RDBMS in the Cloud: PostgreSQL on AWS

June 2013

Miles Ward (AWS)

Laine Campbell, Jay Edwards, and Emanuel Calvo (PalominoDB)

(Please consult **http://aws.amazon.com/whitepapers/** for the latest version of this paper.)

Table of Contents

Introduction

Amazon Web Services (AWS) is a flexible, cost-effective computing platform. Running your own relational data store on Amazon Elastic Compute Cloud (Amazon EC2) is ideal for users whose application requires the familiar operating characteristics of an RDBMS as well as the cloud's flexibility. In this whitepaper, we help you understand one of the most popular options available on Amazon EC2—the open source database system, PostgreSQL. We provide an overview of general best practices and examine important PostgreSQL characteristics such as performance, durability, and security. We also specifically highlight features that support scalability, high-availability, and fault-tolerance.

Relational Databases on Amazon EC2

AWS is an excellent platform for running traditional relational database systems (RDBMS). The public cloud provides strong benefits for database workloads. Understanding the ways that AWS differs from physical infrastructure with respect to RDBMS workloads helps you to design the best architecture possible.

Amazon EC2 Instances Versus Your Server

Let's compare a typical single, 1U rack-mount server to an EC2 instance.

At first glance, these two computers are similar: they provide roughly equivalent CPU, RAM, local disk resources, and network infrastructure. However, the operational differences are enormous:

- The EC2 instance is rapidly replaceable, duplicable, and available on-demand.
- The EC2 instance can dynamically grow and shrink from a single logical CPU with 1.7GB of RAM up to 16 logical CPUs with 244GB of RAM. This requires a simple configuration change through the API or CLI and an instance reboot.
- The EC2 instance only costs you money while powered on. Shutting down even part of your fleet during non-peak times can save significant budget dollars. Persistent storage options protect your data, but have on-going costs even when your instances are "stopped."
- The EC2 instance is supported by the AWS network and facilities infrastructure; you never have to touch the hardware.
- While there is no contention for CPU or memory, the network itself is a shared resource. You might have access to only a fraction of the physical infrastructure's network connection depending on instance size.
- AWS facilities (called Availability Zones) are likely larger than your network environment, and EC2 instances (except for some specialized instances) start in random physical locations within the Availability Zone. This is good for reliability, but server-to-server communications may have higher latencies than on a smaller local network.
- Because of virtualization overhead, there are non-rounded memory sizes (613MB, 1.7GB, 7.5GB, 15GB, 17.1GB, 22GB, 23GB, 34.2GB, 68.4GB, etc.); applications tuned for specific memory footprints might need to opt for more memory than absolutely required or be retuned for these sizes.

- There is more local disk storage (referred to as “instance storage”) on the EC2 instance than our example server; however, instance storage is ephemeral and is deleted when the instance is stopped or terminated. We recommend that you use persistent storage resources in addition to your EC2 instance.

In addition to your EC2 instance, Amazon provides other valuable resources available via our internal network:

- **Amazon Elastic Block Store (Amazon EBS)**—Amazon EBS volumes are durable, high-performance, network-attached block device resources. These “virtual disks” can be attached to your servers and can persist when servers are stopped or terminated, providing durable storage for databases. Amazon EBS volumes that operate with 20GB or less of modified data after their most recent snapshot can expect an annual failure rate (AFR) between 0.1% – 0.5%.

- **Amazon Simple Storage Service (Amazon S3)**—Amazon S3 provides a highly durable storage infrastructure designed for mission-critical and primary data storage; it provides backup storage for snapshots of Amazon EBS disks as well as any other static content your application needs. Amazon S3 is designed for 99.999999999% data durability, making it an ideal target for your database backups.

- **Amazon CloudWatch**—CloudWatch is the AWS monitoring service. It provides detailed and customizable CPU, disk, and network utilization metrics for each enabled EC2 instance and Amazon EBS disk. This data is available in the web-based AWS Management Console as well as through the API, allowing for infrastructure automation and orchestration based on availability and load metrics.

