Backup Best Practices with pgBackRest

David Steele
Crunchy Data

PGConf.Online 2021
March 1, 2021

Agenda

About the Speaker

- Principal Architect at Crunchy Data, the Trusted Open Source Enterprise PostgreSQL Leader.
- Actively developing with PostgreSQL since 1999.
- Maintainer of pgBackRest and pgAudit.
- PostgreSQL Contributor.

Why Backup?

- Hardware Failure:
  - No amount of redundancy can prevent it.
- Replication:
  - WAL archive for when async streaming gets behind.
  - Sync replica from a backup instead of the primary.
- Corruption:
  - Can be caused by hardware or software.
  - Detection is, of course, a challenge.

Why Backup?

- Accidents:
  - So you dropped a table?
  - Deleted your most important account?
- Development:
  - No more realistic data than production!
  - May not be practical due to size / privacy issues.
- Reporting:
  - Use backups to standup an independent reporting server.
  - Recover important data that was removed on purpose.

Making Backups Useful

- Find a way to use your backups
  - Syncing / New Replicas
  - Offline reporting
  - Offline data archiving
  - Development
- Unused code paths will not work when you need them unless they are tested
  - Regularly scheduled automated failover using backups to restore the old primary
  - Regularly scheduled disaster recovery (during a maintenance window if possible) to test restore techniques

What is pgBackRest?

pgBackRest aims to be a reliable, easy-to-use backup and restore solution that can seamlessly scale up to the largest databases and workloads by utilizing algorithms that are optimized for database-specific requirements.

Major features:

- Parallelism and asynchronous operation
- Full, incremental, and differential backups
- Backup and archive expiration
- Integrity checks / page checksum verification
- Delta restore
- S3/Azure compatible object store support (GCS coming soon!)
- Encryption
- Multiple compression types (gzip, bzip, lz4, zstd)

The pgBackRest project started in 2013 and was originally written in Perl. The migration to pure C was completed in 2019.

Backup Types

- Full

  A complete copy of the database, not dependent on any other backup.
- Incremental

  Copy only files that have changed since the last backup (full, diff, or incr).
- Differential

  Like an incremental, but only copy files that have changed since the last **full** backup.

Incremental is the default since it is generally the least expensive backup type in terms of time and space.

Backup Retention (Count-based)

A good default retention is:

```
repo1-retention-full=5
repo1-retention-diff=3
```

Run the full backup once a week and differential backups the other six days:

```
# m h  dom mon dow  command
45 08  *   *   0    pgbackrest --type=full --stanza=demo backup
45 08  *   *   1-6  pgbackrest --type=diff --stanza=demo backup
```

With this retention there will always be at least five full backups.

If full backups are run weekly then there will be four weeks of retention but if they are run more frequently then retention will be reduced.

Backup Retention (Time-based)

A safer option is to use time-based retention.

Four weeks of full retention with three differentials:

```
repo1-retention-full-type=time
repo1-retention-full=28
repo1-retention-diff=3
```

Run the full backup once a week and differentials the other six days.

With this schedule there will always be at least 4 weeks of backups and accidentally making an extra full backup will not reduce the retention period.

Backup Retention (Ad hoc)

If there are extra, unneeded backups they can be pruned with ad hoc expiration:

```
pgbackrest --stanza=demo --set=20201103_190545F expire
```

Note that dependent backups will also be expired, i.e. differential and incremental backups that depend on the full backup.

Use the `--dry-run` option to see which backups will be expired.

Combining the `info` command and ad hoc expiration it is possible to create a custom retention schedule, though this would require some coding.

Archiving

Asynchronous parallel archiving allows compression and transfer to be offloaded to another process which maintains continuous connections to the remote server, improving throughput significantly.

Make sure there is enough space to hold WAL for an extended period if archiving stops.

It is very important to monitor archiving to ensure it continues working.

Recovery Point Objective

A recovery point objective (RPO) is the maximum acceptable amount of data loss measured in time.

Meeting RPO

First, define the RPO. If the RPO is zero then synchronous replication may be the only viable option. Note that synchronous replication can have major performance impacts, especially in high-latency environments, e.g. multiple data centers.

In certain situations, a zero RPO may not be possible, e.g., a table is dropped from the primary.

For a non-zero RPO there are a few good options:

- Asynchronous replication
  Replication lag should be monitored to ensure it does not fall outside the RPO requirement.
- Archive timeout
  The `archive_timeout` setting ensures that a WAL segment is archived at the interval specified which can be tuned to meet RPO.

These two methods can be combined to minimize RPO within the allowance of the RTO.

Backup from Standby

Backup from standby greatly reduces load on the primary.

pgBackRest uses a hybrid technique that uses both the primary and a standby to make a backup:

- Backup is started on primary.
- Backup starts when replay location on standby reaches start backup location.
- Reduces load on the primary because replicated files are copied from the standby.

Backup to a Safe Location

Do not backup to storage on the PostgreSQL host!

Instead, backup to:

- Storage on a repository host.
- NFS (or other network storage) mounted to the PostgreSQL host.
- An object store (S3, Azure, GCS).

Page Checksums

When initializing a new cluster always specify the `-k` option.

This will enable page checksums which pgBackRest will verify on every backup. If corruption is detected early it can often be corrected.

If corruption is detected, don't panic! As long as older non-corrupted backups exist there are options:

- Restore an older backup and perform recovery
  There is a good chance that corruption in the database is not in the WAL.
- Restore an older backup and retrieve corrupted data
  If corruption exists in only one area then it may be possible to retrieve that data from a backup and then load it into the primary database.

Failed Backups

A failed backup should not be cause for panic. Ideally, there are a number of backups that can be used for a restore.

Remember:

Backup + WAL = Recoverable Database

and:

OLDER Backup + WAL = Recoverable Database

Recovery Time Objective

The Recovery Time Objective (RTO) is the maximum time allowed between an unexpected failure and the resumption of normal operation.

Meeting the Recovery Time Objective

First, define the Recovery Time Objective (RTO).

The easiest way to meet a low RTO is to have a standby server ready and failover from the primary. This can be done manually or with a tool that automates the process, e.g. Patroni.

Monitor the standby to make sure replication lag is within the tolerance of your Recovery Point Objective (RPO).

A standby is not a backup! Even with standby it is important to have recent backups in case there are problems with the standby, or there was data loss that has already been replicated.

Daily backups are usually sufficient, but be sure to measure your restore and recovery time to be sure it meets RTO.

Schrödingers Backup

The state of any backup is unknown until a restore is attempted.

Restore and Recovery

Multi-processing can lead to dramatic reductions in restore time and network utilization.

Use the `--delta` option with the `restore` command to save time.

Double-check where you are restoring. pgBackRest will refuse to overwrite a *running* cluster but it will overwrite a shut down cluster when `--delta` is used.

Restore Testing

Test your restores! Also validate that the database looks correct after the restore.

- Check timestamps in a database to verify RPO was met
- Use tools such as amcheck to verify consistency of the cluster
- Run `pg_dump` and (if possible) restore the cluster from the dump
- Run smoke tests and/or regression tests

Make a Disaster Recovery Plan

Make a disaster recovery plan.

Document the disaster recovery plan.

Practice the disaster recovery plan.

Questions?

website: `http://www.pgbackrest.org`

email: david@pgbackrest.org
email: david@crunchydata.com

releases: `https://github.com/pgbackrest/pgbackrest/releases`

slides & demo: `https://github.com/dwsteele/conference/releases`