

# POSTGRESQL AND DELL EMC VMAX ALL FLASH

## Performance Assessment and Backup and Recovery Best Practices

July 2017

### VMAX<sup>®</sup> Engineering White Paper

#### ABSTRACT

This white paper provides details on the performance assessment tests and best practices for backup and recovery of PostgreSQL databases with Dell EMC<sup>™</sup> VMAX<sup>™</sup> All Flash storage arrays.

H16429

This document is not intended for audiences in China, Hong Kong, Taiwan, and Macao.

## Copyright

The information in this publication is provided as is. Dell Inc. makes no representations or warranties of any kind with respect to the information in this publication, and specifically disclaims implied warranties of merchantability or fitness for a particular purpose.

Use, copying, and distribution of any software described in this publication requires an applicable software license.

Copyright 2017 Dell Inc. or its subsidiaries. All Rights Reserved. Dell, EMC, Dell EMC and other trademarks are trademarks of Dell Inc. or its subsidiaries. Intel, the Intel logo, the Intel Inside logo, and Xeon are trademarks of Intel Corporation in the U.S. and/or other countries. Other trademarks may be the property of their respective owners.

Published in the USA July 2017 White Paper H16429.

Dell Inc. believes the information in this document is accurate as of its publication date. The information is subject to change without notice.



# Contents

<b>Executive summary .....</b>	<b>4</b>
<b>VMAX All Flash storage array product overview .....</b>	<b>5</b>
<b>PostgreSQL product overview .....</b>	<b>6</b>
<b>Benefits of running PostgreSQL with VMAX All Flash .....</b>	<b>8</b>
<b>Performance assessment tests .....</b>	<b>10</b>
<b>Using VMAX snapshots as PostgreSQL gold copies .....</b>	<b>14</b>
<b>Using VMAX snapshots for PostgreSQL online backups .....</b>	<b>20</b>
<b>Summary .....</b>	<b>28</b>
<b>References .....</b>	<b>29</b>
<b>Appendix – Linux Logical Volume Manager .....</b>	<b>29</b>

## Executive summary

An increasing number of companies are developing new in-house applications using open source databases. Many are also converting existing brand-name relational database management system (RDBMS) applications to open-source databases. With the increased maturity and popularity of Linux, PostgreSQL on Linux x86-64 is a solution that provides an industry-compliant SQL database. Another benefit of using an x86-64 platform is the ability to build PostgreSQL servers in virtual environments, such as VMware. Using a virtual environment enables additional savings in power consumption, increased deployment agility, and reduced footprint.

PostgreSQL is an open source RDBMS with SQL, stored-procedures, and Object Relational Model (ORM) support. PostgreSQL is not controlled by any corporation or other private entity, and the source code is available free of charge. PostgreSQL features include compression, partitioning, full-text indexing, replications, and native JSON support. PostgreSQL is fully ACID compliant, meaning that data is persistent; it can restart after a crash without data loss, and can use Dell EMC™ VMAX™ storage data services.

The VMAX All Flash family of storage arrays is designed and optimized for high performance while providing ease-of-use, reliability, availability, security, and a robust set of data services. VMAX All Flash delivers unparalleled performance as a mission-critical multi-controller platform. VMAX management is easy using Dell EMC Unisphere™, command-line interface (CLI), or REST APIs. The data is protected with T10-DIF (data integrity field), and can be encrypted with D@RE<sup>1</sup>. With Dell EMC SnapVX™, local snapshots can be created or restored in seconds (regardless of data capacity), and Dell EMC SRDF™ provides consistent remote replications to any distance.

The combination of open source databases with VMAX All Flash storage provides the benefits of high performance, availability, reliability, and data protection to the critical space of hybrid cloud applications and Dev/Ops.

Although not covered in this white paper, some companies provide their own PostgreSQL variants to make it more enterprise-friendly. One such company is EnterpriseDB, with features such as performance enhancements and high Oracle RDBMS compatibility. Another is Pivotal, where Dingo PostgreSQL is used for Cloud Foundry SQL and NoSQL/JSON access.

This white paper describes the performance benefits of PostgreSQL deployed on VMware ESXi servers with VMAX All Flash. The paper also shows you how to create database copies with SnapVX snapshots and how to perform online backups in seconds, regardless of database size.

## Audience

This white paper is intended for database administrators, system administrators, storage administrators, and system architects who are responsible for implementing PostgreSQL with VMAX All Flash storage systems. Readers should have some familiarity with PostgreSQL and VMAX storage arrays.

---

<sup>1</sup> D@RE refers to Data at Rest Encryption, an optional feature of VMAX All Flash.

## VMAX All Flash storage array product overview

### VMAX All Flash family

The VMAX family of storage arrays is built on the strategy of simple, intelligent, modular storage. It incorporates a Dynamic Virtual Matrix interface that connects and shares resources across all VMAX engines, enabling the storage array to seamlessly grow from an entry-level configuration into the world's largest storage array. VMAX storage provides the highest levels of performance, scalability, and availability, and features advanced hardware and software capabilities.

In 2016, Dell EMC announced the VMAX All Flash 250F, 450F, and 850F arrays. In May 2017, Dell EMC introduced VMAX 950F, which replaces the VMAX 450F and 850F, and provides higher performance at a similar cost.

VMAX All Flash arrays, as shown in Figure 1, provide a combination of ease of use, scalability, high performance, and a robust set of data services that makes them an ideal choice for database deployments.



Figure 1. VMAX All Flash 950F (left) and 250F (right) storage arrays

### VMAX All Flash benefits

VMAX All Flash storage arrays provide the following benefits:

- **Ease of use**—Uses virtual provisioning to create new storage devices in seconds. All VMAX devices are thin, consuming only the storage capacity that is actually written to, which increases storage efficiency without compromising performance. VMAX devices are grouped into storage groups and managed as a unit for operations such as: device masking to hosts; performance monitoring; local and remote replications; compression; and host I/O limits. In addition, you can manage VMAX devices by using Unisphere for VMAX, Solutions Enabler CLI, or REST APIs.
- **High performance**—Designed for high performance and low latency. VMAX arrays scale from one up to eight engines (Dell EMC VMAX V-Bricks). Each engine consists of dual directors, where each director includes two-socket Intel CPUs, front-end and back-end connectivity, hardware compression module, InfiniBand internal fabric, and a large mirrored and persistent cache.

All writes are acknowledged to the host as soon as they are registered with VMAX cache<sup>2</sup>. Writes are later written to flash after multiple updates. Reads also benefit from the VMAX large cache. When a read is requested for data that is not already in cache, FlashBoost technology delivers the I/O directly from the back-end (flash) to the front-end (host). Reads are only later staged in the cache for possible future access. VMAX also excels in servicing high bandwidth sequential workloads that leverage pre-fetch algorithms, optimized writes, and fast front-end and back-end interfaces.

- **Data services**—Offers a strong set of data services. It natively protects all data with T10-DIF from the moment data enters the array until it leaves (including replications). With SnapVX and SRDF, VMAX provides many topologies for consistent local and remote replications. Dell EMC ProtectPoint™ provides an integration with Data Domain™, and Dell EMC CloudArray™ provides cloud gateways. Other VMAX data services include D@RE, Quality of Service (QoS)<sup>3</sup>, compression, the “call home” support feature, non-disruptive upgrades (NDU), non-disruptive migrations (NDM), and more. In virtual environments, VMAX also supports VMware vStorage APIs for Array Integration (VAAI) primitives such as write-same and xcopy.

---

**Note:** While outside the scope of this white paper, you can also purchase VMAX as part of a Converged Infrastructure (CI). For details, refer to [Dell EMC VxBlock System 740 and VBlock System 740](#).

---

## PostgreSQL product overview

### PostgreSQL overview

PostgreSQL is an enterprise-class open-source object-relational database system that has been in active development for over 15 years. It runs on all major operating systems, including Linux and Windows. Its SQL implementation strongly conforms to the ANSI-SQL:2008 standard.

