TECHNICAL WHITE PAPER

Protecting PostgreSQL with FlashBlade®

Rapid protection using tools for PostgreSQL.

Contents

Introduction

PostgreSQL, also known as Postgres, is a free and open-source relational database system. The database is designed to handle different workloads from small isolated databases deployed to data warehouses or web services scaled across multiple systems. Deployment is highly versatile allowing for PostgreSQL to be used in containers, directly on a number of operating systems or a software as a service offering.

According to DB-Engines, as of March 2023 PostgreSQL is the fourth most popular database in the world. As many PostgreSQL databases are used by organizations to store critical business information, it is imperative that they implement a strategy to recover business operations in the event of data loss or system failure. PostgreSQL comes with a number of built-in data protection and recovery tools and is also supported by a vibrant third-party tool developer community for more advanced tooling. Even with advanced tools, sometimes core issues such as recovery time objectives (RTO) and recovery point objectives (RPO) are still difficult to achieve. With Pure Storage® FlashBlade® these core issues can be mitigated or eliminated entirely depending on the protection method used.

This technical white paper is intended to be used as a how-to and best practice guide when protecting PostgreSQL with various tools and FlashBlade as the target storage appliance. The target audience for this document includes, but is not limited to, systems architects, database administrators, storage administrators and IT professionals.

How to Use This Guide

This guide will focus on protecting PostgreSQL using tools such as pg_dump, pg_basebackup ,pgbackrest and BARMAN. Tools that are shipped with the standard PostgreSQL installation as well as some open-source tooling will be covered. How to implement and attain the best outcome for each tool will be provided in the form of outcome guidance.

Pure Storage FlashBlade//S™

FlashBlade//S™ is the next-generation FlashBlade, designed for exabyte scale to support the needs of modern data and applications for the next decade and beyond. It builds on the simplicity, reliability and scalability of the first-generation FlashBlade.

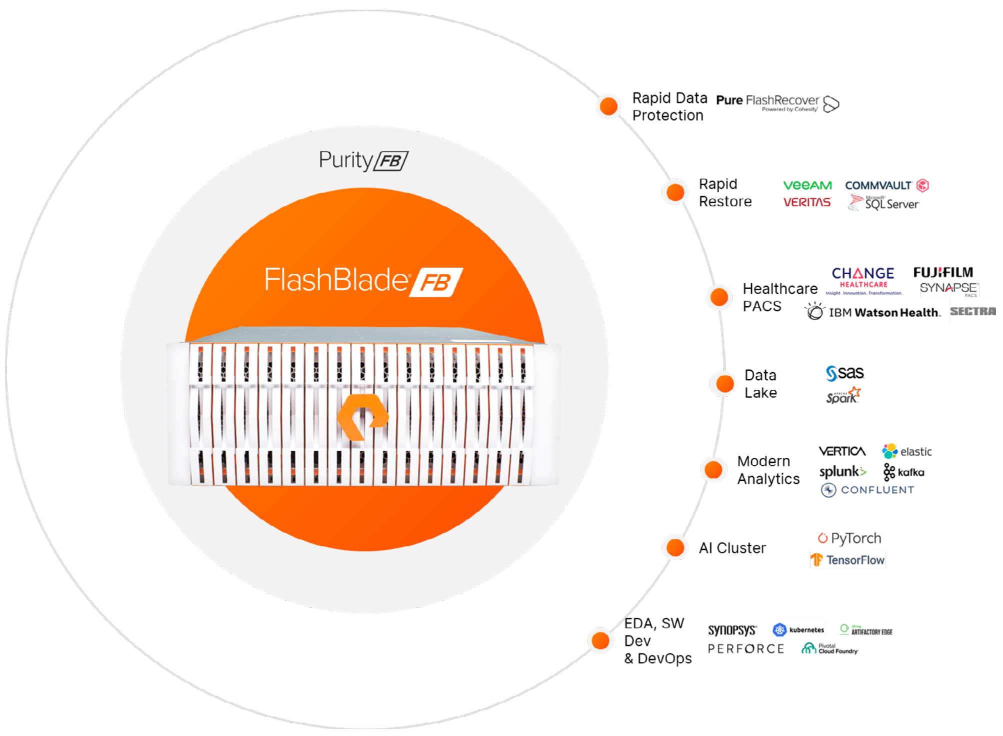


FIGURE 1 Consolidate data silos with FlashBlade//S

FlashBlade//S is the result of our co-innovation of hardware and software. Most storage platforms exist on one of two extremes: either disk or hybrid solutions for capacity-optimized workloads, or all-flash solutions that use TLC or QLC with massive caching to achieve performance. Because of its design, FlashBlade//S can target a wide variety of workload profiles across both spectrums.

Built using DirectFlash® Modules and all-QLC architecture, FlashBlade//S is the ideal foundation for modern workloads. It's a unified fast file and object (UFFO) storage platform that provides rich data services with higher density and capacity than ever before. FlashBlade//S is designed to easily support the most demanding unstructured data workloads, without compromising on system performance or efficiency.

Architectures that use off-the-shelf solid-state drives (SSDs) have an internal controller to manage the flash media on each specific drive without any knowledge of what's happening at the systems level. In contrast, FlashBlade//S uses Pure Storage's innovative DirectFlash Modules, enabling the storage operating system to manage that media on a global level. The DirectFlash Modules include a small amount of NVRAM that scales as the platform grows.

Purity//FB, the operating system for FlashBlade//S, manages all system resources including blades and DirectFlashModules at a global level. Global media management enables FlashBlade//S Direct Flash Modules to unlock as much as 20% more capacity from NAND when compared to competitors using off-the-shelf SSDs. This delivers more consistent performance, better reliability, and higher media endurance without requiring massive storage class memory (SCM) cache.

FlashBlade//S enables enterprise-level data management at scale. Using a distributed metadata architecture, it offers multi-dimensional performance on a consolidated platform with NFS, SMB, and S3 protocol access. The cloud-based Pure1® data management platform provides a single view to monitor, analyze, and optimize your storage from anywhere.

FlashBlade//S architecture scales compute and storage independently. With its unique modular architecture that allows you to easily increase capacity or performance, FlashBlade//S is a customizable platform that gives you the ability to tailor your configuration for specific workload requirements. It provides the flexibility to easily adapt to your data growth projections and evolving storage needs.

Solution Architecture

Hardware Environment

During validation of this solution multiple PostgreSQL clusters are deployed on a Pure Storage FlashArray//X90 using NVMeoF-RoCEv2 connectivity. The hosts for each PostgreSQL cluster are configured with separate network interfaces for the storage network and application/backup network. The storage and application/backup network are completely isolated and no traffic can be routed between them. The environment is made up of 8 (eight) servers each with a total of 64 cores and 512GB memory.

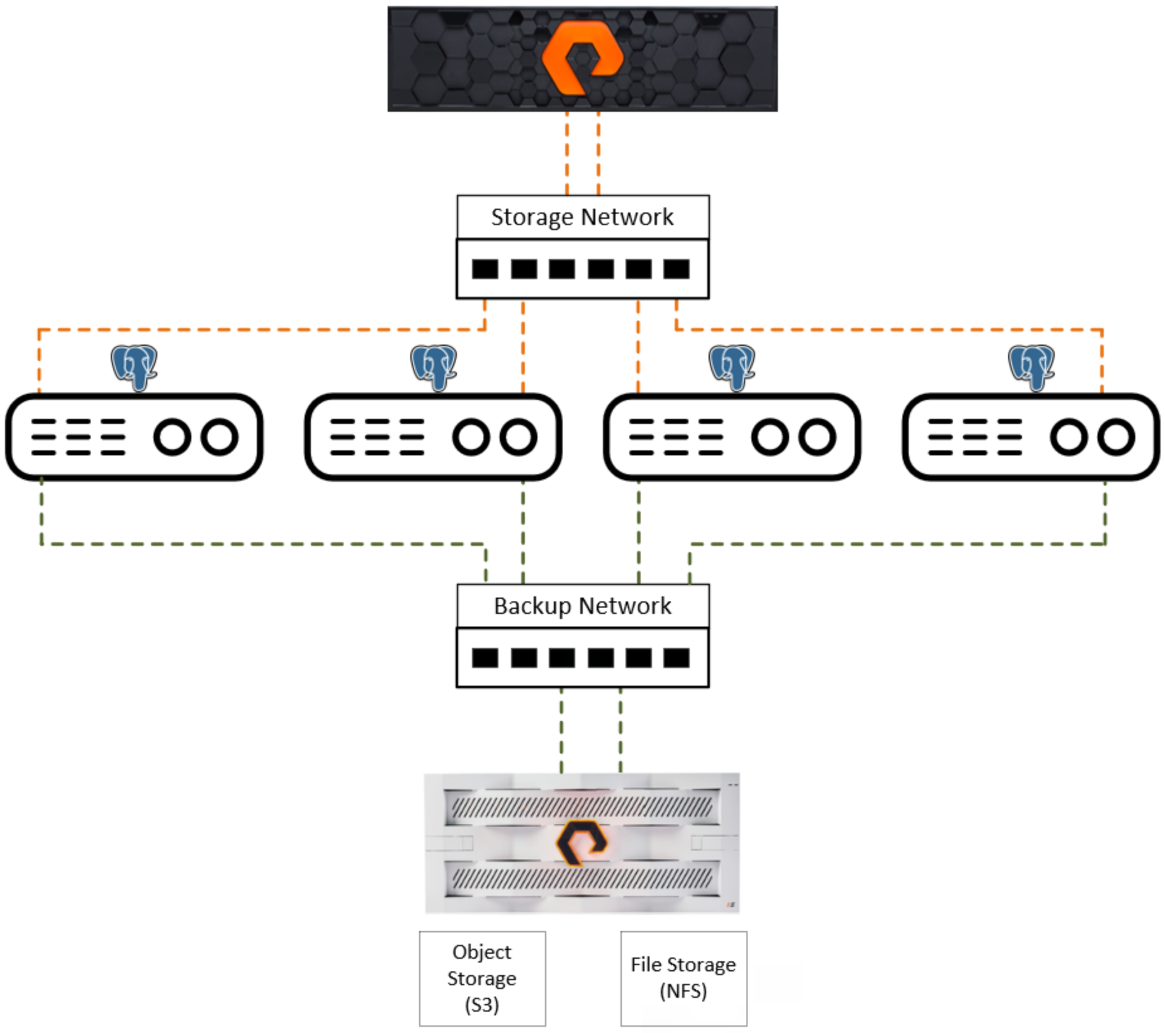


FIGURE 2 Hardware configuration

Software Environment

PostgreSQL uses terminology that can differ in some cases from the standard SQL terminology:

- A PostgreSQL cluster is the database server or instance containing all of the PostgreSQL data management capabilities.
- A catalog is a database created within the cluster to contain database schemas which in turn house database objects.
- Write-Ahead-Logging (WAL) is a concept that will be covered at length in this document. It is a standard method for ensuring data integrity by only allowing for changes to be written to data files after they have been written to a transaction log. This ensures that data can be recovered from the WAL in the event of a system crash. The use of WAL allows for on-line backup and point-in-time recovery when combined with WAL Archiving.

Operating System

Each PostgreSQL cluster is deployed on Red Hat Enterprise Linux 8.

Database Storage Layout

Each system is connected to a single volume from FlashArray, for primary storage capabilities, which is in turn formatted with the XFS filesystem and mounted to the default Linux PostgreSQL location (/var/lib/pgsql). For some scenarios additional volumes are connected and new tablespaces are created on them. The additional tablespaces are used to showcase how backup and recovery operations are done when databases are created using tablespaces not in the default location.

Backup and Recovery Storage

FlashBlade provides both file and object storage capabilities. In the instance where file storage (In this environment NFS is used) is required for backup and recovery there are a number of configuration possibilities

- A separate NFS share for each PostgreSQL cluster backup can be connected to and mounted on the relevant system. This scenario can be useful where data service functionality on FlashBlade needs to be done on a per cluster granular level.
- A single NFS share where all PostgreSQL clusters are isolated to their own folders within it. This scenario is useful where the clusters may need to access backup data from a different cluster and less granular data services may be required.

When using Object Storage, it is recommended to use a single bucket for each cluster.

The FlashBlade user guide (login required) provides guidance on the configuration of File (Chapter 5, Filesystems) and Object Storage (Chapter 5, Object Store).

PostgreSQL Cluster Configuration

This solution has been validated using PostgreSQL 14.6.

For the purposes of this solution the following configuration options were changed in the postgres.conf configuration file:

```
#------------------------------------------------------------------------
# CONNECTIONS AND AUTHENTICATION
#------------------------------------------------------------------------
# - Connection Settings -
listen_addresses = '*'
max_connections = 20000
#------------------------------------------------------------------------
# RESOURCE USAGE (except WAL)
#------------------------------------------------------------------------
# - Memory -
shared_buffers = 64GB
# - Asynchronous Behavior -
effective_io_concurrency = 512
maintenance_io_concurrency = 512
max_worker_processes = 64
max_parallel_maintenance_workers = 16
max_parallel_workers_per_gather = 16
parallel_leader_participation = on
max_parallel_workers = 32
```

For more information on using PostgreSQL with FlashArray see the PostgreSQL on FlashArray Implementation and Best Practice Guide.

