PostgreSQL

Enterprise applications

NetApp
April 25, 2024

Table of Contents

PostgreSQL

PostgreSQL databases on ONTAP

PostgreSQL comes with variants that include PostgreSQL, PostgreSQL Plus, and EDB Postgres Advanced Server (EPAS). PostgreSQL is typically deployed as the back-end database for multitier applications. It is supported by common middleware packages (such as PHP, Java, Python, Tcl/Tk, ODBC, and JDBC) and has historically been a popular choice for open-source database management systems. ONTAP is a excellent choice for running PostgreSQL databases do to its reliability, high performing and efficent data management capabilities.

This documentation on ONTAP and the PostgreSQL database replaces the previously published *TR-4770: PostgreSQL database on ONTAP best practices.*

As data grows exponentially, data management becomes more complex for enterprises. This complexity increases licensing, operational, support, and maintenance costs. To reduce the overall TCO, consider switching from commercial to open-source databases with reliable, high-performing back-end storage.

ONTAP is an ideal platform because ONTAP is literally designed for databases. Numerous features such as random IO latency optimizations to advanced quality of service (QoS) to basic FlexClone functionality were created specifically to address the needs of database workloads.

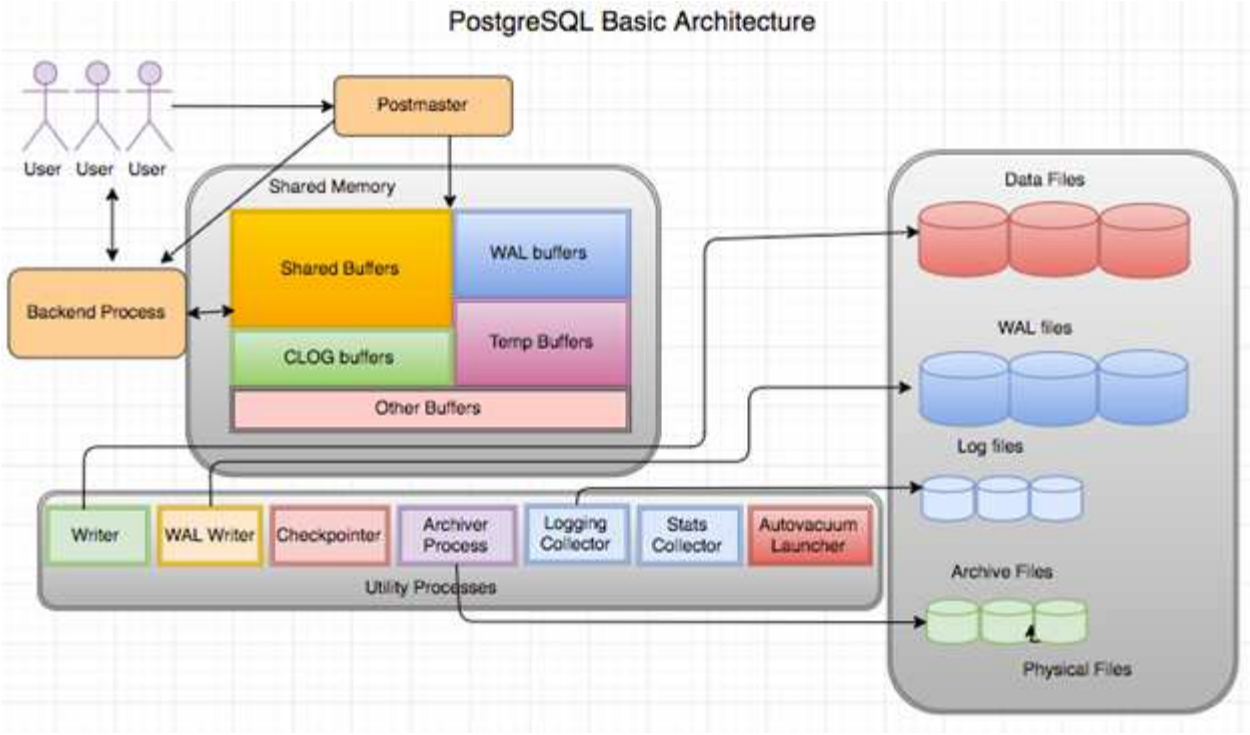
Additional features such as nondisruptive upgrades, (including storage replacement) ensure that your critical databases remain available. You can also have instant disaster recovery for large environments through MetroCluster, or select databases using SnapMirror active sync.