PostgreSQL has full support for foreign keys, joins, views, triggers, and stored procedures. It supports binary large objects (for example, images, sounds, and video). Other features include Multi-Version Concurrency Control (MVCC), meaning each query shows consistent data at the time the query started, hot backups (which allows you to create backups while the database is open and running transactions), point-in-time recovery, asynchronous replications, and more.

PostgreSQL is ACID compliant, which means that it has the following properties:

- **Atomicity**—A transaction succeeds or fails as a unit.
- **Consistency**—A transaction adheres to all its constraint rules (or fail) as a unit.
- **Isolation**—Data of uncommitted transactions is not visible to other sessions.

---

<sup>2</sup> VMAX All Flash cache is large (from 512 GB to 16 TB, based on configuration), mirrored, and persistent due to the vault module that protects the cache content in case of power failure and restores the cache when the system comes back up.

<sup>3</sup> Two separate features support VMAX QoS. The first relates to Host I/O limits that enable placing IOPS or bandwidth limits on “noisy neighbor” applications (set of devices) such as test/dev environments. The second relates to slowing down the copy rate for local or remote replications.

- **Durability**—Committed data is protected through system failure.

**Note:** Due to its ACID qualities, PostgreSQL can leverage VMAX data services such as SnapVX or SRDF to perform local and remote data replications and online backups.

For more information about PostgreSQL features, refer to [PostgreSQL 9.6.3 Documentation](#).

## PostgreSQL processes

From a high-level perspective, a PostgreSQL database cluster<sup>4</sup> requires a set of resources to operate. These include network IP address and port numbers for connectivity, memory, file-system storage space to store the data, and a set of processes that perform the database functions.

The database cluster starts by bringing up a `postgres` process (previously called `postmaster`). This process is the main PostgreSQL database server. With each user connection an additional `postgres` process is started.

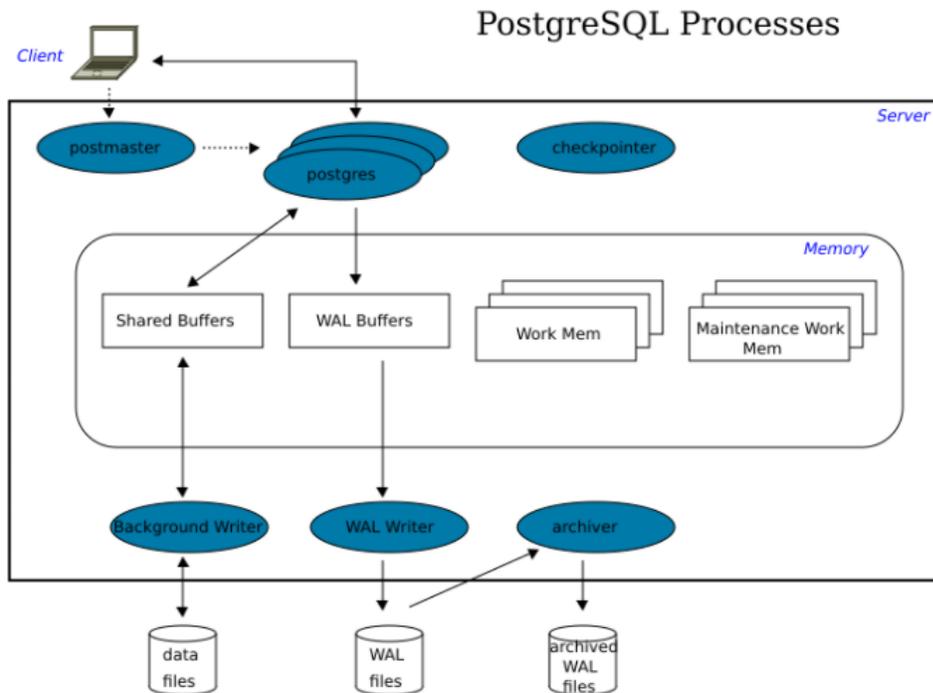


Figure 2. PostgreSQL processes (source: [wikibooks](#))

Other key PostgreSQL processes include:

- **Logger**—Collects information about database activity and health. For details, refer to [Error Reporting and Logging](#).
- **Checkpointer**—Initiates database checkpoints (flush of new or modified database buffers to disk). Checkpoints occur at regular intervals, based on the parameter

<sup>4</sup> The PostgreSQL database cluster can contain multiple distinct databases, where “postgres” is the default database and “template1” is a template database for creating additional databases. They all share the same resources.

`checkpoint_timeout` (5-minute default) or if the parameter `max_wal_size` is about to be exceeded, whichever comes first. For details, refer to [Checkpoints](#).

- **Writer**—To reduce the overhead and write burstiness of checkpoints, the background writer can flush dirty buffers to disk more often, though if used too aggressively, this can cause performance overhead. For details, refer to [Background Writer](#).
- **WAL writer**—Similar to many transactional databases, Write-Ahead Logging (WAL) ensures that before changes are made to the data, the change is first logged in WAL. If a failure occurs while the change is executed, it can be replayed from WAL after the system is operational again. For details, refer to [Write-Ahead Logging \(WAL\)](#).

---

**Note:** A best practice for PostgreSQL with VMAX All Flash is to use a symbolic link to separate the WAL from the data using a different file system and set of storage devices.

---

WAL has multiple [settings](#) that affect its behavior though the defaults ensure the appropriate behavior of the logs for crash protection.

- **Autovacuum launcher**—This process allows PostgreSQL to run `vacuum` and `analyze` commands automatically based on database activity. The launcher process is responsible for starting autovacuum workers as needed. For details, refer to [The Autovacuum Daemon](#), [Vacuum](#), and [Analyze](#).
- **Stats collector**—This process is responsible to collect database statistics. For details, refer to [The Statistics Collector](#).

### PostgreSQL file systems

The location of the datafiles is often indicated by the environment variable `PGDATA` and points to a file-system location where PostgreSQL stores the data files. A subdirectory `./pg_xlog/` points to the write-ahead log files location.

In a VMAX All Flash deployment, mount the data file system on multiple VMAX devices and stripe them with the host logical volume manager (LVM). The WAL subdirectory location (`./pg_xlog/`) should use a symbolic link to another mounted file system, which is also striped across a few VMAX devices. This configuration ensures best performance and availability.

---

**Note:** If the LVM uses the full LUN size of each VMAX device, then the file system is aligned natively. However, if a single partition is created on each LUN, ensure that the LUN is aligned at 1 MB offset (2,048 blocks). With Red Hat Enterprise Linux 7.x, 1 MB offset is the default alignment. In earlier Linux releases, `parted` or `fdisk` commands can be used.

---

While you can separate other file-system locations using symbolic links to new mount-points (for example, temp file-system if the temp files are highly utilized), the separation of data and log is often sufficient.

For more information on PostgreSQL directory structure, refer to [Database File Layout](#).

## Benefits of running PostgreSQL with VMAX All Flash

The combination of PostgreSQL with VMAX All Flash offers a unique set of benefits relative to other deployments, such as public cloud or direct-attached storage (DAS). The benefits are described in the following sections.

## Performance, scale, and ease of use

VMAX All Flash is designed for high scale and performance. All host writes to VMAX are received by its large DRAM-based cache. This allows database updates and WAL writes to execute incredibly fast. In addition, all data is striped across the VMAX back-end storage to provide the best resource utilization, performance, and availability. This design avoids hot spots, such as those that can occur with DAS deployments, where non-uniform access to server storage can create bottlenecks.

VMAX All Flash uses thin provisioning exclusively, such as only data that is written to, is allocated in the array. For example, even if the application is provisioned with a 1TB storage device, only actual writes to that storage allocates capacity in the array, preventing waste and increasing efficiency.

All storage operations are based on storage groups (SGs), which are groups of VMAX devices. SGs are used for device masking to host, performance monitoring, local and remote replication, enable/disable compression, and so on. For example, we recommend one SG that contains all the PostgreSQL data devices and another SG for the WAL devices. Each set of devices is striped by the host LVM and mounted for data and WAL use respectively. Later, you can use these SGs to monitor the performance of the database data or log separately, enable storage compression on only the data, or create a backup/recovery solution. SGs make the management of database performance and replications easy and fast.