PostgreSQL on Amazon EC2

Overview

PostgreSQL is an open-source RDBMS known for a rich set of features and extraordinary stability. A strong focus on performance enhancements in recent releases has enabled PostgreSQL to become a strong competitor to other database solutions on the market today. PostgreSQL provides full ACID compliance for applications requiring reliability and durability.

Using this Whitepaper

Items beginning with “$ “ are entered at a `Bash` shell prompt. Items beginning with “ > “ are typed into the PostgreSQL shell and represent commands to the PostgreSQL database process.

Concepts

To get started, let’s clarify some concepts and terminology used in this whitepaper.

A PostgreSQL *master host* accepts both writes and reads, and may have many replicas. Records are transferred to the replicas via *write-ahead logging* (WAL). The current PostgreSQL community version allows only one master, although there are third-party solutions that provide multi-master clustering.

A *secondary host* receives WAL records from the master. Replication can be real-time through streaming replication or delayed through WAL archiving.

A *hot standby* is a secondary host that can receive read queries. PostgreSQL supports a warm standby state—a host that receives WAL archives but does not receive traffic.

Streaming replication is the native PostgreSQL method of real time replication, and is akin to MySQL's row-based replication.

Replication on PostgreSQL supports two levels of durability: *asynchronous* and *synchronous*. Only one replica can be in synchronous mode. You may provide an ordered list of candidate synchronous replicas if the primary replica is down. Requiring synchronous replication can cause severe performance degradation in cases where the network connection between the master and replica is not high quality.

For version 9.2 and later, PostgreSQL supports *cascading replication*, so that replicas transfer WAL records from the primary server to other hosts, and create a replication topology. You can run the backups against any of the replicas and use them to build new cascading replicas.

If you want faster replicas and do not mind rebuilding them each time they restart, consider using SSD storage for their data. It is a good practice to have one replica for seeding new copies online using `pg_basebackup`.

Basic Installation

Here's how to get started with PostgreSQL on AWS.

1. Launch an EC2 instance using the AMI of your choice. (For this example, use the Amazon Linux 64-bit AMI.)

2. Create an Amazon EBS volume to use for your PostgreSQL storage, and attach it to the instance.

   Note: You need the operating system device name (/dev/xvdc for instance) to complete step 6.

3. Connect to the instance by SSH.

4. Make a file system on your Amazon EBS volume:

```
$ yum install xfsprogs
$ sudo mkfs -t xfs /dev/xvdc
```

5. Make a directory to serve as a mount point:

```
$ sudo mkdir -p /data
$ sudo chown `id -u` /data
```

6. Edit your fstab to mount the volume on startup:

```
$ sudo –I
$ echo '/dev/xvdc /data auto noatime,noexec,nodiratime 0 0' >> /etc/fstab
```

7. Mount the volume:

```
$ sudo mount -a /dev/xvdc /data
```

8. Either download and install PostgreSQL from the source, or install it as a package:

a. Basic source installation:

```
$ wget http://ftp.postgresql.org/pub/source/v9.2.1/postgresql-9.2.1.tar.gz

$ tar xzvf postgresql-9.2.1.tar.gz
```

i. Install the following packages:

```
$ sudo yum install zlib-devel.x86_64 readline-devel.x86_64 python27-
devel.x86_64 python27.x86_64  perl-ExtUtils-MakeMaker.x86_64 perl-
ExtUtils-CBuilder.x86_64 perl-ExtUtils-Embed.x86_64

$ ./configure --prefix=/opt/pg --with-libxml --with-libxslt --with-perl
--with-python

$ make ; sudo make install
```

We are installing PostgreSQL with Perl and Python support. This is necessary if you want to use those languages for triggers or install third-party replication tools trigger based on them (for example, Bucardo uses Perl, Londiste uses Python).

The prefix location is for binaries and libraries.

b. If you aren't comfortable building from the source, you can install a binary package from the AWS repository. Check the most recent version available on the repository, and install:

```
$ yum info postgresql9-server.x86_64 | grep Version

$ yum install postgresql9-server
```

9. Both installation methods require you to initialize the cluster data folder for Amazon EC2:

```
sudo -u postgres /opt/pg/bin/initdb -D /data/
```

Check /etc/init.d/postgresql to ensure that the PGATA variable points to the mounted data directory:

```
PGDATA=/data/pg
```

10. Verify permissions for data and binaries.

a. If you compiled from the source:

```
sudo useradd postgres ; chown -R postgres: /data ; chown -R postgres:
/opt/pg
```

b. If you installed binary packages:

```
chown -R postgres: /data
```

11. Edit your Amazon EC2 security group to allow ingress from your application servers to the database server on port 5432.