Most importantly, ONTAP delivers unmatched performance with the ability to size the solution for your unique needs. Our high-end systems can deliver over 1M IOPS with latencies measured in microseconds, but if you only need 100K IOPS you can rightsize your storage solution with a smaller controller that still runs the exact same storage operating system.

Database configuration

PostgreSQL architecture

PostgreSQL is an RDBMS based on client and server architecture. A PostgreSQL instance is known as a database cluster, which is a collection of databases as opposed to a collection of servers.



There are three major elements in a PostgreSQL database: the postmaster, the front end (client), and the back end. The client sends requests to the postmaster with information such as IP protocol and which database to connect to. The postmaster authenticates the connection and passes it to the back-end process for further communication. The back-end process executes the query and sends results directly to the front end (client).

A PostgreSQL instance is based on a multiprocess model instead of a multithreaded model. It spawns multiple processes for different jobs, and each process has its own functionality. The major processes include the client process, the WAL writer process, the background writer process, and the checkpointer process:

- When a client (foreground) process sends read or write requests to the PostgreSQL instance, it doesn't read or write data directly to the disk. It first buffers the data in shared buffers and write-ahead logging (WAL) buffers.
- A WAL writer process manipulates the content of the shared buffers and WAL buffers to write into the WAL logs. WAL logs are typically transaction logs of PostgreSQL and are sequentially written. Therefore, to improve the response time from the database, PostgreSQL first writes into the transaction logs and acknowledges the client.
- To put the database in a consistent state, the background writer process checks the shared buffer periodically for dirty pages. It then flushes the data onto the data files that are stored on NetApp volumes or LUNs.
- The checkpointer process also runs periodically (less frequently than the background process) and prevents any modification to the buffers. It signals to the WAL writer process to write and flush the checkpoint record to the end of WAL logs that are stored on the NetApp disk. It also signals the background writer process to write and flush all dirty pages to the disk.

PostgreSQL initialization parameters

You create a new database cluster by using the `initdb` program. An `initdb` script creates the data files, system tables, and template databases (template0 and template1) that define the cluster.

The template database represents a stock database. It contains definitions for system tables, standard views, functions, and data types. `pgdata` acts as an argument to the `initdb` script that specifies the location of the database cluster.

All the database objects in PostgreSQL are internally managed by respective OIDs. Tables and indexes are also managed by individual OIDs. The relationships between database objects and their respective OIDs are stored in appropriate system catalog tables, depending on the type of object. For example, OIDs of databases and heap tables are stored in `pg_database` and `pg_class, respectively. You can determine the OIDs by issuing queries on the PostgreSQL client.

Every database has its own individual tables and index files that are restricted to 1GB. Each table has two associated files, suffixed respectively with `_fsm` and `_vm`. They are referred to as the free space map and the visibility map. These files store the information about free space capacity and have visibility on each page in the table file. Indexes only have individual free space maps and don't have visibility maps.

The `pg_xlog/pg_wal` directory contains the write-ahead logs. Write-ahead logs are used to improve database reliability and performance. Whenever you update a row in a table, PostgreSQL first writes the change to the write-ahead log, and later writes the modifications to the actual data pages to a disk. The `pg_xlog` directory usually contains several files, but initdb creates only the first one. Extra files are added as needed. Each xlog file is 16MB long.

PostgreSQL database configuration with ONTAP

There are several PostgreSQL tuning configurations that can improve performance.

The most commonly used parameters are as follows:

- `max_connections = <num>`: The maximum number of database connections to have at one time. Use this parameter to restrict swapping to disk and killing the performance. Depending on your application requirement, you can also tune this parameter for the connection pool settings.
- `shared_buffers = <num>`: The simplest method for improving the performance of your database server. The default is low for most modern hardware. It is set during deployment to approximately 25% of available RAM on the system. This parameter setting varies depending on how it works with particular database instances; you might have to increase and decrease the values by trial and error. However, setting it high is likely to degrade performance.
- `effective_cache_size = <num>`: This value tells PostgreSQL's optimizer how much memory PostgreSQL has available for caching data and helps in determining whether to use an index. A larger value increases the likelihood of using an index. This parameter should be set to the amount of memory allocated to `shared_buffers` plus the amount of OS cache available. Often this value is more than 50% of the total system memory.
- `work_mem = <num>`: This parameter controls the amount of memory to be used in sort operations and hash tables. If you do heavy sorting in your application, you might need to increase the amount of memory, but be cautious. It isn't a system wide parameter, but a per-operation one. If a complex query has several sort operations in it, it uses multiple work_mem units of memory, and multiple back ends could be doing this simultaneously. This query can often lead your database server to swap if the value is too large. This option was previously called sort_mem in older versions of PostgreSQL.
- `fsync = <boolean> (on or off)`: This parameter determines whether all your WAL pages should be synchronized to disk by using fsync() before a transaction is committed. Turning it off can sometimes improve write performance and turning it on increases protection from the risk of corruption when the system crashes.
- `checkpoint_timeout`: The checkpoint process flushes committed data to disk. This involves a lot of