Management of the storage environment is also easy, using Unisphere for VMAX, Solutions Enabler Command Line Interface (CLI), or REST APIs. In this way, PostgreSQL performance can be easily viewed using database tools, or from Unisphere for VMAX, allowing fast root-cause analysis in an event of a failure or performance bottleneck.

## Fast creation of database snapshots

VMAX SnapVX can create “gold” copies of production environments, and create and refresh test, development, or reporting environments. VMAX snapshots are created in seconds, regardless of database size, and do not consume any host resources.

VMAX snapshots do not consume additional storage capacity (except for data changes after the snapshot is taken), and can be linked to up to 1,024 targets. You can quickly and easily create and refresh copies of the primary database as often as required, and introduce those copies to other environments, such as test and development.

## Backup and recovery

Mission-critical databases require protection and therefore a backup and recovery strategy. PostgreSQL allows the creation of hot backups (database backups) that are taken while the database is open, executing client transactions. However, a host-based backup in PostgreSQL requires you to copy the data using host commands. This method not only takes a long time (based on the size of the database) but also consumes host resources such as CPU, I/O, and memory.

By using VMAX SnapVX snapshots, in combination with PostgreSQL hot-backup command, you no longer need to use the copy operation on the host. Instead, you can take single or multiple storage snapshots of the data in seconds—regardless of the database size. Unlike host-based copies, where each copy consumes the same capacity as the source database, SnapVX snapshots are extremely efficient. While each snapshot has full access to its point-in-time data, snapshots only consume capacity based on the amount of modified data since the snapshot’s creation. Moreover, when multiple snapshots are

## Performance assessment tests

taken at different times, they share non-unique data, creating additional capacity efficiency.

Database recovery operations also benefit from SnapVX. If a recovery is required, you can restore any of the backup-associated snapshots. The restored data is available in seconds. Even if background copy of the changes is taking place, the database recovery operations can start immediately.

## Improved geographical resiliency

PostgreSQL uses a variety of ways to maintain geographic resiliency, including log-shipping standby server, file-system replication solution, and more.

SRDF provides an optional alternative solution with consistent synchronous and asynchronous replications, from as little as a few devices to spanning multiple arrays. Storage-based replications simplify management (compared to host-based replications that run separately from each server), provide a consistent storage state across all the servers, multiple databases, and applications, and use either Fibre Channel (FC) or IP-based networks.

## Performance assessment tests

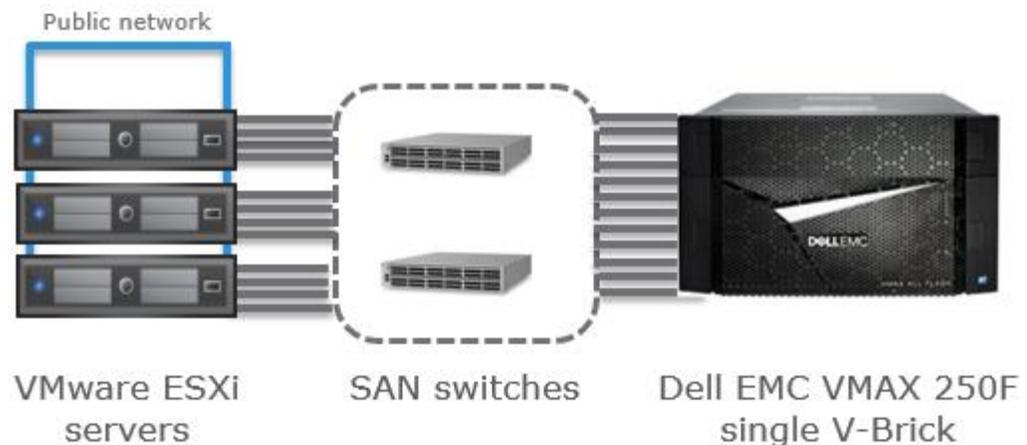
The following sections describe the test configuration that we used to assess the performance of PostgreSQL on Linux, VMware, Cisco UCS servers, and VMAX All-Flash storage.

## Test configuration

### Physical configuration

Figure 3 shows the physical test environment. It consists of a VMAX 250F single V-Brick, three Cisco UCS servers, network, and SAN switches.

Each of the servers uses two dual-port host bus adapters (HBAs) for a total of four initiators per host connected to the SAN switches.



**Figure 3. Physical configuration**

Table 1 describes the hardware and software components of the test configuration.

**Table 1. Hardware and software components**

Device	Quantity	Configuration	Description
Dell EMC VMAX All Flash 250F storage array	1	<ul style="list-style-type: none"> <li>1 x V-Brick (32 solid-state drives (SSDs) in RAID 5)</li> <li>HYPERMAX OS 5977.1125</li> </ul>	Single V-Brick (engine)
VMware ESXi servers	3	<ul style="list-style-type: none"> <li>Cisco UCS-C240</li> <li>10 core x 2 Intel Xeon</li> <li>E5-2680 v2 @ 2.80 GHz</li> <li>96 GB Memory</li> <li>10 GbE network ports (bonded)</li> <li>2 dual port 16 Gb HBAs</li> <li>VMware ESXi 6.0.0</li> <li>RHEL 7.2</li> <li>PostgreSQL 9.6</li> <li>XFS Filesystem</li> <li>pgbench database I/O driver</li> </ul>	Each server has a single virtual machine (VM) with PostgreSQL database configured with pgbench benchmark tool.

### Storage setup

For the tests, we used a VMAX 250F All Flash with a single V-Brick (engine) and 32 SSDs in a RAID 5 (3+1) configuration. A VMAX 250F engine contains two directors, where each can include up to 8 front-end ports, for a total of 16 ports per engine (8 Gb or 16 Gb FC). In our test configuration, we only used 8 front-end ports in total.

VMAX All Flash stripes all data across the back end. As a result, we only focused on the number and size of devices for the hosts. Because we were using host LVM with striping, we created for each host (VM) a set of eight 200 GB thin devices for datafiles, and five 10 GB thin devices for log files. Together with LVM striping and Dell EMC PowerPath™/VE multipathing, this setup allowed for high performance while spreading I/Os across multiple devices and front-end ports.

### VMware ESXi server setup

A set of thin devices for data and log were mapped to each of the three ESXi servers. PowerPath/VE was used for ESXi multipathing. A single VM was created on each of the three ESXi servers to allow for the best resource utilization by the VMs.

---

**Note:** In a real customer configuration, there are likely to be many VMs on each ESXi server, therefore, the choice of server type and size should be decided accordingly.

---

We configured each VM with RHEL 7.2. We added the VMAX devices to each VM as physical Raw Device Mapping (pRDM). By using RDMs, a 1:1 relationship existed between the host and the storage devices. This allowed us to use the storage snapshots at a finer granularity than if we used a VMware datastore.

### VM setup

We set up PostgreSQL 9.6 software and users on each of the VMs.

To set up the file system locations for PostgreSQL data and WAL, we used a combination of host LVM and XFS Filesystem. We created two striped LVMs, one based on eight 200 GB devices for data, and the other based on five 10 GB devices for WAL. We striped both volumes at a 128 KB stripe depth.

We created an XFS Filesystem on each volume and then created two mount points on each host: /mnt/pg\_data and /mnt/pg\_log. We mounted the volumes using XFS default mount options, with the addition of nobarrier and noatime.

We installed the PostgreSQL software on the local disk in directory /opt/PostgreSQL/9.6/ and created the data directory for the database in /mnt/pg\_data/data (subdirectory of the previously created data mount point).

Because the WAL is a subdirectory in the data file system, prior to initializing the PostgreSQL database, we created a symbolic link from /mnt/pg\_data/data/pg\_xlog and pointed it to /mnt/pg\_log/. In this way, when we initialized the database, the WAL was actually created on the /mnt/pg\_log/ mount point.

See Appendix – Linux Logical Volume Manager for more information on how to use RedHat Linux LVM.