PostgreSQL Backup and Recovery Tools

There are a number of tools that can be used to protect PostgreSQL cluster data. Some of these tools such as SQL dump (pg_dump and pg_dumpall) will create a logical backup and others such as pg_basebackup and pgBackRest create physical backups. Continuous archiving is a functional concept that utilizes other tools to accomplish its goals but is discussed in depth due to its importance for point-in-time recovery capabilities.

SQL Dump

An SQL dump can be taken using the pg_dump or pg_dumpall utility provided with each PostgreSQL installation. This is a logical backup technique where the contents of the database are read while the database is online and then exported to one or more script files.

The pg_dump utility will only dump a single database while pg_dumpall will dump all databases in the cluster. These tools create consistent backups even if the database is being used concurrently and will not block new input or changes.

Both utilities are executed in a command line interface, using arguments to specify optimization and output. All SQL dumps should be sent to an NFS or SMB share on FlashBlade.

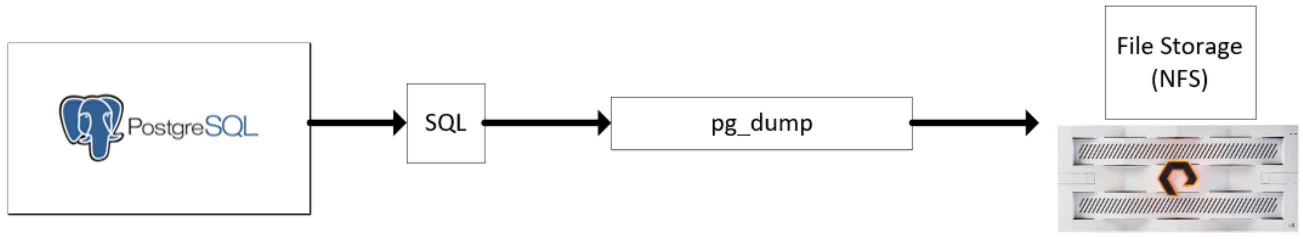


FIGURE 3 SQL Dump data path

pg_dump

This utility expects commands in this form:

```
pg_dump [connection-option...] [option...] [dbname]
```

Backups can be performed from either a remote or local system. Dumps can be output to SQL scripts, plain text files containing SQL commands to reconstruct the database, or one of the archive files formats. SQL scripts are useful when the database needs to be reconstructed on a different system or different architecture. The archive file formats allow for greater flexibility with the following options:

- **Plain:** This is the default format which outputs plain-text SQL script.
- **Custom:** This will output a custom-format archive. During restore this allows for manual selection and recording of archived items. Can be combined with a directory to create greater flexibility. This format is compressed by default.
- **Directory:** This creates a directory with one file for each table and BLOB being dumped and a table of contents file describing the dumped objects in a machine-readable format.
- **Tar:** This outputs a tar-format archive. This is compatible with the directory format but does not support compression.

This utility allows for the dump to be run in parallel using the --jobs argument. Using multiple jobs can reduce the time needed to perform the dump but increases the load on the PostgreSQL cluster. The max_connections configuration option needs to be set high enough to handle the increased connections as pg_dump will open the number of jobs + 1 connections to the database.
To ensure a consistent backup the database needs to support synchronized snapshots which were introduced in PostgreSQL 9.2 for primary servers and PostgreSQL 10 for standbys.

Creating a database dump of the database1 database to a plain text file the pg_dump output can be redirected to a file with this command:

```
pg_dump database1 > database1.sql
```

To restore a database backup taken by pg_dump from a plain text format file the database is reloaded into a freshly created database (in this case named database2 specified using the -d argument) with the psql utility:

```
psql -d database2 -f db.sql
```

To create a pg_dump backup of a database called database1 with 64 jobs at a filesystem folder location of /postgres/backup/pg_dump location using the directory format the following command would be used:

```
pg_dump -f /postgres/backup/pg_dump -j 64 -F directory -d database1
```

To restore a database from an archive file format the pg_restore utility needs to be used. The utility issues the commands required to reconstruct the database to the state in which it was at the time the dump was created. The pg_restore utility can operate in one of two modes:

- If a database name is specified then the database is connected to and the archive contents are restored directly into it.
- If a database name is not specified then a script containing the SQL commands necessary to rebuild the database are created and written to a file or the standard output.

Operations with pg_restore can restore archives to a local or remote PostgreSQL cluster. As with pg_dump , pg_restore allows the number of simulation operations for loading data, creating indexes or creating constraints to be increased using the --jobs argument. Using the jobs argument can dramatically reduce the time needed for a restore for a server running on a multiprocessor system. Only backups taken with pg_dump in the custom or directory format can be used with the --jobs option.

To restore a backup created from pg_dump using the directory format to the local system with 64 jobs from the filesystem location /postgres/backup/pg_dump the following command can be used:

```
pg_restore -h localhost -p 5432 -U postgres -d database1 -j 64 /postgres/backup/pg_dump
```

Pg_dumpall

This utility accepts commands in this form:

```
pg_dumpall [connection-option...] [option...]
```

Pg_dumpall is a wrapper around the execution of pg_dump where this utility will call pg_dump for each database in the cluster and then place the output for all of them into a single plain-text SQL script. The utility allows for a number of options when exporting data these are:

- Dump everything, data definitions, object content and tablespaces.
- Dump only data, no schema or data definitions (--data-only).
- Dump only the tablespaces (--tablespaces-only).

To take a backup using pg_dumpall the following command can be used:

```
pg_dumpall > db.out
```

To restore a backup created by pg_dumpall the database can be reloaded using the psql utility with this command:

```
psql -f db.out postgres
```

File System Level Backup

There are a number of options that can be used to create a filesystem level (physical) backup. Some of the simpler approaches would be to use file copy commands (COPY, cp, rsync, etc) of the PostgreSQL data directory to another location while the database is offline. All file system level backups should be sent to an SMB or NFS share on FlashBlade. This section will only deal with the creation of a filesystem level backup using pg_basebackup.

pg_basebackup

This utility is used to take a base backup of a running PostgreSQL cluster. This backup is taken without affecting client connections to the database and can be used as a starting point for log-shipping or streaming replication and point-in-time recovery options.

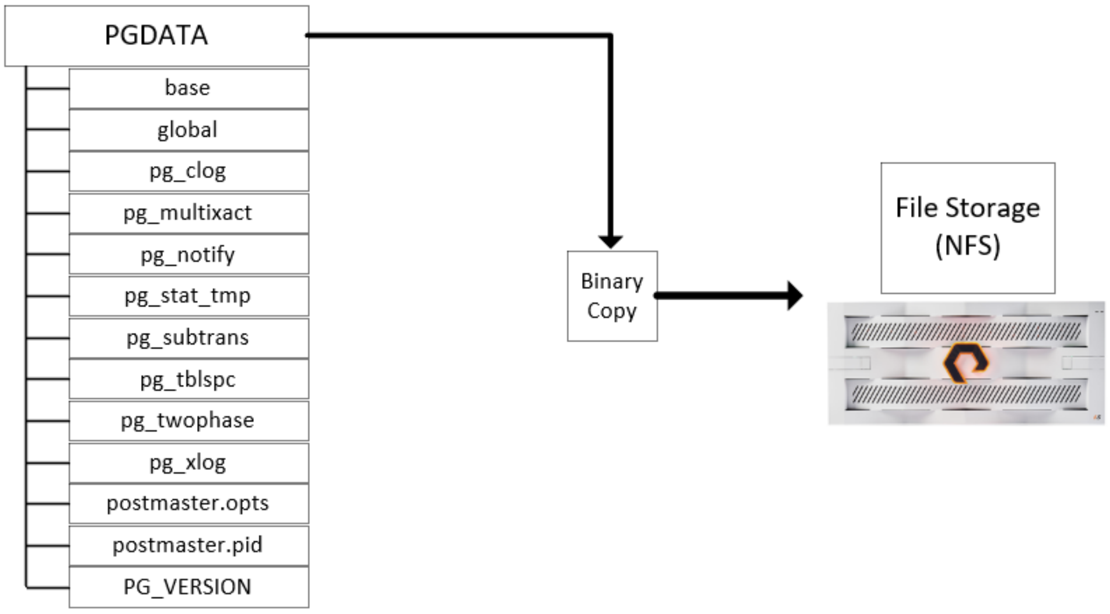


FIGURE 4 pg_basebackup data path

The utility places the database cluster into backup mode, makes an exact copy of the database files at the specified location and then takes the cluster out of backup mode. Backups are always taken of the entire database cluster. Backups can be created using either a plain or tar format. A plain format will output as plain files with the same layout as the source server's data directory and tablespaces. The tar format will output as tar files in the target directory. Where multiple tablespaces are used there is a requirement to map the tablespaces to a target location folder using the --tablespace-mapping argument for each in the cluster.

To create a backup using pg_basebackup with a tar format, WAL streaming mode (pg_basebackup will capture and store WAL log segments created during the basebackup) and progress reporting to the filesystem location /postgres/backup/pg_basebackup/ the following command can be used:

```
pg_basebackup -D /postgres/backup/pg_basebackup/ -Ft -Xs -P
```

To restore from a pg_basebackup all that is required is that the files are first unarchived, if using the tar format, and copied from the backup location to the intended PostgreSQL data path location.

Continuous Archiving and Point-in-time Recovery

PostgreSQL maintains a transaction log in the form of a write-ahead-log (WAL) that records all changes made to the database. At regular intervals a checkpoint of the database is performed where the changes in the WAL are then written to the database data files. If the cluster crashes before the checkpoint, then data integrity is assured by replaying (or repeating) the transactions that occurred between the last checkpoint and the crash to ensure all databases in the cluster remain consistent. Continuous archiving can be sent to either File (NFS or SMB) or Object Storage(S3) on FlashBlade.

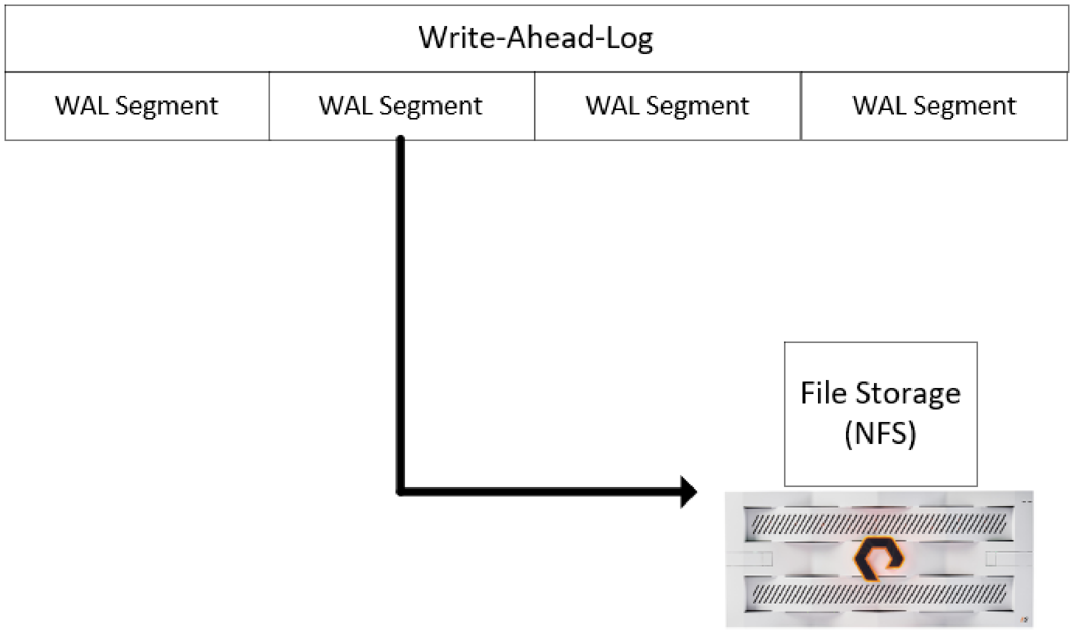


FIGURE 5 Continuous archiving data path

Continuous archiving is where once the WAL segment files (the WAL is an indefinitely long sequence of records, segment files are how the WAL divides this long sequence into manageable sections) are full they are copied to a third location prior to being recycled for later use. For WAL archiving to be enabled the wal_level configuration parameter must be replica or higher, archive_mode must be on and the segment files can be copied to an alternative location using the archive_command configuration parameter.

```
wal_level = replica
archive_mode = on
```

When setting the archive_command configuration parameter the variables %p will be replaced by PostgreSQL with the path name of the file to archive and %f is replaced by the file name. Examples of archiving to different locations for archive_command are shown below:

```
# Copy to NFS share for Linux/Unix OS
archive_command = 'test ! -f /mnt/server/archivedir/%f && cp %p /mnt/server/archivedir/%f'
# Copy to SMB share for Microsoft Windows
archive_command = 'copy "%p" "C:\\server\\archivedir\\%f"'
# Copy to S3 bucket using s3cmd
archive_command = 's3cmd put %p s3://BUCKET/path/%f'
```

Point-in-time Recovery when using continuous archiving requires that two requirements are met:

- A base physical backup is taken using pg_basebackup, a storage snapshot or another physical backup method. The backup is then restored and the PostgreSQL data directory is consistent.
- A file called recovery.signal is created in the data directory. This places the database in read only mode and instructs it to perform the command set out in the restore_command configuration parameter.
- (Optional) Setting the recovery_target_time allows for a more specific point-in-time to be reached. Leaving this configuration parameter unset will apply all archives up to the most recent.