read/write operations on disk. The value is set in seconds and lower values decrease crash recovery time and increasing values can reduce the load on system resources by reducing the checkpoint calls. Depending on application criticality, usage, database availability, set the value of checkpoint_timeout.

- `commit_delay = <num>` and `commit_siblings = <num>`: These options are used together to help improve performance by writing out multiple transactions that are committing at once. If there are several commit_siblings objects active at the instant your transaction is committing, the server waits for commit_delay microseconds to try to commit multiple transactions at once.
- `max_worker_processes / max_parallel_workers`: Configure the optimal number of workers for processes. Max_parallel_workers correspond to the number of CPUs available. Depending on application design, queries might require a lesser number of workers for parallel operations. It is better to keep the value for both parameters the same but adjust the value after testing.
- `random_page_cost = <num>`: This value controls the way PostgreSQL views non-sequential disk reads. A higher value means PostgreSQL is more likely to use a sequential scan instead of an index scan, indicating that your server has fast disks Modify this setting after evaluating other options like plan-based optimization, vacuuming, indexing to altering queries or schema.
- `effective_io_concurrency = <num>`: This parameter sets the number of concurrent disk I/O operations that PostgreSQL attempts to execute simultaneously. Raising this value increases the number of I/O operations that any individual PostgreSQL session attempts to initiate in parallel. The allowed range is 1 to 1,000, or zero to disable issuance of asynchronous I/O requests. Currently, this setting only affects bitmap heap scans. Solid-state drives (SSDs) and other memory-based storage (NVMe) can often process many concurrent requests, so the best value can be in the hundreds.

See the PostgreSQL documentation for a complete list of PostgreSQL configuration parameters.

TOAST

TOAST stands for The Oversized-Attribute Storage Technique. PostgreSQL uses a fixed page size (commonly 8KB) and does not allow tuples to span multiple pages. Therefore, it is not possible to store large field values directly. When you attempt to store a row that exceeds this size, TOAST breaks up the data of large columns into smaller “pieces” and stores them in a TOAST table.

The large values of toasted attributes are pulled out (if selected at all) only at the time the result set is sent to the client. The table itself is much smaller and can fit more rows into the shared buffer cache than it could without any out-of-line storage (TOAST).

VACUUM

In normal PostgreSQL operation, tuples that are deleted or made obsolete by an update are not physically removed from their table; they remain present until VACUUM is run. Therefore, you must run VACUUM periodically, especially on frequently updated tables. The space it occupies must then be reclaimed for reuse by new rows, to avoid disk space outage. However, it does not return the space to the operating system.

The free space inside a page is not fragmented. VACUUM rewrites the entire block, efficiently packing the remaining rows and leaving a single contiguous block of free space in a page.

In contrast, VACUUM FULL actively compacts tables by writing a completely new version of the table file with no dead space. This action minimizes the size of the table but can take a long time. It also requires extra disk space for the new copy of the table until the operation completes. The goal of routine VACUUM is to avoid VACUUM FULL activity. This process not only keeps tables at their minimum size, but also maintains steady-state usage of disk space.

PostgreSQL tablespaces

Two tablespaces are automatically created when the database cluster is initialized.

The `pg_global` tablespace is used for shared system catalogs. The `pg_default` tablespace is the default tablespace of the template1 and template0 databases. If the partition or volume on which the cluster was initialized runs out of space and cannot be extended, a tablespace can be created on a different partition and used until the system can be reconfigured.

An index that is heavily used can be placed on a fast, highly available disk, like a solid-state device. Also, a table storing archived data that is rarely used or not performance critical can be stored on a less expensive, slower disk system like SAS or SATA drives.