### Performance tests overview

For the benchmark, we used [pgbench](#) software. We used two types of workload: one with read/write activity (OLTP-like), where read-ratio was 60 percent; the other with read-only activity.

Each test case was run three times, first with one server running pgbench, then with two servers running pgbench, and finally with three servers. The TPS (transactions per second) were collected from pgbench output, and storage performance metrics were collected from Unisphere for VMAX.

### Mixed-workload test results

Table 2 shows the results for the mixed-workload (OLTP-like) test case.

**Table 2. Mixed-workload test results**

Test parameters				Unisphere						pgbench
Servers	Job	Clients	Runtime (min)	IOPS	Reads/s	Writes/s	MB/s	Read response time (ms)	Write response time (ms)	TPS
One	Mixed	100	30	71,000	41,000	28,000	770	0.2	0.2	11,757
Two	Mixed	100	30	122,000	81,000	52,000	1,350	0.3	0.4	22,441
Three	Mixed	100	30	190,000	108,000	76,000	2,050	0.4	0.8	32,585

Figure 4 shows the matching chart for the I/Os per second (IOPS) and Transactions Per Second (TPS) for the mixed workload.

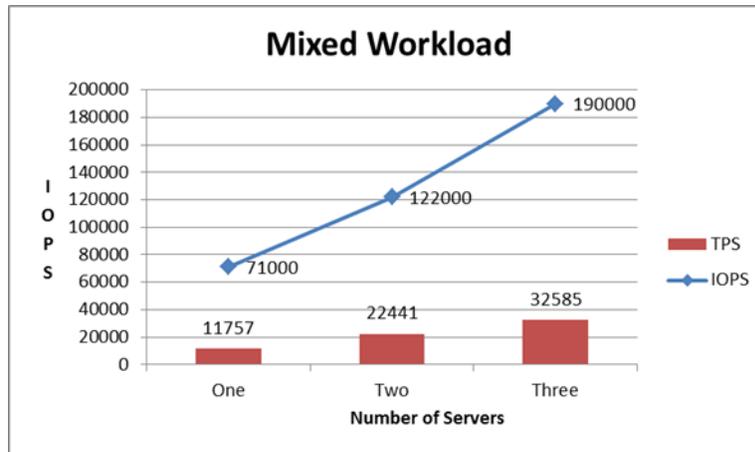


Figure 4. Mixed-workload test results

As the previous table and figure show, not only did we achieve near-linear scalability by adding servers, but read and write latencies were under 1 ms, even as the workload increased. With three servers operating together, the VMAX storage reported 190,000 IOPS at a 0.4 ms read response time, and a 0.8 ms write response time.

### Read-only workload test results

Table 3 shows the results for the read-only workload test case.

Table 3. Read-only workload test results

Test parameters				Unisphere						pgbench
Servers	Job	Clients	Runtime (min)	IOPS	Reads/s	Writes/s	MB/s	Read response time (ms)	Write response time (ms)	TPS
One	Read-only	100	30	128,000	125,000	500	1,150	0.1	0.2	64,269
Two	Read-only	100	30	250,000	240,000	1,000	2,200	0.2	0.3	120,701
Three	Read-only	100	30	315,000	310,000	2,000	2,820	0.6	0.6	150,941

Figure 5 shows the matching chart for the IOPS and TPS for the read-only workload.

As the previous table and figure show, not only did we achieve near-linear scalability by adding servers, but read and write latencies were under 1 ms, even as the workload increased. With three servers operating together, the VMAX storage reported 315,000 IOPS at a 0.6 ms read response time, and a 0.6 ms write response time.

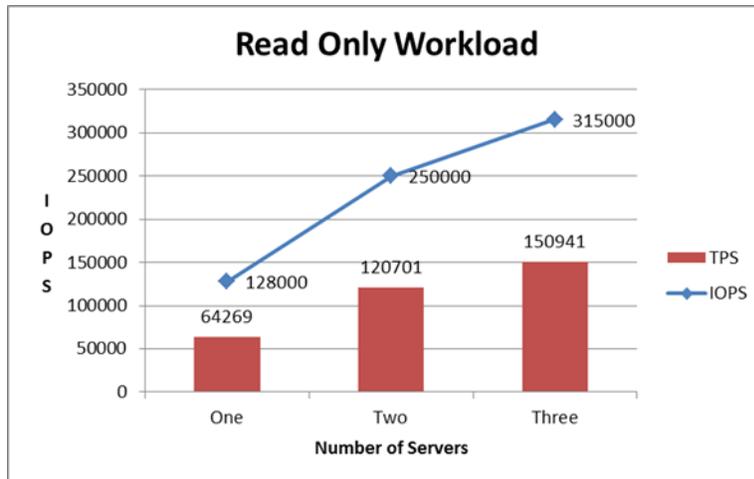


Figure 5. Read-only workload test results

### Performance tests summary

The tests and guidelines provided in this white paper demonstrate the ability of PostgreSQL together with VMAX All Flash to provide an environment that can scale easily while maintaining high performance and low latencies.

## Using VMAX snapshots as PostgreSQL gold copies

### Benefits of using storage snapshots with PostgreSQL

This section describes how you can create and restore storage snapshots of the PostgreSQL database using SnapVX while the database is open.

SnapVX snapshots are always consistent, which means that the database interprets the images they create as if a power failure or database crash has occurred. The database can simply restart using the WAL to recover any of the committed transactions up to the time of the snapshot.

This type of replica does not use archive logs or database recovery to start. Its purpose is the creation of a point in time copy of the database.

---

**Note:** For the database to consider the snapshot content as a “restartable” image, the snapshot must include all the data files and WAL devices and must be executed in a single operation.

---

Some of the key benefits of SnapVX restartable replicas for PostgreSQL include:

- You can create gold copies as a “save point” prior to an operation, such as data load, patch updates, upgrades, or other changes to the PostgreSQL database that may need to be undone if something goes wrong.
- You do not need to stop or quiesce the database operations while you create the snapshot.
- You can use the snapshot to restore the original source devices.
- Alternatively, you can make the snapshots visible to another set of devices, allowing easy and quick creation of database copies for purposes such as test, development, and reporting environments that can be refreshed periodically from production.

## Creating PostgreSQL restartable replicas

### Use case overview

The following use case shows an example of the process you can use to create a point-in-time snapshot of an active PostgreSQL database.

You can manage SnapVX using the Solutions Enabler CLI, Unisphere UI, or REST API. In the following use case, we used Solutions Enabler CLI.

We executed the PostgreSQL commands from the database host. We executed Solutions Enabler CLI commands from a storage-management host, which is often not the same host as the database. If you use Unisphere, it can be installed on the storage-management host or VMAX-embedded Unisphere can be used directly.

To simulate a running system, we used a pgbench workload and kept it running.

To demonstrate the point-in-time concept, we created a test table in the database and inserted known transactions at different stages to show which of them we got back when we restored the database.

### Creating SnapVX snapshots

To create a snapshot:

1. Start pgbench workload to simulate an active PostgreSQL system.
2. Create a test table and insert the first known record.

```
/mnt/pg_data/data/pgbench=# create table testTbl (RecId integer not null,
RedDesc varchar(50));
/mnt/pg_data/data/pgbench=# insert into testTbl values (1, 'Initial
data');
INSERT 0 1
/mnt/pg_data/data/pgbench=# select * from testTbl;
recid | recdesc
-----+-----
      1 | Initial data
(1 row)
```

3. Create the SnapVX snapshot.

The first time a snapshot is taken, the user provides a snapshot name. Additional snapshots can use other distinctive names, or use the same name, and by that create new generations under a single snapshot name. Each snapshot generation can list its unique date and time attributes for ease of management.