To set the restore_command the variables %f and %p will be provided to the command and PostgreSQL will substitute them with the relevant file names and locations. %f in the string will be related by the name of the file to retrieve from the archive and %p is replaced by the copy destination path name on the server. Once the requirements for point-in-time-recovery are met then the PostgreSQL cluster can be started and will apply the changes to the database and bring it up to date. Once the cluster is fully up to date the recovery.signal file can be removed and the cluster restarted.

```
# Copy from NFS share for Linux/Unix OS
restore_command = 'cp /mnt/server/archivedir/%f "%p"'
# Copy from SMB share for Microsoft Windows
restore_command = 'copy "C:\\server\\archivedir\\%f" "%p"'
# Copy from S3 bucket using s3cmd
restore_command = 's3cmd get s3://BUCKET/path/%f' %p'
```

pgBackRest

pgBackRest is a reliable and easy to use backup and recovery solution for PostgreSQL deployed on. It is open-source with the source code available on GitHub. Crunchy Data provides a public repository to assist with the installation as well as comprehensive documentation on how to use the tool.

pgBackRest provides a number of features that cannot be found in the SQL Dump or File System Level Backup tools such as:

- **Parallel backup and restore:** Achieve the maximum throughput capabilities for backup and recovery operations when utilizing a server with multiple cores.
- **Multiple repositories:** Allows for a more granular control over the locality of backups. With multiple repositories different storage mediums can be used to segregate mediums with different retention times.
- **Full, incremental and differential backups:** Shorten the time it takes to complete backup operations by only backing up the files that have changed since the last full backup. Combining different backup types can lead to both resilient and efficient data protection.
- **Backup integrity:** Files are checked during a restore to ensure that they remain consistent with the files that were backed up.
- **Backup resume:** An interrupted backup operation can be resumed from the point where it was interrupted, there is no need to start a backup from the beginning.
- **Delta restore:** It is possible to restore an isolated database or to shorten restore times by only restoring files that are different to the point when the backup was taken. Parallel processing combined with delta restore can drastically reduce recovery times.
- **Parallel, asynchronous WAL Push & Get:** Sending and retrieving data for the WAL archive can be a time-consuming process. Typically, the WAL operates in a serialized fashion for both backup and recovery. pgBackRest allows for asynchronous sending and retrieval of WAL archive segments. This ensures that systems with a high write volume do not slow down or that during recovery repositories with a higher latency do not slow down the recovery process.
- **Tablespace and link support:** Clusters with tablespaces are fully supported and can be remapped to different locations on recovery.
- **Support for multiple repository types:** Repositories can be located in storage mediums such as Block Storage (with a filesystem), NFS and object storage such as S3.
- **Encryption:** – pgBackRest can encrypt repositories to ensure that sensitive data is protected wherever it is stored.

FlashBlade and pgBackRest

pgBackRest can create repositories on FlashBlade using either NFS or S3 object storage. As FlashBlade compresses any data received in file shares or buckets it is not necessary to use the compression capabilities of pgBackRest where there is enough network bandwidth to support backup and recovery targets.

Combining FlashBlade with pgBackRest for backup and recovery capabilities provides the following benefits for P ostgreSQL environments:

- **Restore at hundreds of TB/hr:** FlashBlade delivers Rapid Restore for production and test/dev workloads with up to 270TB/hr data recovery performance.
- **Multi-Protocol support for file and object:** FlashBlade unifies file and object allowing for either multiple pgBackRest repositories each with a different use case such as taking advantage of file solutions such as SafeMode™ Snapshots.
- **Ransomware remediation with SafeMode Snapshots:** Recover quickly from potential ransomware attacks with the use of SafeMode snapshots. Ransomware cannot delete, modify or encrypt a SafeMode snapshot. In the event of a ransomware attack an organization can quickly recover to a point-in-time where ransomware was not present on a system using a protected and reliable file share storage snapshot.
- **Replication allows for greater resiliency:** File shares and Objects can be replicated to another FlashBlade or even an Amazon Web Services S3 (object only) bucket to provide third site disaster recovery capabilities.

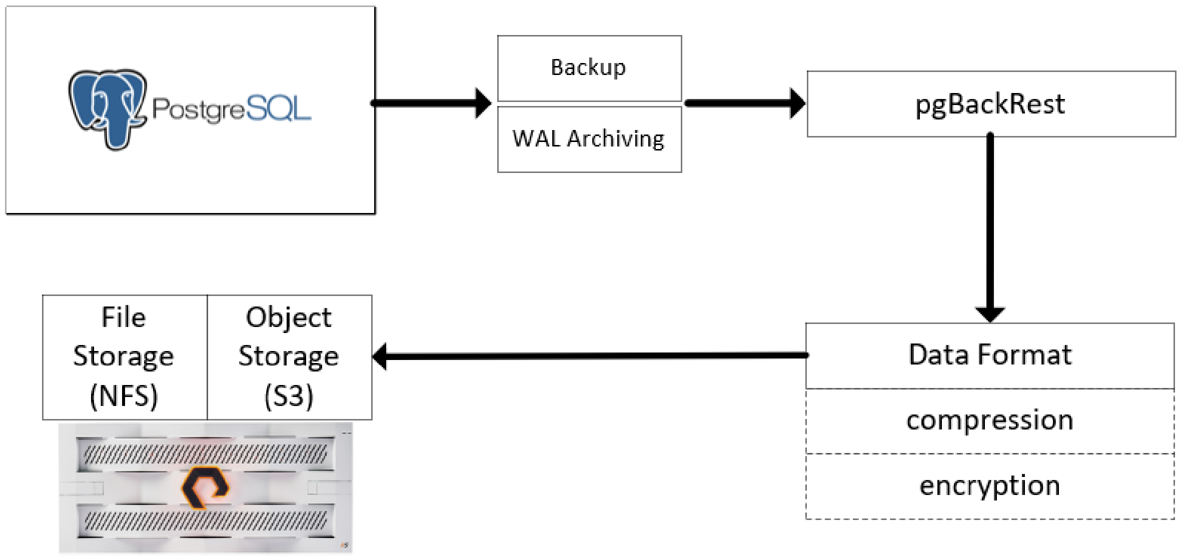


FIGURE 6 pgBackRest data path

The configuration of pgBackRest is done using two concepts: stanzas and repositories.

A stanza is the configuration for a PostgreSQL cluster that defines where it is located, how it will be backed up and associated options. Typical PostgreSQL deployments will have a single cluster on a system and thus if pgBackRest is being run directly from the system with a cluster on it, will only need a single stanza. Where a centralized backup system is present that pgBackRest pulls data from; there could be multiple stanzas, each focusing on a specific cluster. More information on stanza configuration can be found in the pgBackRest documentation. This document will focus on the deployment of pgBackRest on each PostgreSQL cluster without a centralized backup server.

A repository is the storage location where backups and archive WAL segments will be stored. Multiple repositories can be configured to allow for maximum flexibility. For example, a local repository could focus on rapid restore capabilities and a remote repository could be used for long term retention. More information on repositories and the relevant configuration options can be found in the pgBackRest documentation.

To provide stanza and repository configuration a configuration file is created during installation, located at /etc/pgbackrest.conf. When creating the configuration file, the database owner (typically postgres) needs to have at least read privileges on it.

Configuring pgBackRest with FlashBlade

To demonstrate how to configure pgBackRest with FlashBlade a [main] stanza is created on the PostgreSQL cluster. The pg-path variable contains the path to the PostgreSQL cluster data location, this is usually in the form of /var/lib/pgsql/<version>/data. The pg-path configuration will usually have a numeric identifier inside it, for example the first stanza will have pg1-path inside it to identify it. The [main] and [global] sections are present in the configuration file to identify the stanza and repository information. Stanza information is held in the [main] section and repository information and properties are located in the [global] section.

Example NFS configuration: This example assumes that an NFS share has been created on a FlashBlade and mounted to the relevant location in the filesystem. For multiple systems using the same FlashBlade it is possible to use either a single NFS share with multiple PostgreSQL clusters backing up to it, or each PostgreSQL is provided with its own NFS share. Having multiple NFS shares, each focused on a single cluster, can be beneficial where the isolation of backup data is important and a need for risk reduction from ransomware attacks needs to be implemented. Isolating each cluster's backup and recovery data and then using SafeMode Snapshots to protect each file share provides a more granular recovery mechanism that can be useful on a per cluster basis. Consolidating all of the backup data for multiple clusters into a single share provides for easier manageability but does require that each cluster with its own pgBackRest instance has an exclusive folder in the share allocated to it.

This is a simple configuration option for the use of an NFS share on FlashBlade as the backup target.

- The pg1-path is present in the [main] section to provide the data path for the PostgreSQL cluster.
- In the [global] section the repository options are provided with repo1-path offering a folder location on the NFS share as where backups and archived WAL segments will be stored. The folder provided for the repository needs to be owned by the user running the pgBackRest backups, this is most likely the postgres:postgres user and group.
- The process-max=8 option specifies how many processes (in this example 8) should be used to run a standard backup operation. More processes will provide more throughput but also increase the amount of CPU utilization on the system.
- The start-fast=y option forces a checkpoint to complete as soon as possible If start-fast=y is not specified then the backup will only begin after the next scheduled checkpoint.
- Compression is disabled through the use of compress-type=none in this example as there is enough bandwidth available for backup operations and FlashBlade compression provides suitable data reduction.

```
[main]
pg1-path=/var/lib/pgsql/14/data
[global]
repo1-path=/postgres/backup/DB-01/pgbackrest
process-max=8
start-fast=y
compress-type=none
```

Example S3 configuration: This example assumes that the relevant requirements to connect to an S3 bucket on FlashBlade have been met:

- An Object Store Account has been created
- A User has been created in the Object Store account
- An Access Key has been created for the User in the Object Store Account and the Secret Key has been stored in a safe location
- The User has been assigned the access policies required to manage buckets. In this instance the minimum requirements are pure:policy/object-list, pure:policy/object-read, pure:policy/object-write, pure: policy/object-delete
- A bucket has been created in the Object Store Account

There is a 1:1 mapping of PostgreSQL cluster to FlashBlade Object Store, only in the event of restoring a backup to a different system should a bucket be used by a different system.

This is a simple configuration option for the use of an S3 bucket on FlashBlade as the backup target.

- The pg1-path is present in the [main] section to provide the data path for the PostgreSQL cluster.
- In the [global] section the repository options are provided with repo1-path offering a path location to the repository. In the case of S3 Object Storage, this path is appended to the URL and created by pgBackRest if not present.
- The repository type is specified using repo1-type=s3. This is only done for cloud or object storage providers.
- The endpoint is provided with the repo1-s3-endpoint= configuration option. It is recommended that this is a DNS resolvable name or an entry in the local systems host file.
- The repository storage certificate verify option is disabled using repo1-s3-verify-tls=n due to the use of a self-signed certificate. It is recommended to only use this for testing purposes. The FlashBlade certificate can be exported and added to the local system running pgBackRest to ensure that this option is not required.
- The access key is specified using repo1-s3-key=.
- The secret key is specified using repo1-s3-key-secret=.
- The region is specified to be on-prem with the repo1-s3-region= configuration option.
- FlashBlade S3 uses a path style URI's and thus pgBackRest needs to be informed of this using repo1-s3-uri-style=path.
- The process-max=8 option specifies how many processes (in this example 8) should be used to run a standard backup operation. More processes will provide more throughput but also increase the amount of CPU utilization on the system.
- The start-fast=y option forces a checkpoint to complete as soon as possible If start-fast=y is not specified then the backup will only begin after the next scheduled checkpoint.
- Compression is disabled through the use of compress-type=none in this example as there is enough bandwidth available for backup operations and FlashBlade compression provides suitable data reduction.

```
[main]
pg1-path=/var/lib/pgsql/14/data
[global]
repo1-path=/repo1
repo1-type=s3
repo1-s3-endpoint=fb.soln.local
repo1-s3-bucket=db01
repo1-s3-verify-tls=n
repo1-s3-key=PSFBSAZRCLFEHECOLLNPFNIFNNDBBAPGAOKHMGCIN
repo1-s3-key-secret=5DBFC7AE6f0113d/7eae7BDCDF4Adc199780DJEI
repo1-s3-region=on-prem
repo1-s3-uri-style=path
```

Creating the pgBackRest Configuration

Creating the pgBackRest configuration requires that the stanza-create command is run as the postgres user. PostgreSQL must be running for this step. To create the stanza simply execute pgbackrest with the stanza name and command stanza-create. If successful the command should return with “completed successfully” in the response message.

```
# pgbackrest --stanza=main --log-level-console=info stanza-create
2023-02-19 05:34:01.956 P00  INFO: stanza-create command begin 2.35: --exec-id=103709-eee16bc9 --log-
level-console=info --log-level-file=debug --pg1-path=/var/lib/pgsql/14/data --repo1-path=/repo1 --repo1-s3-
bucket=db10 --repo1-s3-endpoint=fb.soln.local --repo1-s3-key=<redacted> --repo1-s3-key-secret=<redacted>
--repo1-s3-region=on-prem --repo1-s3-uri-style=path --no-repo1-storage-verify-tls --repo1-type=s3
--stanza=main
2023-02-19 05:34:02.567 P00  INFO: stanza-create for stanza 'main' on repo1
2023-02-19 05:34:02.636 P00  INFO: stanza-create command end: completed successfully (680ms)
```

Creating a Full Backup

To create a full backup simply execute pgbackrest with the stanza name, backup type of full and command backup. If successful the command should return with "completed successfully" in the response message.

```
# pgbackrest --stanza=main --type=full backup
2023-02-19 05:39:20.576 P00   INFO: backup command begin 2.35: --compress-type=none --exec-id=564857-e15d344c
--log-level-console=info --log-level-file=debug --pg1-path=/var/lib/pgsql/14/data --process-max=16 --repo1-
path=/repo1 --repo1-retention-full=5 --repo1-s3-bucket=db01 --repo1-s3-endpoint=fb.soln.local --repo1-s3-
key=<redacted> --repo1-s3-key-secret=<redacted> --repo1-s3-region=on-prem --repo1-s3-uri-style=path --no-
repo1-storage-verify-tls --repo1-type=s3 --stanza=main --start-fast --type=full
2023-02-19 05:39:21.348 P00   INFO: execute non-exclusive pg_start_backup(): backup begins after the
requested immediate checkpoint completes
2023-02-19 05:39:22.851 P00   INFO: backup start archive = 00000002000000159000000F2, lsn = 159/F2000028
2023-02-19 05:57:03.352 P00   INFO: execute non-exclusive pg_stop_backup() and wait for all WAL segments to
archive
2023-02-19 05:57:03.555 P00   INFO: backup stop archive = 00000002000000159000000F2, lsn = 159/F2000170
2023-02-19 05:57:03.560 P00   INFO: check archive for segment(s) 00000002000000159000000F2:000000020000015900
0000F2
2023-02-19 05:57:03.755 P00   INFO: new backup label = 20230219-053921F
2023-02-19 05:57:03.931 P00   INFO: full backup size = 1361.2GB, file total = 10111
2023-02-19 05:57:03.935 P00   INFO: backup command end: completed successfully (1063360ms)
```

Creating a Differential Backup

To create a full backup simply execute pgbackrest with the stanza name, back up type of diff and command backup. If successful, the command should return with "completed successfully" in the response message.