Tablespaces are a part of the database cluster and cannot be treated as an autonomous collection of data files. They depend on metadata contained in the main data directory, and therefore cannot be attached to a different database cluster or backed up individually. Similarly, if you lose a tablespace (through file deletion, disk failure, and so on), the database cluster might become unreadable or unable to start. Placing a tablespace on a temporary file system like a RAM disk risks the reliability of the entire cluster.

After it is created, a tablespace can be used from any database if the requesting user has sufficient privileges. PostgreSQL uses symbolic links to simplify the implementation of tablespaces. PostgreSQL adds a row to the `pg_tablespace` table (a clusterwide table) and assigns a new object identifier (OID) to that row. Finally, the server uses the OID to create a symbolic link between your cluster and the given directory. The directory `$PGDATA/pg_tblspc` contains symbolic links that point to each of the non-built-in tablespaces defined in the cluster.

Storage configuration

PostgreSQL databases with NFS Filesystems

PostgreSQL databases can be hosted on NFSv3 or NFSv4 filesystems. The best option depends on factors outside the database.

For example, NFSv4 locking behavior may be preferable in certain clustered environments. (See here for additional details)

Database functionality should otherwise be close to identical, including performance. The only requirement is the use of the `hard` mount option. This is required to ensure soft timeouts do not produce unrecoverable IO errors.

If NFSv4 is chosen as a protocol, NetApp recommends using NFSv4.1. There are some functional enhancements to the NFSv4 protocol in NFSv4.1 that improve resiliency over NFSv4.0.

Use the following mount options for general database workloads:

```
rw,hard,nointr,bg,vers=[3|4],proto=tcp,rsize=65536,wsize=65536
```

If heavy sequential IO is expected, the NFS transfer sizes can be increased as described in the following section.

NFS Transfer Sizes

By default, ONTAP limits NFS I/O sizes to 64K.

Random I/O with an most applications and databases uses a much smaller block size which is well below the 64K maximum. Large-block I/O is usually parallelized, so the 64K maximum is also not a limitation to obtaining maximum bandwidth.

There are some workloads where the 64K maximum does create a limitation. In particular, single-threaded operations such as backup or recovery operation or a database full table scan run faster and more efficiently if the database can perform fewer but larger I/Os. The optimum I/O handling size for ONTAP is 256K.

The maximum transfer size for a given ONTAP SVM can be changed as follows:

```
Cluster01::> set advanced
Warning: These advanced commands are potentially dangerous; use them only
when directed to do so by NetApp personnel.
Do you want to continue? {y|n}: y
Cluster01::*> nfs server modify -vserver vserver1 -tcp-max-xfer-size
262144
Cluster01::*>
```

Caution
Never decrease the maximum allowable transfer size on ONTAP below the value of rsize/wsize of currently mounted NFS file systems. This can create hangs or even data corruption with some operating systems. For example, if NFS clients are currently set at an rsize/wsize of 65536, then the ONTAP maximum transfer size could be adjusted between 65536 and 1048576 with no effect because the clients themselves are limited. Reducing the maximum transfer size below 65536 can damage availability or data.

Once the transfer size is increased at the ONTAP level, the following mount options would be used:

```
rw,hard,nointr,bg,vers=[3|4],proto=tcp,rsize=262144,wsize=262144
```

NFSv3 TCP Slot Tables

If NFSv3 is used with Linux, it is critical to properly set the TCP slot tables.

TCP slot tables are the NFSv3 equivalent of host bus adapter (HBA) queue depth. These tables control the number of NFS operations that can be outstanding at any one time. The default value is usually 16, which is far too low for optimum performance. The opposite problem occurs on newer Linux kernels, which can automatically increase the TCP slot table limit to a level that saturates the NFS server with requests.

For optimum performance and to prevent performance problems, adjust the kernel parameters that control the TCP slot tables.

Run the `sysctl -a | grep tcp.*.slot_table` command, and observe the following parameters:

```
# sysctl -a | grep tcp.*.slot_table
sunrpc.tcp_max_slot_table_entries = 128
sunrpc.tcp_slot_table_entries = 128
```

All Linux systems should include `sunrpc.tcp_slot_table_entries`, but only some include `sunrpc.tcp_max_slot_table_entries`. They should both be set to 128.

Caution
Failure to set these parameters may have significant effects on performance. In some cases, performance is limited because the linux OS is not issuing sufficient I/O. In other cases, I/O latencies increases as the linux OS attempts to issue more I/O than can be serviced.