---

**Note:** Use the parent SG containing both DATA and WAL devices when creating a restartable snapshot.

---

```
# symsnapvx -sg PG_VM89_DATABASE -name PG_VM89_Snap -v establish
```

4. List the snapshots for the SG.

```
# symsnapvx -sg PG_VM89_DATABASE -snapshot_name PG_VM89_Snap list -detail
-gb
```

5. Add another record to the test table.

Do not expect to find this record after the snapshot is restored, because the record was created *after* the snapshot was taken.

```

/mnt/pg_data/data/pgbench=# insert into testTbl values (2, 'After
snapshot taken');
INSERT 0 1
/mnt/pg_data/data/pgbench=# select * from testTbl;

recid |          recdesc
-----+-----
      1 | Initial data
      2 | After snapshot taken
(2 rows)

```

## Restoring PostgreSQL using restartable replicas

### Use case overview

The following use case shows an example of the process you can use with any of the snapshots previously created to restore the PostgreSQL database to the point-in-time of the snapshot creation.

#### Restoring a PostgreSQL database using a snapshot

To restore a PostgreSQL database:

1. Simulate a disaster by deleting data files from the data directory `$PGDATA`.
2. As user `postgres`, stop the database and check for any remaining processes.

In most disasters, the database crashes, however not always. Prior to restoring the snapshot, ensure that the database is stopped. Failure to do so may result in database processes keeping locks on the file system.

```

$ pg_ctl -D /mnt/pg_data/data stop
$ ps -ef | grep postgres

```

3. Prior to restoring from the snapshot, the affected filesystems need to be unmounted and LVM volumes deactivated.

As user `root`, unmount the PostgreSQL DATA and WAL mount points, and deactivate the volumes.

```

# umount /mnt/pg_data
# umount /mnt/pg_log

# lvchange -a n /dev/pg_data_vg/pg_data_lv
# lvchange -a n /dev/pg_log_vg/pg_log_lv

```

4. Restore the database using the appropriate snapshot. If multiple generations exist, choose the relevant one. By default, the latest snapshot is restored (generation 0 is always the latest).

```

# symsnapvx -sg PG_VM89_DATABASE -snapshot_name PG_VM89_Snap restore

```

---

**Note:** You do not need to wait for the snapshot restore background copy to complete before proceeding with the database restart/recovery operations. When the SnapVX restore operation starts, the data is available for use immediately, even while the background copy runs. If data that is not copied yet is requested, its copy is prioritized. This feature of SnapVX allows database operations to proceed much faster, although performance may be affected until the copy is done. We recommend that you wait for the background copy to complete before allowing full user access.

---

5. Make the DATA and WAL volumes available. Display the volumes to ensure they are online, and then remount the file systems.

```
# lvchange -a y /dev/pg_data_vg/pg_data_lv
# lvchange -a y /dev/pg_log_vg/pg_log_lv
# lvsdisplay
# mount -t xfs -o noatime,nobarrier /dev/pg_data_vg/pg_data_lv /mnt/pg_data
# mount -t xfs -o noatime,nobarrier /dev/pg_log_vg/pg_log_lv /mnt/pg_log
```

6. As user postgres, start the database. It will perform an automatic crash recovery for a short time.

```
$ pg_ctl -D /mnt/pg_data/data -l logfile start
pg_ctl: another server might be running; trying to start server anyway
server starting
```

7. After the database has started, verify that the test table contains the first record, which was inserted prior to the snapshot creation.

```
/mnt/pg_data/data/pgbench=# select * from testTbl;
recid | recdesc
-----+-----
      1 | Initial data
(1 row)
```

This step confirms that the snapshot was restored and that committed transactions up to the point of the snapshot are available.

## Creating a PostgreSQL duplicate database

### Use case overview

The following use case shows an example of the process you can use with any of the snapshots previously created to instantiate a new image of the PostgreSQL database on another host (or VM). This procedure also can be used to easily create additional test, development, or reporting instances. The production database (the source of the snapshot) remains active and is not affected.

For another host to use a snapshot, you must create a new set of devices matching in size to the source, and add them to a target SG. You then run a snapshot link operation, which makes the snapshot data available to the target SG.

---

**Note:** by default, SnapVX link operation does not consume additional storage capacity and is merely a sharing of pointers to the same data. Only when changes are made to the data on the source or target SGs, additional storage capacity is consumed. Some customers prefer to create a new *full copy* of the original data on the target SG during the link. This is achieved by using a `'-copy'` link option. It is not recommended or deemed necessary with VMAX All Flash to do so.

---

### Creating a target SG

A prerequisite for the SnapVX link operation is creating the target SG. Once the target SG is created, you can link and relink (refresh) any of the source SG snapshots to it.

You can create the target SG using Solutions Enabler CLI. Alternatively, you can use Unisphere or REST APIs.

1. As user root, from the storage management host, create devices matching in size to the source devices. Write down the device IDs; in the next step, you will add them to the target SGs.

```
# symdev create -v -tdev -cap 10 -captype gb -N 5 # WAL
# symdev create -v -tdev -cap 200 -captype gb -N 8 # DATA
```

2. Create the target SGs for DATA and WAL, and add the newly created device IDs to them. Then create a parent SG for the target SGs.

```
# symmsg create PG_VM89_WAL_TGT
# symmsg create PG_VM89_DATA_TGT
# symmsg -sg PG_VM89_WAL_TGT addall -devs 57:5B
# symmsg -sg PG_VM89_DATA_TGT addall -devs 5C:63

# symmsg create PG_VM89_DATABASE_TGT # Parent target SG
# symmsg -sg PG_VM89_DATABASE_TGT add sg PG_VM89_WAL_TGT, PG_VM89_DATA_TGT
# symmsg show PG_VM89_DATABASE_TGT
```

3. Create a VMAX masking view.

The details of this step depend on the type of deployment used. In order for the target devices to be visible to a new host or VM, a VMAX masking view is required. A masking view contains host initiators (HBA port WWNs), VMAX front-end ports, and the SGs containing the devices, which are made visible to the host using these ports and initiators. With a VM, the initiators are those of the ESXi host, while vSphere later maps these devices to the VM.

4. After you create the masking view, reboot or rescan the SCSI bus of the target host (or VM) and ensure the target devices are visible.

### Duplicating and refreshing a database

To duplicate and refresh a PostgreSQL database:

1. On the source database (production), start pgbench workload to simulate an active PostgreSQL system. This workload can remain working because production is undisturbed by the creation of a duplicate database.
2. On the target host (not production), ensure the database is stopped. Failure to do so may result in database processes keeping locks on the filesystem.

As user postgres, stop the database and check for any remaining processes.

```
$ pg_ctl -D /mnt/pg_data/data stop
$ ps -ef | grep postgres
```

3. On the target host (not production), as user root, unmount the PostgreSQL DATA and WAL mount points and deactivate the volumes.

```
# umount /mnt/pg_data
# umount /mnt/pg_log

# lvchange -a n /dev/pg_data_vg/pg_data_lv
# lvchange -a n /dev/pg_log_vg/pg_log_lv
```

4. Perform a link, or re-link operation.
  - a. If this is the first time the snapshot is linked to the target SG, perform a *link* operation of the appropriate snapshot to the target SG.

```
$ symsnapvx -sg PG_VM89_DATABASE -snapshot_name PG_VM89_Snap -lnsg
PG_VM89_DATABASE_TGT -link
```

- b. If refreshing a previously linked target, perform a *relink* operation of the appropriate snapshot to the target SG (specify the snapshot generation number if necessary).

```
# relink example of a previously linked target using generation number
# symsnapvx -sg PG_VM89_DATABASE -snapshot_name PG_VM89_Snap -lnsg
PG_VM89_DATABASE_TGT -generation 1 relink
```

5. On the target host, make the DATA and WAL volumes available. Display the volumes to ensure they are online, and then mount the file systems.

```
# lvchange -a y /dev/pg_data_vg/pg_data_lv
# lvchange -a y /dev/pg_log_vg/pg_log_lv
# lvsdisplay

# mount -t xfs -o noatime,nobarrier /dev/pg_data_vg/pg_data_lv
/mnt/pg_data
# mount -t xfs -o noatime,nobarrier /dev/pg_log_vg/pg_log_lv /mnt/pg_log
```

---

**Note:** The first time devices are linked to a new host, the host may need to be rebooted to discover them. A reboot also registers device partitions, if used. Ensure that postgres permissions are given to the file system mount points.