Differential backups only back up files that have changes since the previous full backup.

```
# pgbackrest --stanza=main --type=diff backup
2023-02-19 09:24:33.629 P00   INFO: backup command begin 2.35: --compress-type=none --exec-id=462215-4950ca58
--log-level-console=info --log-level-file=debug --pg1-path=/var/lib/pgsql/14/data --process-max=16 --repo1-
path=/repo1 --repo1-retention-full=5 --repo1-s3-bucket=db09 --repo1-s3-endpoint=fb.soln.local --repo1-s3-
key=<redacted> --repo1-s3-key-secret=<redacted> --repo1-s3-region=on-prem --repo1-s3-uri-style=path --no-
repo1-storage-verify-tls --repo1-type=s3 --stanza=main --start-fast --type=diff
2023-02-19 09:24:34.453 P00   INFO: last backup label = 20230219-085640F, version = 2.35
2023-02-19 09:24:34.453 P00   INFO: execute non-exclusive pg_start_backup(): backup begins after the
requested immediate checkpoint completes
2023-02-19 09:24:38.057 P00   INFO: backup start archive = 0000000100000033000000B9, lsn = 33/B904B798
2023-02-19 09:27:30.343 P00   INFO: execute non-exclusive pg_stop_backup() and wait for all WAL segments to
archive
2023-02-19 09:27:30.544 P00   INFO: backup stop archive = 0000000100000035000000F8, lsn = 35/F8065EB8
2023-02-19 09:27:30.548 P00   INFO: check archive for segment(s) 0000000100000033000000B9:000000010000003500
0000F8
2023-02-19 09:27:32.812 P00   INFO: new backup label = 20230219-085640F_20230219-092434D
2023-02-19 09:27:33.016 P00   INFO: diff backup size = 205.4GB, file total = 10054
2023-02-19 09:27:33.019 P00   INFO: backup command end: completed successfully (179391ms)
```

Creating an Incremental Backup

To create a full backup simply execute pgbackrest with the stanza name, backup type of incr and command backup. If successful the command should return with "completed successfully" in the response message.

Incremental backups back up data that has changed since the last backup, which could be a full backup or an incremental one.

```
# pgbackrest --stanza=main --type=incr backup
2023-02-19 09:30:26.065 P00  INFO: backup command begin 2.35: --compress-type=none --exec-id=465814-a44138bd
--log-level-console=info --log-level-file=debug --pg1-path=/var/lib/pgsql/14/data --process-max=16 --repo1-
path=/repo1 --repo1-retention-full=5 --repo1-s3-bucket=db09 --repo1-s3-endpoint=fb.soln.local --repo1-s3-
key=<redacted> --repo1-s3-key-secret=<redacted> --repo1-s3-region=on-prem --repo1-s3-uri-style=path --no-
repo1-storage-verify-tls --repo1-type=s3 --stanza=main --start-fast --type=incr
2023-02-19 09:30:26.899 P00  INFO: last backup label = 20230219-085640F_2020219-092434D, version = 2.35
2023-02-19 09:30:26.899 P00  INFO: execute non-exclusive pg_start_backup(): backup begins after the
requested immediate checkpoint completes
2023-02-19 09:30:30.504 P00  INFO: backup start archive = 000000010000003700000092, lsn = 37/9275AB40
2023-02-19 09:33:37.596 P00  INFO: execute non-exclusive pg_stop_backup() and wait for all WAL segments to
archive
2023-02-19 09:33:37.797 P00  INFO: backup stop archive = 00000001000003A00000010, lsn = 3A/1063B138
2023-02-19 09:33:37.801 P00  INFO: check archive for segment(s) 000000010000003700000092:000000010000003A00
000010
2023-02-19 09:33:40.520 P00  INFO: new backup label = 20230219-085640F_20230219-093026I
2023-02-19 09:33:40.739 P00  INFO: incr backup size = 220.5GB, file total = 10054
2023-02-19 09:33:40.741 P00  INFO: backup command end: completed successfully (194677ms)
```

Recovering from a Backup

Once a set of backups have been created, they can be used to recover to the system from where the backup was taken or to a different system entirely.

To view backups which have already been created, execute pgbackrest with the stanza name and command info. All backups available within the configured repositories should be shown.

```
pgbackrest --stanza=main info
stanza: main
  status: ok
  cipher: none
  db (current)
    wal archive min/max (13): 000000010000000000000001/000000010000010E0000007F
    full backup: 20230220-021657F
      timestamp start/stop: 2023-02-20 02:16:57 / 2023-02-20 03:39:14
      wal start/stop: 000000010000010E00000041 / 000000010000010E00000072
      database size: 1094.4GB, database backup size: 1094.4GB
      repo1: backup set size: 429.6GB, backup size: 429.6GB
    full backup: 20230220-034534F
      timestamp start/stop: 2023-02-20 03:45:34 / 2023-02-20 04:01:05
      wal start/stop: 000000010000010E00000077 / 000000010000010E0000007D
      database size: 1094.4GB, database backup size: 1094.4GB
      repo1: backup set size: 429.6GB, backup size: 429.6GB
    full backup: 20230220-050720F
      timestamp start/stop: 2023-02-20 05:07:20 / 2023-02-20 05:23:57
      wal start/stop: 000000010000010E0000007F / 000000010000010E0000007F
      database size: 1094.4GB, database backup size: 1094.4GB
      repo1: backup set size: 429.6GB, backup size: 429.6GB
```

In order to restore a specific backup, the PostgreSQL cluster must first be shut down otherwise the following error message will be shown:

```
ERROR: [038]: unable to restore while PostgreSQL is running
HINT: presence of 'postmaster.pid' in '/var/lib/pgsql/14/data' indicates PostgreSQL is running.
HINT: remove 'postmaster.pid' only if PostgreSQL is not running.
```

All files should be removed from the PostgreSQL data directory prior to executing recovery operations. It is possible to skip removal of the files in the data directory through the use of a delta recovery if data is being recovered to the system it was taken from and the objective is to restore objects which have been lost. A delta recovery will check for any files which have changed since the backup was taken and automatically remove them, thus only restoring the required files which have changed and shortening the recovery operation.

To restore a specific backup, execute pgbackrest with the set option, specifying the backup ID from any entry provided in the info command output, the process-max option if required to decrease the recovery and the restore command. If successful the command should return with "completed successfully" in the response message.

```
pgbackrest --stanza=main --process-max=16 --set=20230220-050720F restore
2023-02-20 06:24:31.162 P00  INFO: restore command begin 2.35: --exec-id=259668-3568432c --log-level-
console=info --pg1-path=/var/lib/pgsql/14/data --process-max=16 --repo1-path=/repo1 --repo1-s3-bucket=db01
--repo1-s3-endpoint=fb.soln.local --repo1-s3-key=<redacted> --repo1-s3-key-secret=<redacted> --repo1-s3-
region=on-prem --repo1-s3-uri-style=path --no-repo1-storage-verify-tls --repo1-type=s3 --set=20230220-050720F
--spool-path=/var/lib/pgsql/14/data/spool --stanza=main
2023-02-20 06:24:31.286 P00  INFO: repo1: restore backup set 20230220-050720F
2023-02-20 06:34:47.615 P00  INFO: write updated /var/lib/pgsql/14/data/postgresql.auto.conf
2023-02-20 06:34:47.620 P00  INFO: restore global/pg_control (performed last to ensure aborted restores
cannot be started)
2023-02-20 06:34:47.621 P00  INFO: restore size = 1094.4GB, file total = 10080
2023-02-20 06:34:47.625 P00  INFO: restore command end: completed successfully (616463ms)
```

Configure Retention

Retention is configured on a per repository basis for specific backup types. The repo-retention-* options are the properties that define how long backups will be retained for. This is provided in the form of a count or time, where both are a number. With the count option when backups of the specified type exceed the count the oldest backups of that type will be expired and removed from the repository.

To set retention to only retain five previous full differential and incremental backups use the following in the configuration file:

```
[global]
repo1-retention-full-type=count
repo1-retention-diff-type=count
repo1-retention-incr-type=count
repo1-retention-full=5
repo1-retention-diff=5
repo1-retention-incr=5
```

With the time option then when the number of days has passed since a backup has been created, then it will be expired. To set retention to only retain 7 days' worth of previous full, differential and incremental backups use the following in the configuration file:

```
[global]
repo1-retention-full-type=time
repo1-retention-diff-type=time
repo1-retention-incr-type=time
repo1-retention-full=5
repo1-retention-diff=5
repo1-retention-incr=5
```

If retention is configured properly then every time a backup is run there will be an expiry command run after a backup is complete.

```
2023-02-19 10:53:24.985 P00  INFO: expire command begin 2.35: --exec-id=487989-3f259247 --log-level-
console=info --log-level-file=debug --repo1-path=/repo1 --repo1-retention-full=5 --repo1-s3-bucket=db09
--repo1-s3-endpoint=fb.soln.local --repo1-s3-key=<redacted> --repo1-s3-key-secret=<redacted> --repo1-s3-
region=on-prem --repo1-s3-uri-style=path --no-repo1-storage-verify-tls --repo1-type=s3 --stanza=main
2023-02-19 10:53:25.013 P00  INFO: expire command end: completed successfully (28ms)
```

Using pgBackRest with WAL Segment Archiving

Backup

To back up a running PostgreSQL cluster with pgBackRest WAL archiving needs to be enabled. The archive-push command for pgbackrest is used to send WAL segments to the repository for archiving. The WAL archiving options in postgresql.conf can be configured as follows for the main stanza:

```
archive_command = 'pgbackrest --stanza=main archive-push %p'
archive_mode = on
listen_addresses = '*'
log_line_prefix = ''
max_wal_senders = 10
wal_level = replica
```

For WAL archiving to work properly the stanza needs to be created and the repository available.

Recovery

To use archive segments backed up by pgBackRest for point-in-time-recovery when combined with solutions like storage snapshots there are only two requirements to ensure the segments are used to roll forward to the point in time:

- Post restoration of the storage snapshot prior to starting the database ensure that a file named recovery.signal is created in the PostgreSQL cluster data directory.
- In the PostgreSQL configuration file (postgresql.conf) ensure that the restore command is set as follows:

```
restore_command = 'pgbackrest --stanza=main archive-get %f "%p"'
```

Once these requirements have been met then the PostgreSQL cluster can be started. In the event that pgBackRest is performing a recovery from a full, differential or incremental backup it will create the recovery.signal file , there is still a requirement to set the restore command.

Asynchronous WAL Segment Archiving

Typical archiving is a sequential and synchronous operation that only deals with one archive segment at a time. pgBackRest can convert this process into a parallel and asynchronous operation that is much more efficient for both backup and restore operations.

To enable asynchronous archiving the pgBackRest [global] configuration needs to contain an entry for archive-async=y and a spool-path filesystem location. The spool path is used to store transient data and should be stored on the same filesystem as pg_xlog/pg_wal for the greatest efficiency.

```
[global]
archive-async=y
spool-path=/var/lib/pgsql/14/data/spool
```

Once asynchronous archiving has been configured the backup/recovery performance of archived WAL segments can be optimized by increasing the process-max configuration parameter of the archive-get and archive-push commands:

```
[global:archive-get]
process-max=4
[global:archive-push]
process-max=8
```

Using Compression

To use compression in scenarios where bandwidth reduction is required the compress-type option can be used. The default compression is the gzip format but there is also support for bz2, lz4 and zst.