12. Pre-configuration steps:

    a. Edit the postgresql.conf file so that the postgres.exe or PostgreSQL process listens on every IP address:

    ```
    listen_addresses = '*'
    ```

    b. If you are planning to run a master-slave configuration from the beginning, you may want to set `max_wal_senders` > 0. This variable indicates the number of processes that ship WAL records to replicas. Each replica requires `max_wal_sender` to be incremented by one. Changing this variable require a restart so you need to plan carefully. A total of four connected replicas is more than enough for most situations. If you need more than four replicas, you should run cascade streaming to avoid overloading the master with the business of shipping WAL records.

13. Start PostgreSQL.

    a. If you compiled from the source:

    ```
    /opt/pg/bin/pg_ctl start -l logfile -D /data/pg
    ```

    b. If you installed binary packages:

    ```
    service postgresql start
    ```

Temporary Data and SSD Instance Storage

As an advanced technique, you can create a normal tablespace on instance storage with unlogged tables to take advantage of increased performance available with SSDs, like those available on Amazon EC2's hi1.4xlarge and cr1.8xlarge instance types. It is important to remember that instance storage isn't permanent and will be lost if your EC2 instance is terminated or fails. This technique is only suited for data that you can afford to lose, typically because you have it replicated elsewhere in your setup.

When you create a new table, query the `relfilenode` of the new table and back up the filesystem identified by the query results into permanent storage. (Be sure to do this before you put any data in the table.)

To restore, just copy the backup to the same location, and run the ANALYZE command on those tables. This technique is compatible with unlogged tables if you want to have a primary-replica configuration. This avoids replication inconsistencies, as unlogged tables do not get replicated. For standalone servers, you can use this technique with permanent tables. You can create handmade materialized views here too.

Step-by-Step Laboratory

This example walks through a basic example of a temporary data configuration over ephemeral storage, and what to do after a server shutdown.

```
postgres=# CREATE UNLOGGED TABLE prueba(i serial primary key, something
text);
NOTICE:  CREATE TABLE will create implicit sequence "prueba_i_seq" for
serial column "prueba.i"
NOTICE:  CREATE TABLE / PRIMARY KEY will create implicit index
"prueba_pkey" for table "prueba"
CREATE TABLE
```

The OID column contains the name under the folder “base” of the data directory where the relfilenode resides. Copy that file into a permanent storage:

```
postgres=# SELECT relfilenode, (SELECT oid FROM pg_database o WHERE
o.datname = current_database()::text) FROM pg_class WHERE relname =
'prueba';
 relfilenode |  oid
-------------+-------
    16386 | 12870
(1 row)

[root@ip-10-250-227-15 temp_ts]# ls -l
total 0
drwx------ 3 postgres postgres 18 Dec  3 01:24 PG_9.2_201204301
[root@ip-10-250-227-15 temp_ts]# ls -l PG_9.2_201204301/
total 0
drwx------ 2 postgres postgres 129 Dec  3 01:24 12870
[root@ip-10-250-227-15 temp_ts]# ls -l PG_9.2_201204301/*
total 24
-rw------- 1 postgres postgres    0 Dec  3 01:24 16396
-rw------- 1 postgres postgres    0 Dec  3 01:24 16396_init
-rw------- 1 postgres postgres    0 Dec  3 01:24 16397
-rw------- 1 postgres postgres    0 Dec  3 01:24 16397_init
-rw------- 1 postgres postgres 8192 Dec  3 01:24 16398
-rw------- 1 postgres postgres 8192 Dec  3 01:24 16398_init
-rw------- 1 postgres postgres    0 Dec  3 01:24 16401
-rw------- 1 postgres postgres    0 Dec  3 01:24 16405
-rw------- 1 postgres postgres 8192 Dec  3 01:24 16407

[root@ip-10-250-227-15 ~]# mkdir /data/temp_ts
[root@ip-10-250-227-15 ~]# chown -R postgres: /data/

postgres=# CREATE TABLESPACE temp_ts LOCATION '/data/temp_ts';
CREATE TABLESPACE

postgres=# ALTER TABLE prueba SET TABLESPACE temp_ts;
ALTER TABLE

postgres=# CREATE TABLE prueba2 (i serial primary key, misc text, start
timestamp)  TABLESPACE temp_ts;
NOTICE:  CREATE TABLE will create implicit sequence "prueba2_i_seq" for
serial column "prueba2.i"
NOTICE:  CREATE TABLE / PRIMARY KEY will create implicit index
"prueba2_pkey" for table "prueba2"

CREATE TABLE

[root@ip-10-250-227-15 data]# cp -R temp_ts/ bak_temp_ts/

postgres=# INSERT INTO prueba(something) VALUES ('o');
INSERT 0 1
postgres=# INSERT INTO prueba2() VALUES ('o');
i   misc   start
postgres=# INSERT INTO prueba2(start) VALUES (now());
INSERT 0 1
```