PostgreSQL with SAN Filesystems

PostgreSQL databases with SAN are generally hosted on xfs filesystems, but others can be used if supported by the OS vendor

While a single LUN can generally support up to 100K IOPS, IO-intensive databases generally require the use of LVM with striping.

LVM Striping

Before the era of flash drives, striping was used to help overcome the performance limitations of spinning drives. For example, if an OS needs to perform a 1MB read operation, reading that 1MB of data from a single drive would require a lot of drive head seeking and reading as the 1MB is slowly transferred. If that 1MB of data was striped across 8 LUNs, the OS could issue eight 128K read operations in parallel and reduce the time required to complete the 1MB transfer.

Striping with spinning drives was more difficult because the I/O pattern had to be known in advance. If the striping wasn’t correctly tuned for the true I/O patterns, striped configurations could damage performance. With Oracle databases, and especially with all-flash configurations, striping is much easier to configure and has been proven to dramatically improve performance.

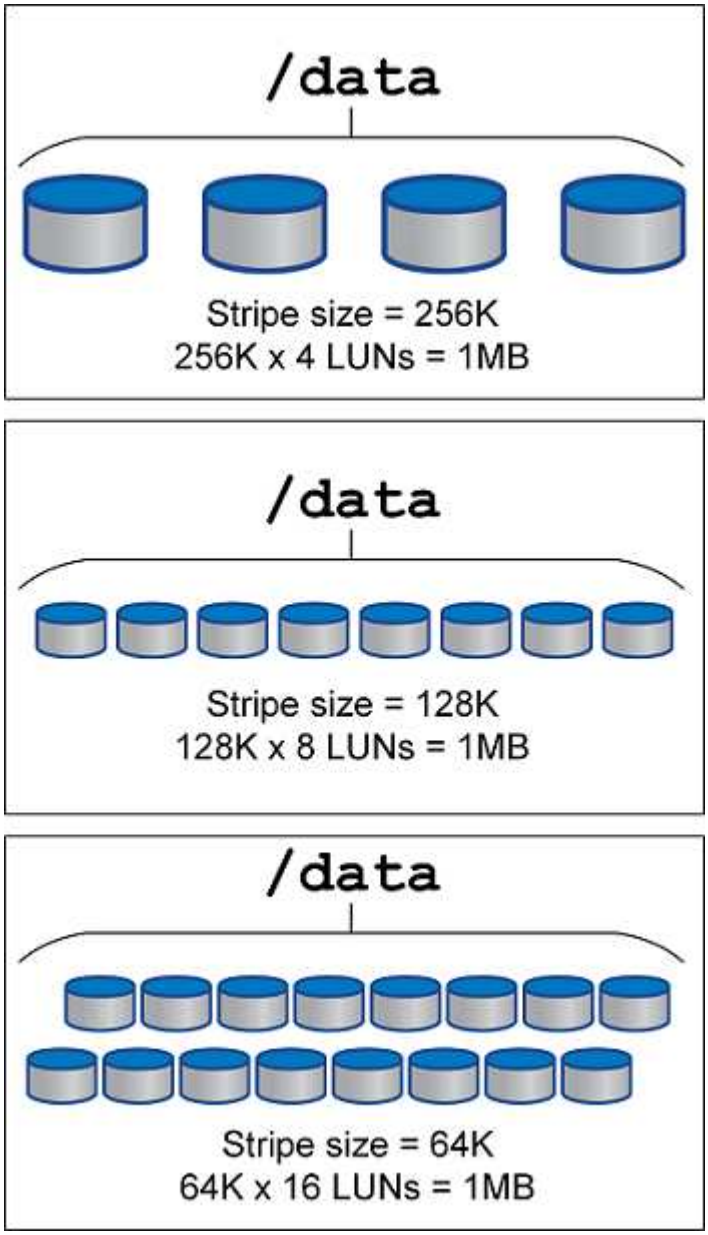
Logical volume managers such as Oracle ASM stripe by default, but native OS LVM do not. Some of them bond multiple LUNs together as a concatenated device, which results in datafiles that exist on one and only one LUN device. This causes hot spots. Other LVM implementations default to distributed extents. This is similar to striping, but it’s coarser. The LUNs in the volume group are sliced into large pieces, called extents and typically measured in many megabytes, and the logical volumes are then distributed across those extents. The result is random I/O against a file should be well distributed across LUNs, but sequential I/O operations are not as efficient as they could be.