To configure compression for backups, add the compress-type to the pgbackrest configuration file under the [global] section:

```
[global]
compress-type=gz
```

To scale performance when using compression, the process-max parameter may need to be increased for backups. CPU usage when using compression will be high thus it should not be done when using pgBackRest directly from a production system when it is under heavy load.

Barman

Barman is an open source administration tool for the disaster recovery of PostgreSQL databases. It allows for organizations to orchestrate the remote backup and recovery of PostgreSQL servers. Barman provides features such as full hot physical backup of PostgreSQL clusters, point-in-time recovery, remote backup via rsync/SSH or pg_basebackup, support for remote or local recovery, support for WAL archiving and streaming, incremental backup and recovery and WAL file compression.

Additional information can be found on the Barman website.

Using Barman with FlashBlade NFS

After downloading and installing the Barman packages from any of the sources, follow the below steps to create a backup :

1. Create an NFS share on FlashBlade, export it to the relevant PostgreSQL cluster and then mount it at a relevant location. This example uses /var/lib/barman
2. Switch to the postgres user in a command shell and then run the following command to create the barman operating system user : createuser -s -P barman
3. Switch to the barman user and then create a .pgpass file. This file is used by Barman to read authentication information for various PostgreSQL databases. The contents of the file should follow this syntax : Hostname:port:database:username:password. The following is an example of a file for any host on any port using the postgres database and barman user :

```
*:*:postgres:barman:barmanpass
```

4. As the root operating system user edit the barman configuration file, /etc/barman.conf, to have the below information :

```
[barman]
barman_user = barman
configuration_files_directory = /etc/barman.d
barman_home = /var/lib/barman
barman_lock_directory = /var/run/barman
log_file = /var/log/barman/barman.log
log_level = INFO
network_compression = true
streaming_archiver = on
archiver = on
```

5. As the root operating system user create a file, /etc/barman.d/pg.conf, with the following contents :

```
[pg]
description = "PostgreSQL server"
conninfo = host=localhost user=barman dbname=postgres password=barmanpass
streaming_conninfo = host=localhost user=barman password=barmanpass
backup_method = postgres
slot_name = barman
```

6. As the barman user run the command barman receive-wal --create-slot pg to create the WAL slots
7. Edit postgres.conf to enable WAL archiving with the archive_command 'barman-wal-archive backup pg %p'. Ensure that the wal_level is replica.Restart the PostgreSQL cluster to apply these changes.
8. As the root operating system user, create the /var/run/barman lock directory and ensure it is owned by the barman user (chown barman:barman /var/run/barman).
9. Ensure that the barman home directory (where backups are stored and the FlashBlade NFS export is mounted) is owned by the barman user (chown barman:barman /var/run/barman).
10. As the barman user create or edit the bash profile file (.bash_profile) to include the PostgreSQL in the PATH variable (export PATH=$PATH:/usr/pgsql-14/bin)
11. To verify the configuration use the barman check pg command

```
barman check pg
Server pg:
    PostgreSQL: OK
    superuser or standard user with backup privileges: OK
    PostgreSQL streaming: OK
    wal_level: OK
    replication slot: OK
    directories: OK
    retention policy settings: OK
    backup maximum age: OK (no last_backup_maximum_age provided)
    backup minimum size: OK (0 B)
    wal maximum age: OK (no last_wal_maximum_age provided)
    wal size: OK (0 B)
    compression settings: OK
    failed backups: OK (there are 0 failed backups)
    minimum redundancy requirements: OK (have 0 backups, expected at least 0)
    pg_basebackup: OK
    pg_basebackup compatible: OK
    pg_basebackup supports tablespaces mapping: OK
    systemid coherence: OK (no system Id stored on disk)
    pg_receivexlog: OK
    pg_receivexlog compatible: OK
    receive-wal running: OK
    archive_mode: OK
    archive_command: OK
    continuous archiving: OK
    archiver errors: OK
```

12. As the barman user run the following command to switch the log and then run the barman receive-wal pg command to start it:

```
barman switch-xlog --force --archive pg
barman cron
```

13. As the barman user run the barman backup pg command to run a backup :

```
barman backup pg
Starting backup using postgres method for server pg in /var/lib/barman/pg/
base/20230123T051907
Backup start at LSN: 19C/E0000148 (0000001A0000019C000000E0, 00000148)
Starting backup copy via pg_basebackup for 20230123T051907
Copy done (time: 41 minutes, 55 seconds)
Finalising the backup.
This is the first backup for server pg
WAL segments preceding the current backup have been found:
    0000001A0000019C000000DF from server pg has been removed
    0000001A0000019C000000E0 from server pg has been removed
Backup size: 1.0 TiB
Backup end at LSN: 19C/E3000060 (0000001A0000019C000000E3, 00000060)
Backup completed (start time: 2023-01-23 05:19:07.639488, elapsed time: 41 minutes, 57
seconds)
Processing xlog segments from streaming for pg
    0000001A0000019C000000E3
```

FlashBlade Replication for Greater Resiliency

FlashBlade allows for the replication of file and object storage to provide greater resiliency for critical storage operations. File replication enables scalable snapshot-based protection between two or more FlashBlade storage appliances while Object Replication allows for objects stored in S3 buckets to be replicated to another FlashBlade or Amazon Web Services.

Combining data protected with the various PostgreSQL backup and recovery methods with hardware replication allows organizations to rest assured that in the event of any disaster there will always be a path to remediation and recovery.

The PostgreSQL tools for SQL dump (pg_dump/pg_dumpall), physical backups(pg_basebackup) and any continuous archiving method using file system copy commands can only use file-based replication (Figure 6) while pgBackRest can use both file and object replication functionality (Figure 7) on FlashBlade.

File Replication

File replication is the asynchronous replication of snapshots from a source FlashBlade to a target FlashBlade. This replication provides the following capabilities:

- Disaster recovery capabilities with ongoing protection against site or data center failures.
- Efficient distribution of data through the. snapshot directory that can be restored to intended replicas for streaming replication or other use cases

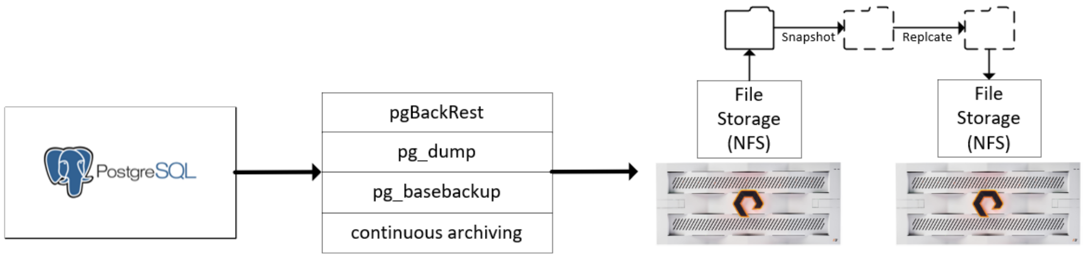


FIGURE 7 PostgreSQL tools and file replication

The FlashBlade user guide provides information on how to implement and use file replication on FlashBlade in Chapter 5 : File Replication Failover , Reprotect and Failback.

Object Replication

FlashBlade object replication can be used to asynchronously copy an object in a bucket to a different FlashBlade or Amazon S3. Objects are automatically replicated as soon as they are written to the source bucket with no need to manage complex policies. This replication provides the following capabilities:

- Disaster recovery capabilities with ongoing protection against site or data center failures.
- No need for additional hardware when implementing object copy to Amazon S3.

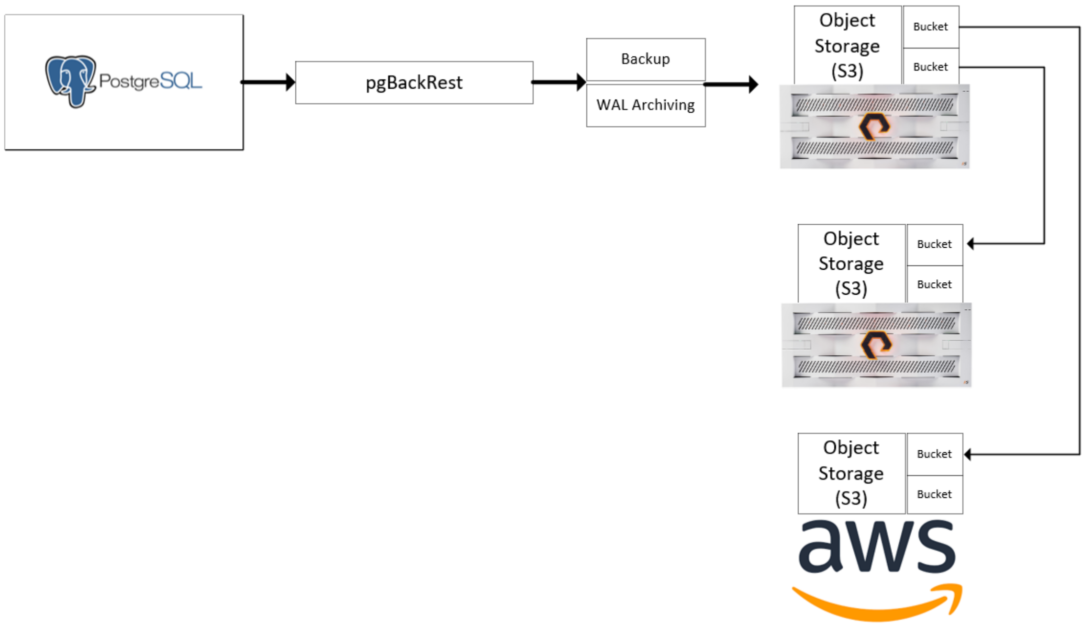


FIGURE 8 pgBackRest hardware replication for file and object

The FlashBlade user guide provides information on how to implement and use object replication on FlashBlade in Appendix B Replication and Recovery.

During a disaster scenario pgBackRest can be reconfigured to use the AWS S3 bucket that FlashBlade is replicating to. To configure the tool to use the objects replicated to AWS S3 see the pgBackRest administration guide for S3-Compatible Object Store Support.

Ransomware Remediation with SafeMode Snapshots

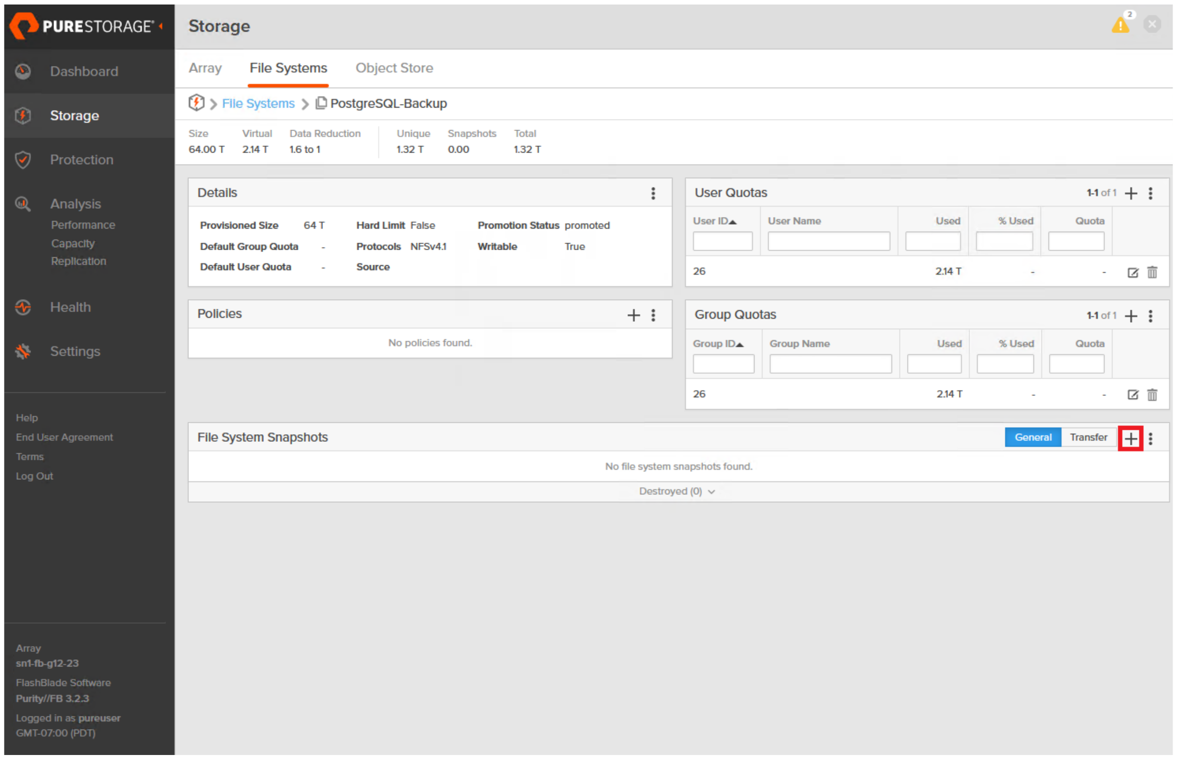
FlashBlade SafeMode Snapshots creates a separation of power between FlashBlade administration functions and the eradication of data. SafeMode Snapshots only apply to File data and can be combined with replication to offer superior resilience to data loss and integrity concurrently.

Filesystem snapshots on FlashBlade are immutable and cannot be changed. The snapshot can be used to overwrite the parent filesystem but the state of the snapshot will remain unchanged from the moment it was taken. This assists with ransomware remediation as data cannot be changed or affected by malicious software attempting to encrypt it and will thus provide a recovery point that organizations can rest assured has not been compromised.