What happens to the table if nothing exists on the ephemeral storage? You are not able to access the table itself. Consequently, PostgreSQL raises an error. This example shows a file deletion and a subsequent attempt to access the table:

```
[root@ip-10-250-227-15 data]# rm -rf temp_ts/
[root@ip-10-250-227-15 data]# psql -Upostgres
psql (9.2.1)
Type "help" for help.

postgres=# SELECT * FROM prueba;
ERROR:  could not open file
"pg_tblspc/16395/PG_9.2_201204301/12870/16396": No such file or directory
```

How do you restore the contents? Just copy the content from your backup to the temporary folder:

```
[root@ip-10-250-227-15 data]# cp -R bak_temp_ts/ temp_ts
[root@ip-10-250-227-15 data]# chown -R postgres: temp_ts/
```

The tables are empty, as expected:

```
postgres=# SELECT * FROM prueba2;
 i | misc | start
---+------+-------
(0 rows)
postgres=# SELECT * FROM prueba;
 i | something
---+-----------
(0 rows)
```

Architecture

The design of your PostgreSQL installation on Amazon EC2 is largely dependent on the scale at which you are trying to operate. Given the building blocks of a master and streaming replication, how would a system scale to accommodate an increasing load over time?

- **Functional partitioning**—Separate distinct workloads to their own masters and replicas to minimize contention for resources.
- **Vertical scaling**—Scale to the largest sizes of instances and storage as AWS can provide for each component.
- **Tuning**—Carefully tune for the available hardware including connection pool tuning, archival and purging of unneeded data, or using PostgreSQL partitioning where appropriate.
- **Replication**—If bound by reads, whether on I/O or CPU, use replication to create multiple replicas to distribute query load.
- **Sharding**—If bound by writes, sharding your data set across multiple clusters is an appropriate next step.

Anti-Patterns

- Vertical scaling doesn't offer all of the same benefits of horizontal scaling. Higher-performance instances can provide many of the performance benefits of a more complex replicated and sharded topology, but they offer none of the significant fault-tolerance benefits.
- Scaling step by step when you know you need a big system isn't efficient. This is an exercise in probability of success. Planning for growth in AWS is easier because of your ability to scale both quickly. You do need a plan for identifying growth triggers.
- ACID compliance has a cost. If logs or session data are a large subset of your data, you may not need the guarantees offered by ACID and should consider a NoSQL key store such as Amazon DynamoDB, Cassandra, or Riak.
- You might not need to do everything in your database. You can keep your databases running a lot more smoothly by adding caching tiers, using specialized search engines, and off-loading certain work to your application servers.

Performance Suggestions

Storage

In many ways, the methods you use to scale your I/O workload are so closely tied to the characteristics of your specific case that it is difficult to provide detailed information. For instance, increasing the number of Amazon EBS volumes in a software RAID set increases performance up to a certain number. While we have seen 8 volumes work well for some workloads, we've seen 22 volumes work better for other workloads. With that caveat in place, we provide some general guidance and suggest that you test everything.

