used to display the metadata for a Barman backup taken with `barman-cloud-backup`;
- `barman-cloud-restore`: script to be used to restore a backup directly taken with `barman-cloud-backup` from cloud storage;

These commands require the appropriate library for the cloud provider you wish to use:

- AWS S3: boto3
- Azure Blob Storage: azure-storage-blob and (optionally) azure-identity
- Google Cloud Storage: google-cloud-storage

For information on how to setup credentials for the aws-s3 cloud provider please refer to the "Credentials" section in Boto 3 documentation.

For credentials for the azure-blob-storage cloud provider see the "Environment variables for authorization parameters" section in the Azure documentation. The following environment variables are supported: `AZURE_STORAGE_CONNECTION_STRING`, `AZURE_STORAGE_KEY` and `AZURE_STORAGE_SAS_TOKEN`. You can also use the `--credential` option to specify either `azure-cli` or `managed-identity` credentials in order to authenticate via Azure Active Directory.

Installation

Barman client utilities for the Cloud need to be installed on those PostgreSQL servers that you want to directly backup to a cloud provider, bypassing Barman.

In case you want to use `barman-cloud-backup` and/or `barman-cloud-wal-archive` as hook scripts, you can install the `barman-cli-cloud` package on the Barman server also.

Please refer to the main "Installation" section to install the repositories.

To install the package on RedHat/CentOS system, as `root` type:

```
yum install barman-cli-cloud
```

On Debian/Ubuntu, as `root` user type:

```
apt-get install barman-cli-cloud
```

barman-cloud hook scripts

Install the `barman-cli-cloud` package on the Barman server as described above.

It is possible to use `barman-cloud-backup` as a post backup script for the following Barman backup flavours:

- Backups taken with `backup_method = rsync`.
- Backups taken with `backup_method = postgres` where `backup_compression` is not used.

To do so, add the following to a server configuration in Barman:

```
post_backup_retry_script = 'barman-cloud-backup [*OPTIONS*] *DESTINATION_URL* ${BARMAN_SERVER}
```

> **WARNING:** When running as a hook script barman-cloud-backup requires that the status of the backup is DONE and it will fail if the backup has any other status. For this reason it is recommended backups are run with the `-w / --wait` option so that the hook script is not executed while a backup has status `WAITING_FOR_WALS`.

Configure `barman-cloud-wal-archive` as a pre WAL archive script by adding the following to the Barman configuration for a PostgreSQL server:

```
pre_archive_retry_script = 'barman-cloud-wal-archive [*OPTIONS*] *DESTINATION_URL* ${BARMAN_SE
```

Selecting a cloud provider

Use the `--cloud-provider` option to choose the cloud provider for your backups and WALs. This can be set to one of the following:

- `aws-s3` [DEFAULT]: AWS S3 or S3-compatible object store.
- `azure-blob-storage`: Azure Blob Storage service.
- `google-cloud-storage`: Google Cloud Storage service.

Specificity by provider

Google Cloud Storage

set up

It will need google_storage_client dependency:

```
pip3 install google-cloud-storage
```

To set credentials:

- Create a service account And create a service account key.
- Set bucket access rights:

  We suggest to give Storage Admin Role to the service account on the bucket.
- When using barman_cloud, If the bucket does not exist, it will be created. Default options will be used to create the bucket. If you need the bucket to have specific options (region, storage class, labels), it is advised to create and set the bucket to match all you needs.
- Set env variable `GOOGLE_APPLICATION_CREDENTIALS` to the service account key file path.

  If running barman cloud from postgres (archive_command or restore_command), do not forget to set `GOOGLE_APPLICATION_CREDENTIALS` in postgres environment file.

Usage

Some details are specific to all barman cloud commands: * Select Google Cloud Storage`--cloud-provider=google-cloud-storage` * `SOURCE_URL` support both gs and https format. ex: `gs://BUCKET_NAME/path` or `https://console.cloud.google.com/storage/browser/BUCKET_NAME/path`

barman-cloud and snapshot backups

The barman-cloud client utilities can also be used to create and manage backups using cloud snapshots as an alternative to uploading to a cloud object store.

When using barman-cloud in this manner the backup data is stored by the cloud provider as volume snapshots and the WALs and backup metadata, including the backup_label, are stored in cloud object storage.

The prerequisites are the same as for snapshot backups using Barman with the added requirement that the credentials used by barman-cloud must be able to perform read/write/update operations against an object store.

barman-cloud-backup for snapshots

To take a snapshot backup with barman-cloud, use `barman-cloud-backup` with the following additional arguments:

- `--snapshot-disk` (can be used multiple times for multiple disks)
- `--snapshot-instance`

If the `--cloud-provider` is `google-cloud-storage` then the following arguments are also required:

- `--gcp-project`
- `--gcp-zone`

If the `--cloud-provider` is `azure-blob-storage` then the following arguments are also required:

- `--azure-subscription-id`
- `--azure-resource-group`

If the `--cloud-provider` is `aws-s3` then the following optional arguments can be used:

- `--aws-profile`
- `--aws-region`

The following options cannot be used with `barman-cloud-backup` when cloud snapshots are requested:

- `--bzip2`, `--gzip` or `--snappy`
- `--jobs`

Once a backup has been taken it can be managed using the standard barman-cloud commands such as `barman-cloud-backup-delete` and `barman-cloud-backup-keep`.

barman-cloud-restore for snapshots

The process for recovering from a snapshot backup with barman-cloud is very similar to the process for barman backups except that `barman-cloud-restore` should be run instead of `barman recover` once a recovery instance has been provisioned. This carries out the same pre-recovery checks as `barman recover` and copies the backup label into place on the recovery instance.