To implement SafeMode Snapshots administrators need to be aware of and utilize the snapshot capability for file shares in FlashBlade.

Creating a File System Snapshot

To create a snapshot in the FlashBlade graphical user interface a user with the correct permissions needs to be logged in and navigate to the Storage view and select an appropriate filesystem. Once a filesystem has been selected a snapshot can be created by identifying the File System Snapshots section and selecting this + in the top right-hand corner.



Size	Virtual	Data Reduction	Unique	Snapshots	Total
64.00 T	2.14 T	1.6 to 1	1.32 T	0.00	1.32 T

Provisioned Size	64 T	Hard Limit	False	Promotion Status	promoted
Default Group Quota	-	Protocols	NFSv4.1	Writable	True
Default User Quota	-	Source			

User Quotas 1-1 of 1

User ID	User Name	Used	% Used	Quota
26		2.14 T	-	-

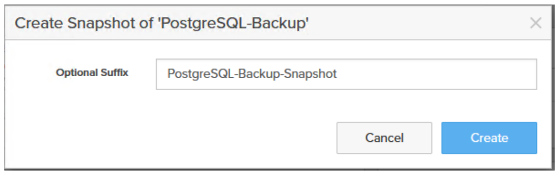
Group Quotas 1-1 of 1

Group ID	Group Name	Used	% Used	Quota
26		2.14 T	-	-

Array sn1-fb-g12-23
FlashBlade Software Purity//FB 3.2.3
Logged in as pureuser
GMT-07:00 (PDT)

FIGURE 9 FlashBlade graphical user interface filesystem view

Once the create snapshot dialog is open for a specific File System an optional suffix can be provided prior to its creation.

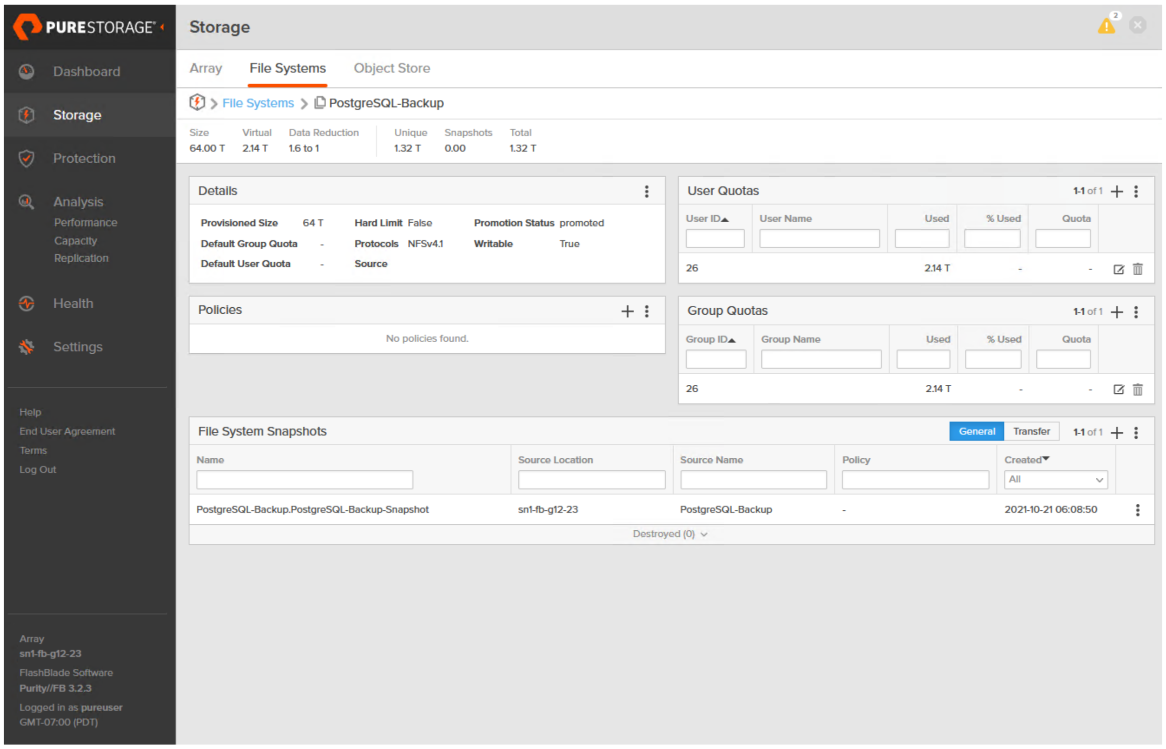


Create Snapshot of 'PostgreSQL-Backup'

Optional Suffix: PostgreSQL-Backup-Snapshot

FIGURE 10 Create a snapshot of file system dialog

Once the filesystem snapshot has been created it will be displayed in the File System Snapshots section.



Array sn1-fb-g12-23
FlashBlade Software Purity//FB 3.2.3
Logged in as pureuser
GMT-07:00 (PDT)

Size	Virtual	Data Reduction	Unique	Snapshots	Total
64.00 T	2.14 T	1.6 to 1	1.32 T	0.00	1.32 T

Details

Provisioned Size	64 T	Hard Limit	False	Promotion Status	promoted
Default Group Quota	-	Protocols	NFSv4.1	Writable	True
Default User Quota	-	Source			

User Quotas 1-1 of 1

User ID	User Name	Used	% Used	Quota
26		2.14 T	-	-

Policies

No policies found.

Group Quotas 1-1 of 1

Group ID	Group Name	Used	% Used	Quota
26		2.14 T	-	-

File System Snapshots — General | Transfer — 1-1 of 1

Name	Source Location	Source Name	Policy	Created
PostgreSQL-Backup.PostgreSQL-Backup-Snapshot	sn1-fb-g12-23	PostgreSQL-Backup	-	2021-10-21 06:08:50

Destroyed (0)

FIGURE 11 View created filesystem snapshot

Enabling SafeMode Snapshots

SafeMode snapshots will ensure that any Snapshot will not be eradicated for a set period of time. This allows for a period of time where data can be recovered without being changed.

To configure SafeMode snapshots only Pure Storage support can perform the required operations. This ensures that no local user can compromise a system and disable the functionality without being a trusted contact of the organization.

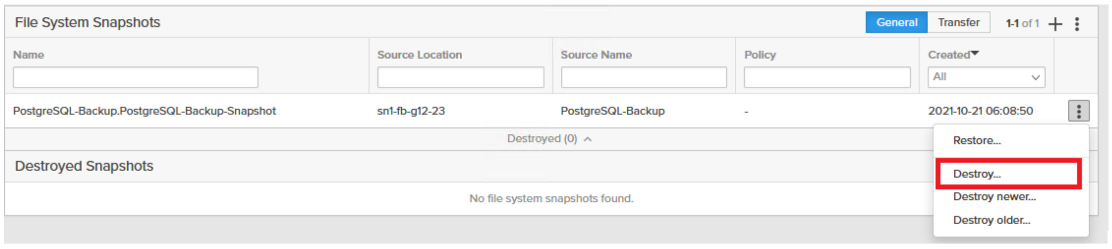
Deleting and Eradicating a SafeMode Snapshot

In FlashBlade there are a number of steps that must be performed before data is completely eradicated:

1. A File System or snapshot must first be deleted. The deletion simply places it in a queue that will delay the complete destruction of the data for a set period of time. During this time the File System or snapshot can still be recovered and used as if it were never deleted.
2. Once deleted a File System is still on the system until it is eradicated by a user or a set period of time passes, where eradication is the complete destruction of the data.

In summary, file system snapshots on FlashBlade are constrained where data from it is not eradicated until a set period of time passes or a user manually eradicates it. Where SafeMode Snapshots protect this data is that eradication is completely blocked from being executed by a user until a set period of time has passed. When contacting Pure Storage support to enable SafeMode Snapshots organizations need to provide a minimum period of time that snapshots must be retained for. This period of time will be used to specify how long File System snapshots need to be retained before they can be completely eradicated.

With SafeMode Snapshots enabled a Filesystem Snapshot can still destroyed in the File System view:


File System Snapshots

Name	Source Location	Source Name	Policy	Created
PostgreSQL-Backup.PostgreSQL-Backup-Snapshot	sn1-fb-g12-23	PostgreSQL-Backup	-	2021-10-21 06:08:50

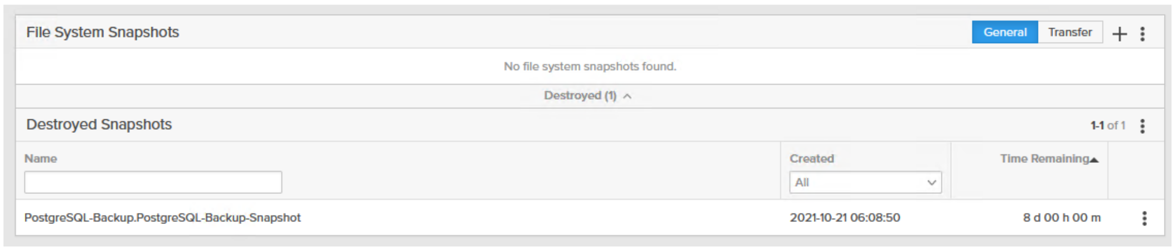
Destroyed (0)

Destroyed Snapshots

No file system snapshots found.

FIGURE 12 Destroying a file system snapshot

Once destroyed a Filesystem Snapshot will show in the Destroyed Snapshots section.


File System Snapshots

No file system snapshots found.

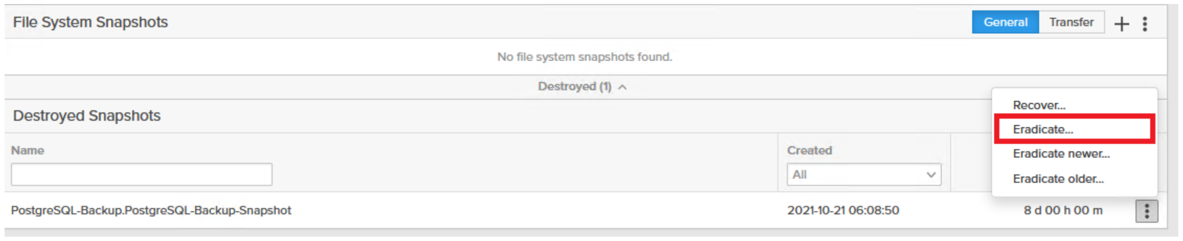
Destroyed (1)

Destroyed Snapshots

Name	Created	Time Remaining
PostgreSQL-Backup.PostgreSQL-Backup-Snapshot	2021-10-21 06:08:50	8 d 00 h 00 m

FIGURE 13 Viewing destroyed filesystem snapshots

Selecting the three ellipses on the right-hand side of the destroyed snapshot will bring up the context menu. This menu will allow for eradication or recovery of the snapshot.

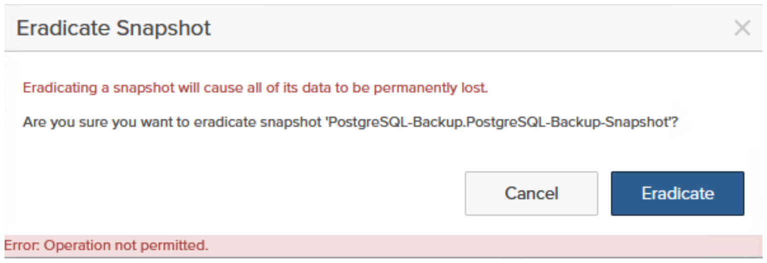


No file system snapshots found.

Name	Created	
PostgreSQL-Backup.PostgreSQL-Backup-Snapshot	2021-10-21 06:08:50	8 d 00 h 00 m

FIGURE 14 Eradicating file system snapshot

A dialog is always shown prior to eradication, when SafeMode snapshots are enabled, this activity is blocked with an error: Operation not permitted.



Eradicating a snapshot will cause all of its data to be permanently lost.

Are you sure you want to eradicate snapshot 'PostgreSQL-Backup.PostgreSQL-Backup-Snapshot'?

Error: Operation not permitted.

FIGURE 15 With SafeMode Snapshots enabled, eradication of a snapshot is blocked while within the time remaining

zThe section on PostgreSQL Backup and Recovery tools covers at length the tools that can be used to protect PostgreSQL clusters. This section will highlight how each of the tools performs under different scenarios with FlashBlade as the target. Details on the resources used in the test cases can be found in the Hardware Environment section. Some of the scenarios explored in this section are:

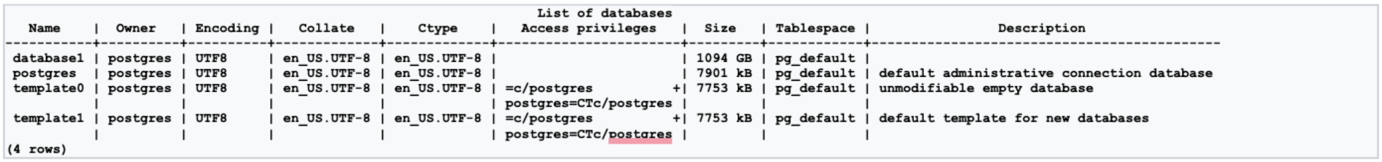
- The backup performance of pg_dump when increasing the number of jobs
- The recovery performance of pg_restore when increasing the number of jobs.
- The backup performance of pg_basebackup
- The backup and recovery performance of pgBackRest with Object Storage repositories
- The impact of compression on pgBackRest backup and recovery performance

In each scenario, both single cluster and at scale scenarios will be evaluated. The process of identifying performance in any of the scenarios is done with the following method:

1. Start the relevant scenario through a bash shell, prepending the PostgreSQL backup tool with the time command.
2. When the command ends, take note of the real portion of the time output. For scenarios with multiple clusters the longest time value is taken.
3. Note the amount of database data backed up, this output can be taken using the \l+ command in the psql utility under the size column. For scenarios with multiple clusters all of the database values are taken.
4. Divide the amount of database data backed up/recovered by the amount of real time that has elapsed in seconds. The MB/s value provided is then converted into TB/hour.