Performance-intensive application I/O is nearly always either (a) in units of the basic block size or (b) one megabyte.

The primary goal of a striped configuration is to ensure that single-file I/O can be performed as a single unit, and multiblock I/Os, which should be 1MB in size, can be parallelized evenly across all LUNs in the striped volume. This means that the stripe size must not be smaller than the database block size, and the stripe size multiplied by the number of LUNs should be 1MB.

The following figure shows three possible options for stripe size and width tuning. The number of LUNs is selected to meet performance requirements as described above, but in all cases the total data within a single

stripe is 1MB.



Data protection

PostgreSQL data protection

One of the major aspects of storage design is enabling protection for PostgreSQL volumes. Customers can protect their PostgreSQL databases either by using the dump approach or by using file system backups. This section explains the different approaches of backing up individual databases or the entire cluster.

There are three approaches to backing up PostgreSQL data:

- SQL Server dump
- File-system-level backup

- Continuous archiving

The idea behind the SQL Server dump method is to generate a file with SQL Server commands that, when returned to the server, can re-create the database as it was at the time of the dump. PostgreSQL provides the utility programs `pg_dump` and `pg_dump_all` for creating individual and cluster-level backup. These dumps are logical and do not contain enough information to be used by WAL replay.

An alternative backup strategy is to use file-system-level backup, in which administrators directly copy the files that PostgreSQL uses to store the data in the database. This method is done in offline mode: the database or cluster must be shut down. Another alternative is to use `pg_basebackup` to run hot streaming backup of the PostgreSQL database.

PostgreSQL databases and storage snapshots

Snapshot-based backups with PostgreSQL requires configuration of snapshots for datafiles, WAL files, and archived WAL files to provide full or point-in-time recovery.

For PostgreSQL databases, the average backup time with snapshots is in the range of a few seconds to a few minutes. This backup speed is 60 to 100 times faster than `pg_basebackup` and other file-system-based backup approaches.

Snapshots on NetApp storage can be both crash-consistent and application-consistent. A crash-consistent snapshot is created on storage without quiescing the database, whereas an application-consistent snapshot is created while the database is in backup mode. NetApp also ensures that subsequent snapshots are incremental-forever backups to promote storage savings and network efficiency.

Because snapshots are rapid and do not affect system performance, you can schedule multiple snapshots daily instead of creating a single daily backup as with other streaming backup technology. When a restore and recovery operation is necessary, the system downtime is reduced by two key features:

- NetApp SnapRestore data recovery technology means that the restore operation is executed in seconds.
- Aggressive recovery point objectives (RPOs) mean that fewer database logs must be applied and forward recovery is also accelerated.

For backing up PostgreSQL, you must ensure that the data volumes are protected simultaneously with (consistency-group) WAL and the archived logs. While you are using Snapshot technology to copy WAL files, make sure that you run `pg_stop` to flush all the WAL entries that must be archived. If you flush the WAL entries during the restore, then you only need to stop the database, unmount, or delete the existing data directory and perform a SnapRestore operation on storage. After the restore is done, you can mount the system and bring it back to its current state. For point-in-time recovery, you can also restore WAL and archive logs; then PostgreSQL decides the most consistent point and recovers it automatically.

Consistency groups are a feature in ONTAP and are recommended when there are multiple volumes mounted to a single instance or a database with multiple tablespaces. A consistency group snapshot ensures all volumes are grouped together and protected. A consistency group can be managed efficiently from ONTAP System Manager and you can even clone it to create an instance copy of a database for testing or development purposes.

For more information on Consistency groups, see the NetApp Consistency groups overview.

PostgreSQL data protection software

NetApp SnapCenter plugin for PostgreSQL database, combined with Snapshot and

NetApp FlexClone technologies, offer you benefits such as:

- Rapid backup and restore.
- Space-efficient clones.
- The ability to build a speedy and effective disaster recovery system.

You might prefer to choose NetApp's premium backup partners such as Veeam Software and Commvault under the following circumstances:

- Managing workloads across a heterogenous environment
- Storing backups to either cloud or tape for long-term retention
- Support for a wide range of OS versions and types

SnapCenter plugin for PostgreSQL is community supported plugin and the setup and documentation is available on NetApp Automation store. Through SnapCenter, user can backup database, clone and restore data remotely.

Copyright information

Trademark information