---

6. On the target host, as user postgres, start the database. It will run an automatic crash recovery for a short time.

```
$ pg_ctl -D /mnt/pg_data/data -l logfile start
pg_ctl: another server might be running; trying to start server anyway
server starting
```

7. After the database is running, verify that the table test contains only the first record, which was inserted prior to the snapshot creation.

```
/mnt/pg_data/data/pgbench=# select * from testTbl;
recid | recdesc
-----+-----
      1 | Initial data
(1 row)
```

This step confirms that the snapshot was linked to the target host and that committed transactions up to the point of the snapshot are available at the target database, which is a duplicate copy of production up to the snapshot time.

## Test conclusion

SnapVX can help in creating multiple point-in-time storage-based snapshots of a running PostgreSQL database that are space efficient, fast to create and restore, and easy to use. As shown, these snapshots can be used to create gold copies, to restore these gold copies back to the source database, or to create and refresh new database environments.

## Using VMAX snapshots for PostgreSQL online backups

### Benefits of using storage snapshots for PostgreSQL backups

This section describes how you can use VMAX snapshots to create and restore PostgreSQL online backups, which are taken while the database is open.

In general, there are several methods for creating PostgreSQL backups:

- **Logical backup**—You can create a SQL dump backup using the `pg_dump` command. This method is also referred to as a logical backup. A logical backup is performed from a database client connecting to the database server and reading the tables' data like a normal user. The logical backup then stores the data in a safe location.

A logical backup has two major disadvantages. First, it relies on the physical coherency of the database and cannot read or write to a database that has suffered a corruption or other disaster. The second disadvantage is that a logical backup cannot use archive logs and cannot recover the database to a point in time beyond the time it was created.

- **Offline file-system backup**—This type of backup is created after shutting down the database and making a copy of the data files to a safe location.

Most databases cannot afford to be shut down, which is a major drawback of this method.

- **Continuous archiving and point-in-time recovery (PITR) backup**—With this preferred backup method, the database remains open and continues to process transactions. Archive logs (copy of WAL files) are continuously saved and can be used during recovery to roll forward (replay) the database transactions to any point in time after the backup is done. We recommend this method of creating database backups.

For more information on PostgreSQL backup methods, refer to [Backup and Restore](#).

Continuous archiving and PITR is a more powerful backup solution than other methods because it allows you to overcome a physical database corruption, such as deleted data files, block corruption, and other disasters. This backup solution also allows you to restore all committed transactions from the archive logs and WAL to bring the database up to date (complete recovery), or to stop the recovery process at a specific time (incomplete recovery).

Without storage snapshots, once the database is placed in backup mode, a file system copy of the data files takes place. For small databases, this can be done quickly. However, as database size increases, this option is not suitable. In addition, while the database is in backup mode, the WAL writes full blocks (instead of just changes), which adds overhead. Therefore, we recommend that you reduce the time the database remains in backup mode to a minimum by using storage snapshots. Finally, a host copy also slows down recovery operations because data files must be copied back first.

### SnapVX benefits for PostgreSQL backup and recovery

Key benefits of using SnapVX snapshots for PostgreSQL backup and recovery include:

- SnapVX snapshots are created and restored in seconds, regardless of data size. All snapshots get a user-defined name and an optional duration (time-to-live). As

snapshots are pointer-based, they do not consume any capacity. Only changes made after the snapshot is taken can add capacity.

- You do not need to stop database operations as the snapshot is created. The database is in backup mode for the duration of the snapshot. As soon as the snapshot is taken, database backup mode ends.
- If a database recovery is needed, you can restore any of the snapshots taken as backups in seconds. Even if a background copy is taking place at the storage level, database recovery operations can proceed in parallel.
- The overall combination of database online backup with SnapVX snapshot technology allows for a very short Recovery Time Objective (RTO), which is the time it takes to recover the database. Recovery Point Objective (RPO) can be zero, because you can restore all committed transactions if the archive logs and latest WAL are available.

### Exclusive versus non-exclusive online backups

PostgreSQL has two distinct online-backup methods—exclusive backup and non-exclusive backup.

#### Exclusive online backups

Prior to PostgreSQL 9.6, this was the only online backup method available. However, it allows for only a single backup operation to run concurrently. According to PostgreSQL documentation, exclusive online backup can cause issues if the server crashes during the backup. For this reason, for PostgreSQL 9.6 and later, the PostgreSQL documentation recommends that you use non-exclusive backups.

When using an exclusive backup, a `backup_label` file is created automatically. That label file is required during the recovery because it contains critical metadata information regarding the end-backup time. During recovery, the label file is copied to the `$PGDATA` directory.

---

**Note:** After every backup, ensure you keep the label file in a safe place. You can rename it for easy tracking though in that case you'll rename/copy it back to its original name: 'backup\_label', if it is needed for recovery. Don't leave the file in `$PGDATA` as this directory may get refreshed as part of the recovery process.

---

An exclusive backup is specified using two parameters in the `psql` backup command:

```
SELECT PG_START_BACKUP('backup description', true);
```

- The first parameter is a user text description for the backup.
- The second parameter specifies whether to start a database checkpoint immediately. If you perform an immediate checkpoint, the overall backup time is shorter. Otherwise, the backup proceeds only after a checkpoint completes normally, which can take a long time.

For more information, refer to [Continuous Archiving and Point-in-Time Recovery \(PITR\)](#).

## Non-exclusive online backups

This method allows you to make concurrent backups. PostgreSQL documentation recommends that you use this online backup method for PostgreSQL 9.6 and later.

When running a non-exclusive backup, a `backup_label` file is not created automatically. Instead, the content of the label file is printed as part of the `stop backup psql` command. You must manually extract and save the label file.

A label file is required during the recovery because it contains critical metadata information regarding the end-backup position. During recovery, the label file is copied to the `$PGDATA` directory.

---

**Note:** After every backup, ensure you keep the label file in a safe place. You can rename it for easy tracking though in that case you'll rename/copy it back to its original name: 'backup\_label', if it is needed for recovery. Don't leave the file in `$PGDATA` as this directory may get refreshed as part of the recovery process.

---

A non-exclusive backup is specified using three parameters in the `psql backup` command: `SELECT PG_START_BACKUP('backup description', true, false);`

- The first parameter is a user text description for the backup.
- The second parameter specifies whether to start a database checkpoint immediately. If you perform an immediate checkpoint, the overall backup time is shorter. Otherwise, the backup proceeds only after a checkpoint completes normally, which can take a long time.
- The third parameter should be false, indicating a non-exclusive type backup.

For more information, refer to [Continuous Archiving and Point-in-Time Recovery \(PITR\)](#).

## SnapVX and exclusive or non-exclusive backups

When using SnapVX snapshots, the backup operation is so fast that there is no reason for concurrent backups. Both exclusive and non-exclusive online backup methods were tested successfully. Either of these methods can be used with SnapVX for online backup and recovery.

## Scripting PostgreSQL online backups

The PostgreSQL mechanism for creating online backups requires you to run the start and stop backup commands in the same `psql` session. If you exit the session, the online backup terminates prematurely and automatically. This behavior is unrelated to SnapVX but poses a challenge when scripting the backup process.

A simple way to overcome this challenge is by having the `psql` session call a shell script that waits for a known file to be created. The file is created by the script that creates the snapshot. As soon as the `psql` session sees the file, it resumes and then ends the backup. The use case in the following section demonstrates this method.

## Creating PostgreSQL recoverable replicas

### Use case overview

The following use case shows an example of the process you can use to create a valid backup snapshot of an active PostgreSQL database.

You can manage SnapVX using the Solutions Enabler CLI, Unisphere UI, or REST API. We used CLI in the following example.

The PostgreSQL commands are executed from the database host. Solutions Enabler CLI commands are executed from a storage management host, which is often not the same host as the database. If Unisphere is used, it can be installed on the storage management host as well, or VMAX embedded Unisphere can be used directly.

To simulate a running system, we used `pgbench` workload and kept it running.