Ascertaining the impact on capacity is done by taking the database size and then analyzing the impact on capacity utilized on FlashBlade.

For the test environment, separate PostgreSQL clusters are created on isolated hosts. Each cluster is populated with 1 terabyte of database data in a single database.



```
                                                      List of databases
   Name    |  Owner   | Encoding |   Collate   |    Ctype    |   Access privileges   |  Size   | Tablespace |                Description
-----------+----------+----------+-------------+-------------+-----------------------+---------+------------+--------------------------------------------
 database1 | postgres | UTF8     | en_US.UTF-8 | en_US.UTF-8 |                       | 1094 GB | pg_default |
 postgres  | postgres | UTF8     | en_US.UTF-8 | en_US.UTF-8 |                       | 7901 kB | pg_default | default administrative connection database
 template0 | postgres | UTF8     | en_US.UTF-8 | en_US.UTF-8 | =c/postgres          +| 7753 kB | pg_default | unmodifiable empty database
           |          |          |             |             | postgres=CTc/postgres |         |            |
 template1 | postgres | UTF8     | en_US.UTF-8 | en_US.UTF-8 | =c/postgres          +| 7753 kB | pg_default | default template for new databases
           |          |          |             |             | postgres=CTc/postgres |         |            |
(4 rows)
```

pg_dump and pg_restore

The logical backup and recovery utilities pg_dump and pg_restore are discussed at length in the section for SQL dump. All test cases and results here utilized Network File System(NFS) shares on FlashBlade.

Backup Performance with pg_dump

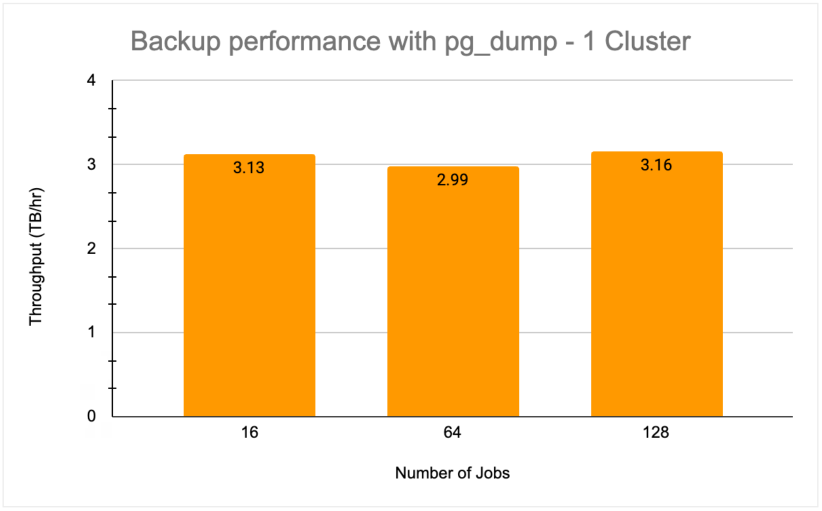


FIGURE 16 Backup performance for a single cluster with pg_dump

The pg_dump utility increases performance through the use of the jobs argument where each job represents a unique connection to the database. The best performance for a single cluster was achieved using 128 jobs, which yielded 3.16TB/hour performance.

This test scenario targeted a single NFS share on FlashBlade.

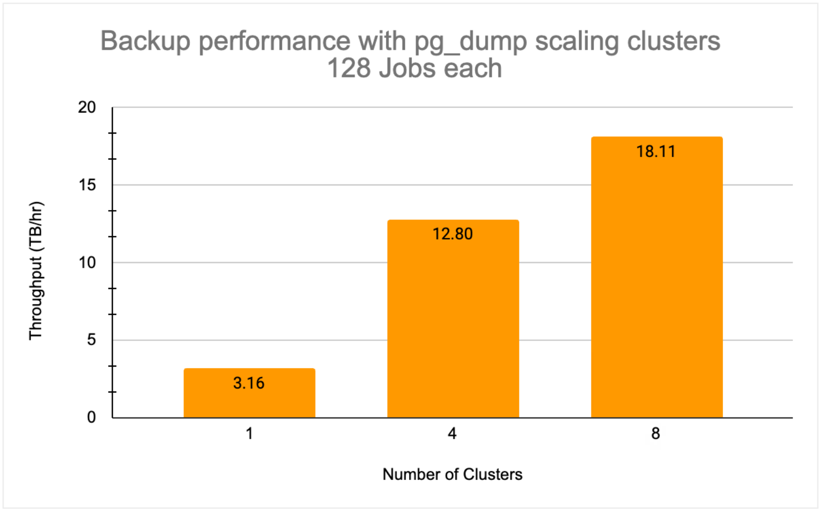


FIGURE 17 Scaling backup performance for pg_dump with multiple clusters

The best backup performance with pg_dump and a single cluster utilized 128 jobs to a single NFS share. To identify the scaling values with pg_dump a single NFS share was used but with each cluster targeting different folders within that share.

Increasing the number of clusters running pg_dump concurrently yielded backup performance of 12.8TB/hour with 4 clusters and 18.11TB/hour with 8.

Recovery Performance with pg_restore

As with pg_dump, pg_restore is capable of increasing recovery performance through the use of the jobs argument. A single database was restored from a backup created with pg_dumpusing the directory format on a FlashBlade NFS share.

While increasing the jobs argument did increase recovery performance, there was a limit as to how quickly pg_restore could recover the database. As pg_restore is a logical recovery utility it is repeating data definition (DDL) and data manipulation commands (DML) to place database objects and contents into the specific catalog.

With 64 jobs, the recovery performance for a single cluster was 1.1TB/hour.

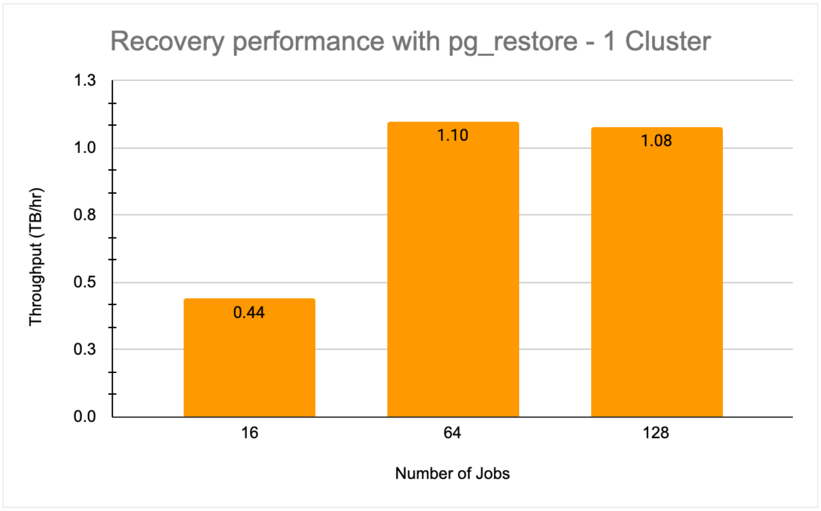


FIGURE 18 Recovery performance for a single cluster using pg_restore

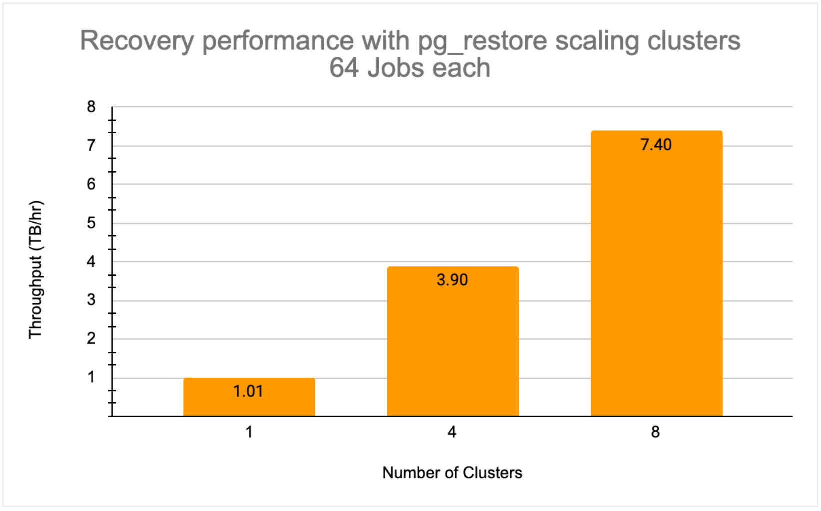


FIGURE 19 Scaling recovery performance for pg_restore with multiple clusters

Using the best jobs configuration with a single cluster with 64 jobs, this was scaled to 4 and 8 clusters.

Each cluster restored data from a single NFS share, but with pg_dump data stored in separate folders for each cluster.

Performance scaled linearly with 3.9TB/hour with 4 clusters and 7.4TB/hour with 8 clusters.

Filesystem-level Backup

This test case utilized the pg_basebackup physical backup tool. All test cases and results here utilized Network File System (NFS) shares on FlashBlade.

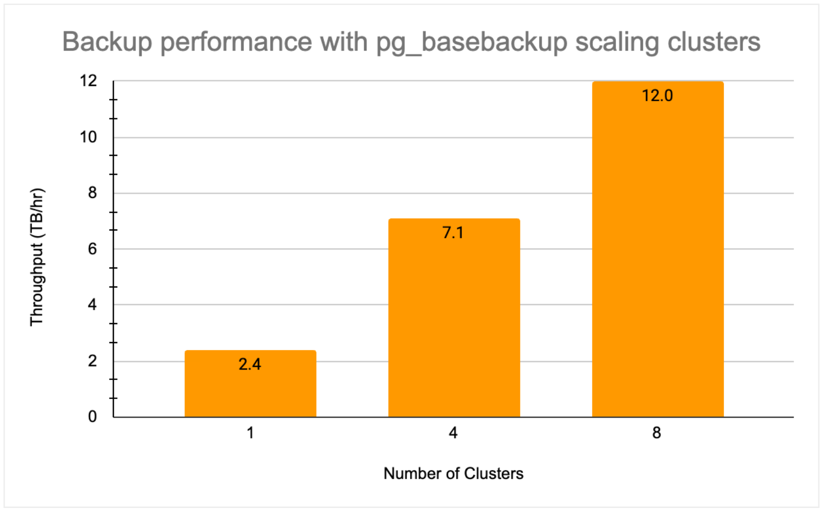


FIGURE 20 Scaling backup performance for pg_basebackup with multiple cluster

The pg_basebackup utility simply copies the contents of the PostgreSQL data directory and any associated tablespaces to a third location. A single cluster achieved 2.4TB/hour and scaled to 7.1TB/hour for 4 clusters and 12TB/hour with 8 clusters.

This performance was achieved using the tar format and WAL streaming mode for a set of online clusters each backing up data to a single NFS share with multiple separate directories for each cluster.

pgBackRest

The test cases and results shown here are isolated to the use of pgBackRest which is discussed in the section of the same name. Test cases here are focused on the use of the S3 Object Storage protocol on FlashBlade.

Backup Performance of pgBackRest

The two options identified to make the most impact to performance are the number of processes and compression. When not utilizing compression, the number of processes can be kept low with a single cluster maximizing its performance at 32 processes and 7.1TB/Hour. Using the same number of processes for backup performance with compression achieved less than half the capability of the non-compressed option (16 processes without compression achieves 4.1TB/hour and 16 processes with compression achieves only 1.5TB/hour). In order to scale the backup performance of compression for a single cluster the number of processes used needed to be four times more than the non-compressed counterpart of 32 processes (128 processes with compression achieves 4.9TB/hour). The impact of both compression and the increased number of processes on the host system was that at 64 processes with compression 65% of the host CPU was utilized and 128 processes with compression utilized 97.8%.