The snapshot metadata required to provision the recovery instance can be queried using `barman-cloud-backup-show`.

Note that, just like when using `barman-cloud-restore` with an object stored backup, the command will not prepare PostgreSQL for the recovery. Any PITR options, custom `restore_command` values or WAL files required before PostgreSQL starts must be handled manually or by external tooling.

The following additional argument must be used with `barman-cloud-restore` when restoring a backup made with cloud snapshots:

- `--snapshot-recovery-instance`

The following additional arguments are required with the `gcp` provider:

- `--gcp-zone`

The following additional arguments are required with the `azure` provider:

- `--azure-resource-group`

The following additional argument is available with the `aws-s3` provider:

- `--aws-region`

The `--tablespace` option cannot be used with `barman-cloud-restore` when restoring a cloud snapshot backup:

Troubleshooting

Diagnose a Barman installation

You can gather important information about the status of all the configured servers using:

```
barman diagnose
```

The `diagnose` command output is a full snapshot of the barman server, providing useful information, such as global configuration, SSH version, Python version, `rsync` version, PostgreSQL clients version, as well as current configuration and status of all servers.

The `diagnose` command is extremely useful for troubleshooting problems, as it gives a global view on the status of your Barman installation.

Requesting help

Although Barman is extensively documented, there are a lot of scenarios that are not covered.

For any questions about Barman and disaster recovery scenarios using Barman, you can reach the dev team using the community mailing list:

https://groups.google.com/group/pgbarman

or the IRC channel on freenode: irc://irc.freenode.net/barman

In the event you discover a bug, you can open a ticket using GitHub: https://github.com/EnterpriseDB/barman/issues

EnterpriseDB provides professional support for Barman, including 24/7 service.

Submitting a bug

Barman has been extensively tested and is currently being used in several production environments. However, as any software, Barman is not bug free.

If you discover a bug, please follow this procedure:

- execute the `barman diagnose` command
- file a bug through the GitHub issue tracker, by attaching the output obtained by the diagnostics command above (`barman diagnose`)

  WARNING: Be careful when submitting the output of the diagnose command as it might disclose information that are potentially dangerous from a security point of view.

The Barman project

Support and sponsor opportunities

Barman is free software, written and maintained by EnterpriseDB. If you require support on using Barman, or if you need new features, please get in touch with EnterpriseDB. You can sponsor the development of new features of Barman and PostgreSQL which will be made publicly available as open source.

For further information, please visit:

- Barman website
- Support section
- EnterpriseDB website
- Barman FAQs
- 2ndQuadrant blog: Barman

Contributing to Barman

EnterpriseDB has a team of software engineers, architects, database administrators, system administrators, QA engineers, developers and managers that dedicate their time and expertise to improve Barman's code. We adopt lean and agile methodologies for software development, and we believe in the *devops* culture that allowed us to implement rigorous testing procedures through cross-functional collaboration. Every Barman commit is the contribution of multiple individuals, at different stages of the production pipeline.

Even though this is our preferred way of developing Barman, we gladly accept patches from external developers, as long as:

- user documentation (tutorial and man pages) is provided.
- source code is properly documented and contains relevant comments.
- code supplied is covered by unit tests.
- no unrelated feature is compromised or broken.
- source code is rebased on the current master branch.
- commits and pull requests are limited to a single feature (multi-feature patches are hard to test and review).
- changes to the user interface are discussed beforehand with EnterpriseDB.

We also require that any contributions provide a copyright assignment and a disclaimer of any work-for-hire ownership claims from the employer of the developer.

You can use GitHub's pull requests system for this purpose.

Authors

In alphabetical order:

- Andre Marchesini
- Barbara Leidens
- Giulio Calacoci
- Gustavo Oliveira
- Israel Barth
- Martín Marqués

Past contributors (in alphabetical order):

- Abhijit Menon-Sen (architect)
- Anna Bellandi (QA/testing)
- Britt Cole (documentation reviewer)
- Carlo Ascani (developer)
- Didier Michel (developer)
- Francesco Canovai (QA/testing)
- Gabriele Bartolini (architect)
- Gianni Ciolli (QA/testing)
- Giulio Calacoci (developer)
- Giuseppe Broccolo (developer)
- Jane Threefoot (developer)
- Jonathan Battiato (QA/testing)
- Leonardo Cecchi (developer)
- Marco Nenciarini (project leader)
- Michael Wallace (developer)
- Niccolò Fei (QA/testing)
- Rubens Souza (QA/testing)
- Stefano Bianucci (developer)

Links

- check-barman: a Nagios plugin for Barman, written by Holger Hamann (MIT license)
- puppet-barman: Barman module for Puppet (GPL)
- Tutorial on "How To Back Up, Restore, and Migrate PostgreSQL Databases with Barman on CentOS 7", by Sadequl Hussain (available on DigitalOcean Community)
- BarmanAPI: RESTFul API for Barman, written by Mehmet Emin Karakaş (GPL)

License and Contributions

Barman is the property of EnterpriseDB UK Limited and its code is distributed under GNU General Public License 3.

Barman has been partially funded through 4CaaSt, a research project funded by the European Commission's Seventh Framework programme.

Contributions to Barman are welcome, and will be listed in the `AUTHORS` file. EnterpriseDB UK Limited requires that any contributions provide a copyright assignment and a disclaimer of any work-for-hire ownership claims from the employer of the developer. This lets us make sure that all of the Barman distribution remains free code. Please contact barman@enterprisedb.com for a copy of the relevant Copyright Assignment Form.