To demonstrate the backup, recovery, and rolling-forward archive log concepts, we created a test table and inserted known transactions at different stages to show which of them we get back after database recovery.

### Use case setup

A prerequisite for online backups is that archive logs are enabled in the PostgreSQL database. This is done by creating a mount-point location for the archive logs. In this use case, we used `/mnt/pg_arch/` for that purpose.

The `postgresql.conf` file requires the following important updates:

- Set `wal_level` to **replica**
- Set `archive_mode` to **on**
- Point `archive_command` to the archive logs mount point.

These updates to the `postgresql.conf` file will look like this example:

```
wal_level = replica
archive_mode = on
archive_command = 'test ! -f /mnt/pg_arch/%f && cp %p /mnt/pg_arch/%f'
```

Ensure that the directory gets populated when WAL files are created during workload. The database may need to be restarted when the archive logs are enabled for the first time.

As mentioned earlier, to script the backup process and avoid exiting the `psql` session prematurely (and terminating the online backup), you can use a script that creates a delay between the start and stop of the backup by waiting for a known file to appear. The script that is creating the snapshot can also create the empty file, and the backup operation discover it and resumes.

The following shows an example of the script, where the directory and file names can be changed as necessary. The script also demonstrates how to capture the output from the `PG_STOP_BACKUP()` command so that a backup label file can be created in the case of a non-exclusive backup.

```

$ vi pg_wait.sh
# WAIT SCRIPT
while [ ! -e /tmp/snap_taken.txt ]
do
    echo "Waiting for snapshot..."
    sleep 15
done
rm -f /tmp/snap_taken.txt

$ vi pg_backup.sh
# BACKUP SCRIPT
set -x
export PGDATABASE=/mnt/pg_data/data/pgbench
export PGPORT=5432
export DATE=`date +%F_%H_%M_%S`

# call psql passing DATE as a parameter
psql -v DATE="$DATE" << EOF

\set outfile '/tmp/PG_Backup_Label_':DATE'.txt'

select checkpoints_timed, checkpoints_req, checkpoint_write_time,
checkpoint_sync_time from pg_stat_bgwriter;

SELECT PG_START_BACKUP('SnapVX backup', true, false);

select checkpoints_timed, checkpoints_req, checkpoint_write_time,
checkpoint_sync_time from pg_stat_bgwriter;

\! ./pg_wait.sh

SELECT * FROM PG_STOP_BACKUP(false);

\o
\q
EOF

```

**Creating an online backup**

To create an online backup using either the exclusive or non-exclusive backup method:

1. Start pgbench workload to simulate an active PostgreSQL system.
2. Create the test table and insert the first known record.

```

/mnt/pg_data/data/pgbench=# create table testTbl (RecId integer not null,
RedDesc varchar(50));
/mnt/pg_data/data/pgbench=# insert into testTbl values (1, 'Initial
data');
INSERT 0 1
/mnt/pg_data/data/pgbench=# select * from testTbl;
recid | recdesc
-----+-----
      1 | Initial data
(1 row)

```

3. Start the backup. In this example, we used a non-exclusive backup.

As user postgres, start the backup from the production database:

- a. Directly from psql session:

```

/mnt/pg_data/data/pgbench=# SELECT PG_START_BACKUP('SnapVX backup', true,
false);

```

- b. By using a script:

```
$ ./pg_backup.sh
```

4. Create the snapshot.

```
# symsnapvx -sg PG_VM89_DATABASE -name PG_VM89_Snap -v establish
```

If the backup was taken using a script, create the known file so the pg\_backup.sh script can proceed

```
# ssh <database_host> "touch /tmp/snap_taken.txt; chown postgres.postgres /tmp/snap_taken.txt"
```

5. If using a script, the backup will end automatically after you create the file. Otherwise, from the same session in which the online backup started, end the backup and save the output to prepare the label file that is required during recovery.

```
/mnt/pg_data/data/pgbench=# SELECT * FROM PG_STOP_BACKUP(false);
lsn | labelfile |
spcmapfile
-----+-----+-----
649/3B000210 | START WAL LOCATION: 649/3B000028 (file 0000000D000006490000003B)+|
| CHECKPOINT LOCATION: 649/3B000060 |
| BACKUP METHOD: streamed |
| BACKUP FROM: master |
| START TIME: 2017-06-16 10:50:22 EDT |
| LABEL: SnapVX backup |
|
(1 row)
```

Example of a non-exclusive backup label file extracted from the PG\_STOP\_BACKUP() psql command:

```
START WAL LOCATION: 649/3B000028 (file 0000000D000006490000003B)
CHECKPOINT LOCATION: 649/3B000060
BACKUP METHOD: streamed
BACKUP FROM: master
START TIME: 2017-06-16 10:50:22 EDT
LABEL: SnapVX backup
```

Example of an exclusive backup label file created automatically from an exclusive PG\_STOP\_BACKUP() psql command:

```
START WAL LOCATION: 642/72000028 (file 000000010000064200000072)
CHECKPOINT LOCATION: 642/72000060
BACKUP METHOD: pg_start_backup
BACKUP FROM: master
START TIME: 2017-06-12 12:48:42 EDT
LABEL: SnapVX backup
```

6. Add another record to the test table. This time, you should find this record after database recovery.

```
/mnt/pg_data/data/pgbench=# insert into testTbl values (2, 'After
snapshot taken and backup ended');
INSERT 0 1
/mnt/pg_data/data/pgbench=# select * from testTbl;

recid | recdesc
-----+-----
1 | Initial data
2 | After snapshot taken and backup ended
(2 rows)
```

## PostgreSQL database recovery

### Use case overview

The following use case shows an example of the process you can use with any of the snapshots previously created during the online database backup to restore and recover the production PostgreSQL database. Both complete and incomplete recoveries are described.

A `backup_label` file is required during recovery. The label file is created during exclusive backup, or its content is printed during non-exclusive backup and needs to be captured and saved as a `backup_label` file. The label file contains critical information about the time the backup mode ended, so the recovery process can know what is the minimal recovery needed for the data files.

The `recovery.conf` file is also required for the database recovery. The file contains information about the location of the archive logs and whether to perform a full or incomplete recovery. A sample file can be found in the PostgreSQL installation sub-directory: `./share/postgresql/recovery.conf.sample`.

You must update two important parts in the `recovery.conf` file:

- **Restore command**, which needs to know where to find the archive logs:

```
restore_command = 'cp /mnt/pg_arch/%f %p'
```

- **Recovery target**—By default, this will be a complete recovery. If an incomplete recovery is required, specify it in the recovery target:

```
recovery_target_name = ''      # e.g. 'daily backup 2011-01-26'
recovery_target_time = ''     # e.g. '2004-07-14 22:39:00 EST'
recovery_target_xid = ''
```

Both the `backup_label` and the `recovery.conf` files need to be copied to the `$PGDATA` directory after you complete the snapshot restore and the file system remount, and before the database is started. When the database finds these files during startup, it performs the database recovery.

---

**Warning:** Although we created the snapshot using the parent DATABASE SG containing both DATA and WAL, it is critical that you only restore the DATA SG. By using the parent DATABASE SG to create the snapshot, the snapshot can be used for either a restart or a recovery solution. However, in a recovery solution case, if the WAL mountpoint survived the disaster, and still contain the logs, *you do not want to over-write it* (or you'll lose the ability to recover from these logs!). Therefore, you must only use the DATA SG with SnapVX restore command. Failure to do so will overwrite the latest database WAL, and a complete recovery will not be possible.

---

### Use case setup

Prior to starting the database recovery, ensure you have done the following prerequisites:

- **Identify the correct `backup_label` file from the time of the backup**
- **Identify the correct snapshot name and generation to restore**—The snapshots can be easily listed in Unisphere or you can use the CLI command:

```
# symsnapvx -sg PG_VM89_DATA -snapshot_name PG_VM89_Snap list -detail -gb
```

For the recovery purpose, we listed the DATA SG snapshots.