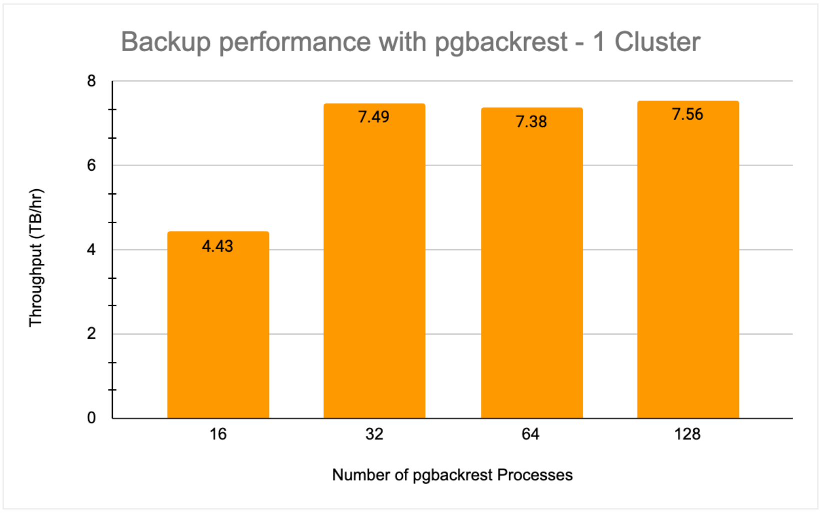


FIGURE 21 pgBackRest backup performance without compression for a single cluster comparing different option

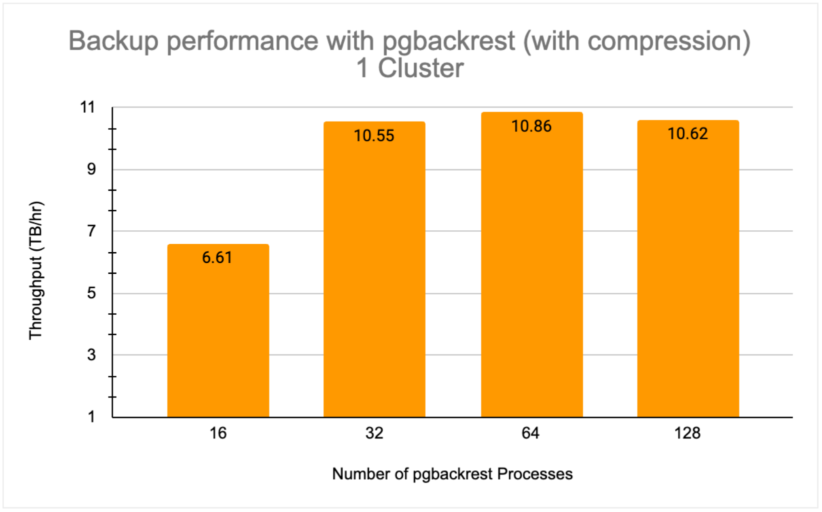


FIGURE 22 pgBackRest backup performance with compression for a single cluster comparing different option

When scaling pgBackRest backups and options. the inverse becomes true; Because compression will utilize less bandwidth and send less data the backup performance at scale increases. The best performance with 8 clusters each backing up to separate S3 buckets on the same FlashBlade without compression achieved 7.3TB/hour using 64 processes and with compression achieving 10.86TB/hour.

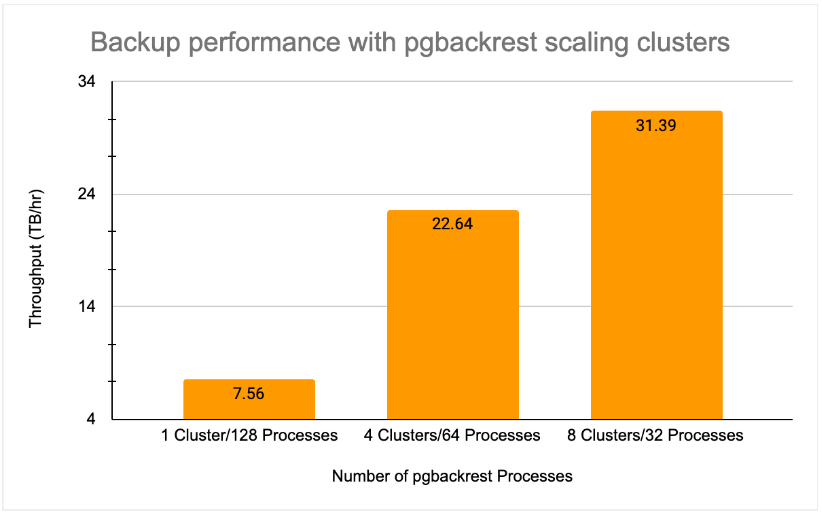


FIGURE 23 pgBackRest backup scaling for multiple clusters without compression using different process configurations

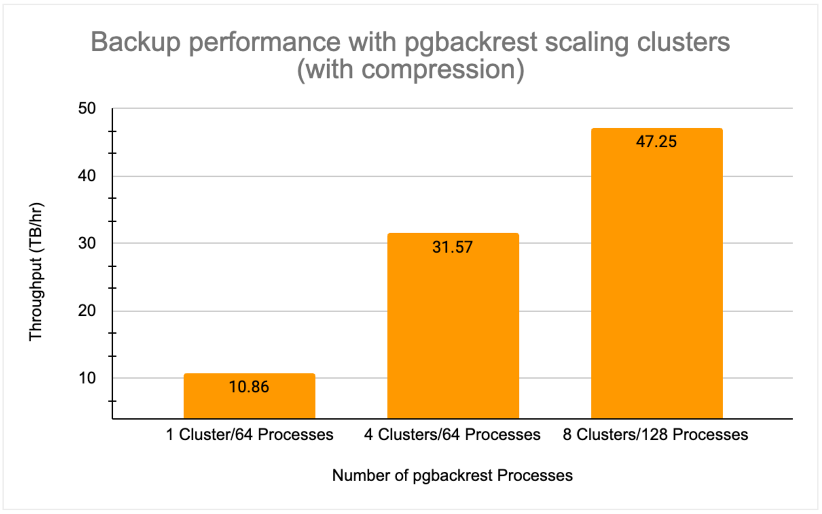


FIGURE 24 pgBackRest backup scaling for multiple clusters with compression using different process configurations

During high database load for a single cluster where there is significant bandwidth available it therefore is advised to not use compression. During low database load where bandwidth may be limited, compression is a suitable method in which to ensure database backups can still be completed in a suitable time.

Where multiple clusters are backing up to the same FlashBlade at the same time compression is useful as it will bypass both bandwidth and ingest speed limitations. However significant insight into workload behavior on the databases should be used to plan backup activities due to the increased CPU load required to achieve the scaling performance.

Recovery Performance of pgBackRest

When recovering from a pgBackRest backup, the entire cluster is offline and thus we advise using all the available system resources to ensure recovery takes as little time as possible. An important aspect to note is that the recovery instruction is the same regardless of how a backup was created, with the exception that the number of processes can be increased. A compressed backup will use the same command, but will utilize less bandwidth and more CPU cycles.

When recovering a single cluster, the best-case scenario was found to be 16 processes restoring a compressed backup—achieving 7.9TB/hour. The best non-compressed backup recovery utilized 16 processes and achieved 3.9TB/hour. Utilizing more than 32 processes decreased overall recovery performance when using 64 processes for both compressed and uncompressed backups.

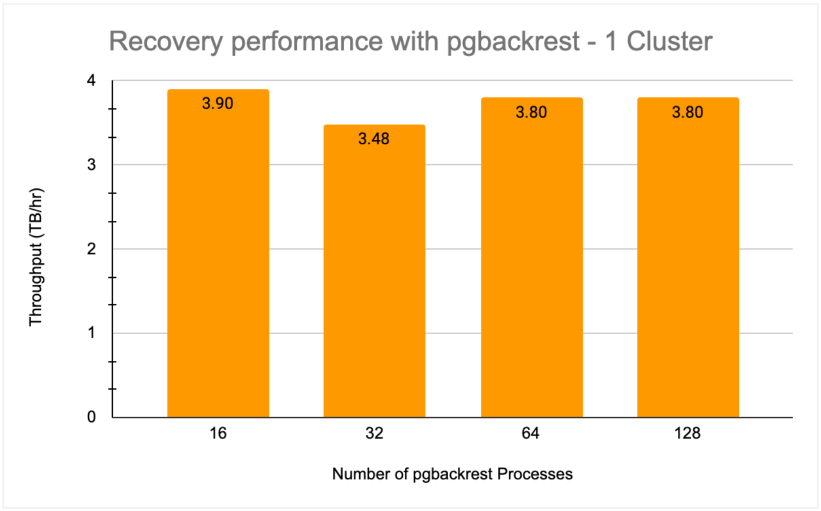


FIGURE 25 pgBackRest recovery scaling for a single cluster using with different process configurations

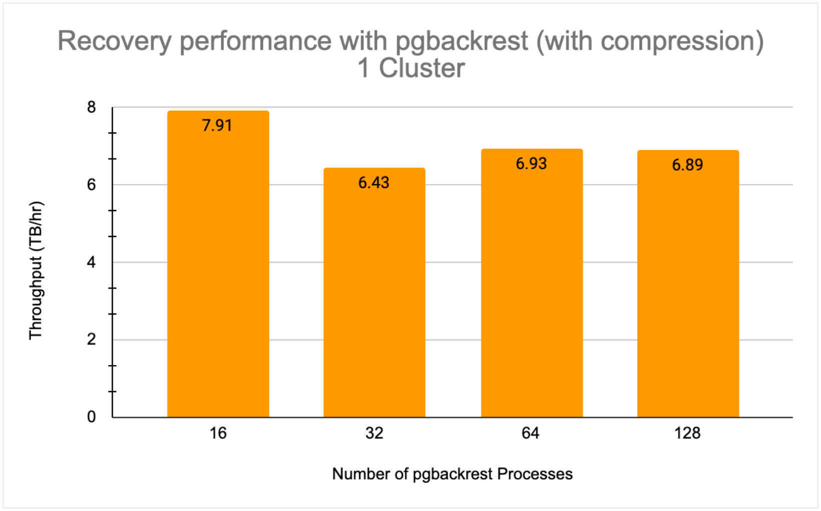


FIGURE 26 pgBackRest recovery scaling for a single cluster using compression with different process configurations

Where FlashBlade shows its greatest performance advantage is restoring multiple systems at scale. Isolating the best scenarios for each configuration option it can be seen that a phenomenal recovery scale can be achieved for many scenarios. The best recovery scale for eight systems can be achieved by utilizing compression and 64 processes on each system—providing 31.5TB/hour recovery performance. The recovery performance in this test case was limited by the primary storage used for each cluster.

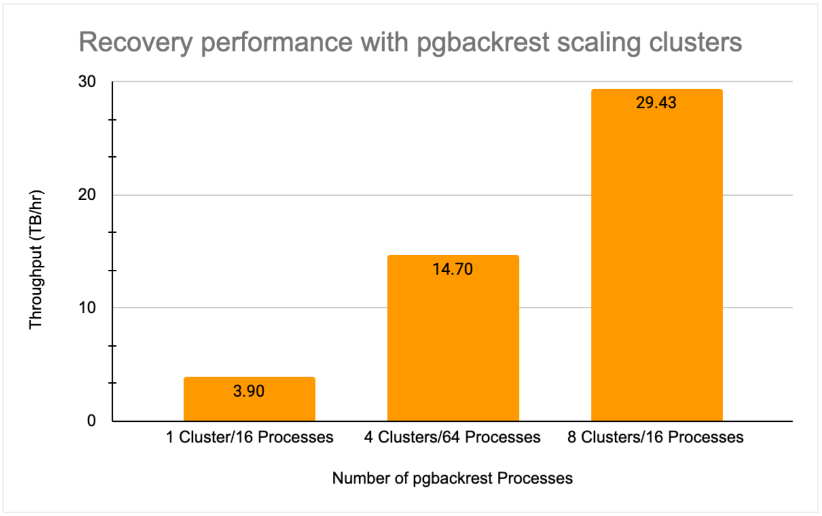


FIGURE 27 pgBackRest recovery scaling for multiple clusters using different process configurations.

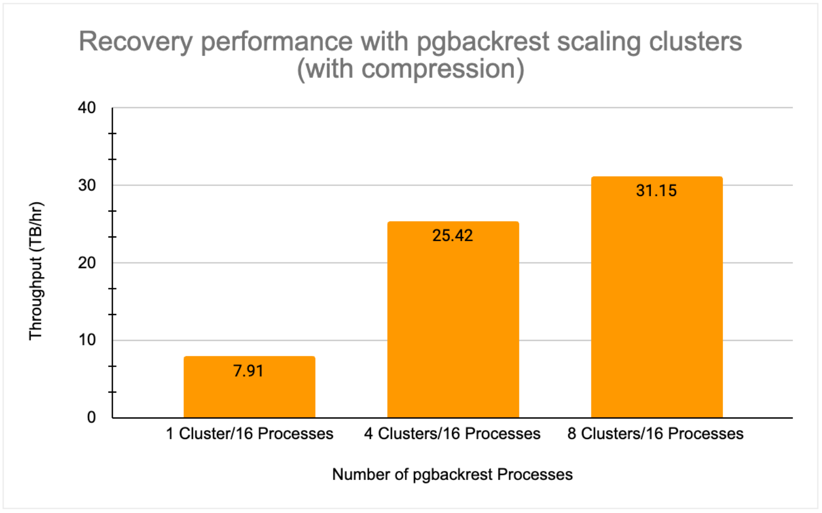


FIGURE 28 pgBackRest recovery scaling for multiple clusters using different process configurations with compression

Conclusion

While there are a number of backup and recovery tools for PostgreSQL the one which stands above all is pgBackRest. Combining pgBackRest and FlashBlade provides a number of benefits including (but not limited to) recovery performance in many terabytes per hour, rich data services for replication and ransomware protection through the use of SafeMode Snapshots. As databases scale and organizations look into further digital transformation, FlashBlade as a scalable, easy to use and unified platform for file and object storage creates many opportunities to simplify and streamline business protection.

purestorage.com

800.379.PURE

PURESTORAGE®
Uncomplicate Data Storage, Forever