- **Identify and update correctly the `recovery.conf` file**—Specifically, ensure the archive log mount points are correct in the `restore_command` section. If incomplete recovery is required, update the `recovery_target_time = ''`. The target time should correspond to any time after the stop backup time and before a complete recovery.

### Recovering a database

To recover a database:

1. Before restoring the snapshot, ensure that the database is stopped. Failure to do so may result in the database processes keeping locks on the file system.

As user `postgres`, stop the database and check for any remaining processes.

```
$ pg_ctl -D /mnt/pg_data/data stop
$ ps -ef | grep postgres
```

2. Prior to restoring the DATA SG snapshot, unmount the DATA file system and deactivate its LVM volume.

As user `root`, unmount the DATA mount points and deactivate its volume.

```
# umount /mnt/pg_data
# lvchange -a n /dev/pg_data_vg/pg_data_lv
```

3. Restore the DATA SG using the appropriate snapshot. If multiple snapshot generations exist, select the relevant one (it should match the selected label file). By default, the latest snapshot is restored (generation 0).

```
# symsnapvx -sg PG_VM89_DATA -snapshot_name PG_VM89_Snap restore
```

---

**Warning:** Ensure that only the DATA SG is restored to avoid overwriting the latest WAL.

---

4. Make the DATA volume available, then remount the DATA file system.

```
# lvchange -a y /dev/pg_data_vg/pg_data_lv
# lvdisplay
# mount -t xfs -o noatime,nobarrier /dev/pg_data_vg/pg_data_lv
/mnt/pg_data
```

5. Copy the `backup_label` and `recovery.conf` files to the `$PGDATA` location.

6. As user `postgres`, start the database. It will perform the database recovery.

```
$ pg_ctl -D /mnt/pg_data/data -l logfile start
pg_ctl: another server might be running; trying to start server anyway
server starting
```

7. Check the `pg_log` file for progress and any errors.

```
2017-06-16 11:32:20 EDT LOG:  starting archive recovery
2017-06-16 11:32:20 EDT LOG:  restored log file "0000000D.history" from
archive
2017-06-16 11:32:20 EDT LOG:  restored log file
"0000000D0000064900000003B" from archive
2017-06-16 11:32:20 EDT LOG:  redo starts at 649/3B000028
2017-06-16 11:32:20 EDT LOG:  consistent recovery state reached at
649/3B000210
2017-06-16 11:32:20 EDT LOG:  restored log file
"0000000D0000064900000003C" from archive
```

## Summary

```
cp: cannot stat â/mnt/pg_arch/0000000D0000006490000003Dâ: No such file or
directory
2017-06-16 11:32:20 EDT LOG:  invalid record length at 649/3D000098:
wanted 24, got 0
2017-06-16 11:32:20 EDT LOG:  redo done at 649/3D000028
2017-06-16 11:32:20 EDT LOG:  last completed transaction was at log time
2017-06-16 11:13:22.321379-04
cp: cannot stat â/mnt/pg_arch/0000000D0000006490000003Dâ: No such file or
directory
cp: cannot stat â/mnt/pg_arch/0000000E.historyâ: No such file or
directory
2017-06-16 11:32:20 EDT LOG:  selected new timeline ID: 14
2017-06-16 11:32:20 EDT LOG:  restored log file "0000000D.history" from
archive
2017-06-16 11:32:20 EDT LOG:  archive recovery complete
2017-06-16 11:32:20 EDT LOG:  MultiXact member wraparound protections are
now enabled
2017-06-16 11:32:20 EDT LOG:  autovacuum launcher started
2017-06-16 11:32:20 EDT LOG:  database system is ready to accept
connections
```

8. After the database has started, verify that the test table contains the first record from before the snapshot and backup, and the second record from after the snapshot and backup.

```
/mnt/pg_data/data/pgbench=# select * from testTbl;

recid |          recdesc
-----+-----
      1 | Initial data
      2 | After snapshot taken and backup ended
(2 rows)
```

This step confirms that the snapshot was restored, and committed transactions were recovered.

## Summary

The tests and guidelines provided in this white paper demonstrate the ability of PostgreSQL together with Dell EMC VMAX All Flash to provide a high-performance environment that is highly available and enriched with data services from both the database and storage.

## References

### Dell EMC documentation

The following documentation on [Dell EMC.com](#) or [Dell EMC Online Support](#) provides additional and relevant information. Access to these documents depends on your login credentials. If you do not have access to a document, contact your Dell EMC representative.

- [Dell EMC VMAX All Flash Data Sheet](#)
- [Dell EMC VBlock System 740 and VxBlock System 740](#)

### PostgreSQL documentation

The following documentation on the [PostgreSQL 9.6.3 documentation](#) website provides additional and relevant information:

- [Analyze](#)
- [Background Writer](#)
- [Backup and Restore](#)
- [Checkpoints](#)
- [Continuous Archiving and Point-in-Time Recovery \(PITR\)](#)
- [Error Reporting and Logging](#)
- [pgbench](#)
- [The Autovacuum Daemon](#)
- [The Statistics Collector](#)
- [Write-Ahead Logging \(WAL\)](#)
- [Vacuum](#)

## Appendix – Linux Logical Volume Manager

### Linux Logical Volume Manager

Logical Volume Manager (LVM) provides higher flexibility and ease of use in managing storage devices on the host. It creates an abstraction layer between the physical volumes (e.g. VMAX devices), a volume group (also called VG, which is a group of physical volumes), and a logical volume (also called LV, the construct that the host kernel presents to the application, and often is used for filesystem creation).

When not using LVM, each filesystem is created on a single VMAX device (or a partition on that device, if partitions are used). For high performance databases this may create a performance bottleneck on the host, as each device has a limited queue depth.

When using LVM, physical volumes can be added to the VG at any time, while the LV remains online. The LV capacity can be determined not by the individual physical volumes, but by the VG capacity as a whole. In addition, LVs can be striped across the physical volumes in the VG. Striping the LV offers additional performance advantages as the IOs are spread across the devices, allowing much higher total IOPS (each device adds its queue depth).

Since PostgreSQL uses filesystems for its data and logs, it is a good practice to use LVM for these filesystems. In that way, each VG consists of the appropriate devices for the

filesystem (e.g. data\_vg, log\_vg, archive\_vg, etc.). A single LV is created on each such VG (e.g. data\_lv, log\_lv, archive\_lv), striped across the devices in the VG for performance reasons. Finally, a single filesystem is created on each LV. In this way both performance and ease of management considerations are taken care of.

## LVM and filesystem creation

To create a filesystem on top of Linux LVM a few steps are required:

1. Initialize each device as a physical volume.

The following example shows how to initialize a few devices as an LVM physical devices:

```
pvccreate -f /dev/sdk
pvccreate -f /dev/sdl
pvccreate -f /dev/sdm
pvccreate -f /dev/sdn
...
```

2. Create a volume group (VG) and add the appropriate physical volumes to it.

The following example shows how to create pg\_data\_vg volume group with the first physical volume, and then add additional physical volumes to the VG:

```
vgcreate pg_data_vg --physicalextentssize 32MiB --force --yes /dev/sdk
vgextend pg_data_vg /dev/sdl
vgextend pg_data_vg /dev/sdm
vgextend pg_data_vg /dev/sdn
...
```

3. Create a striped logical volume (LV) using the VG.

The following example assumes that we've added 8 physical volumes to the VG, and shows how to create an 8-way striped logical volume. Note that we used 128KB stripe depth to match VMAX track size:

```
lvcreate --name pg_data_lv --stripes 8 --stripesize 128 --size 200G
pg_data_vg
```

4. Create a filesystem using the LV, then mount it with appropriate permissions.

The following example shows how to create an XFS type filesystem, add it to `/etc/fstab` file, and mount it:

```
mkfs.xfs -n size=64k -f /dev/pg_data_vg/pg_data_lv
mkdir /mnt/pg_data
mount -t xfs -o noatime,nobarrier /dev/pg_data_vg/pg_data_lv /mnt/pg_data
chown postgres.postgres /mnt/pg_data
```

These steps can be repeated for other filesystems, such as log, and arch. They can also be scripted for faster creation.