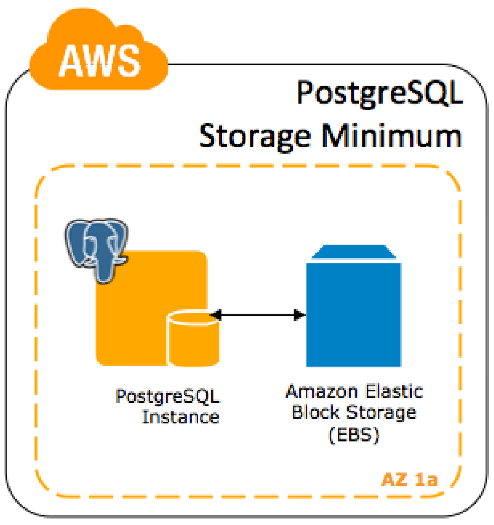

Figure 1: Minimum Storage Scale: Use Amazon EBS

Minimum production scale

Always use Amazon EBS. Amazon EBS has significant write cache, superior random I/O performance, and provides enhanced durability in comparison to the ephemeral disks on the instance. If you are going to use ephemeral disk on instances that expose more than a single volume, mirror those volumes to enhance operational durability. Remember, if the instance is lost or terminated, you'll lose all of your data even though the volumes are mirrored. Non-SSD instance storage is slower than Amazon EBS; don't put data or WAL files on it.

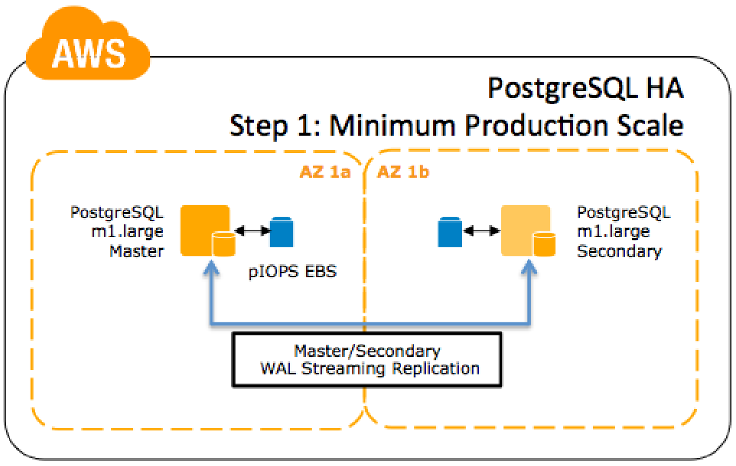
AWS
PostgreSQL HA
Step 1: Minimum Production Scale
AZ 1a
AZ 1b
PostgreSQL m1.large Master
pIOPS EBS
PostgreSQL m1.large Secondary
Master/Secondary WAL Streaming Replication

Figure 2 Basic Multi-AZ PostgreSQL architecture

Medium production scale

Move up to higher bandwidth instance types (m1.xlarge, c1.xlarge, or m2.4xlarge), and increase the number of volumes in your RAID set.

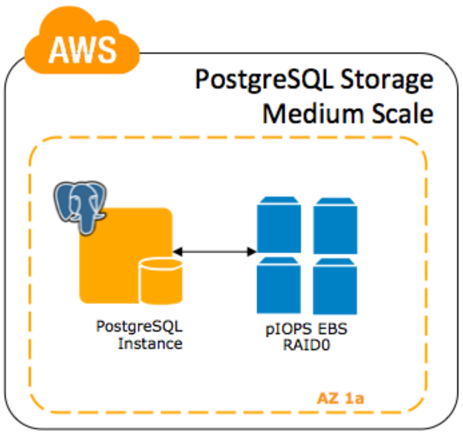

Figure 3: Medium Scale: pIOPS Amazon EBS RAID0

Extra-large production scale

If you are working in our US East (Northern Virginia), US West (Oregon), Asia Pacific (Tokyo), or EU (Ireland) regions, AWS offers Cluster Compute instances, which have more bandwidth for communications with Amazon EBS; CC2 and CR1 instances would make excellent primary nodes, particularly when paired with a large number of Amazon EBS volumes (8 or more).

Amazon EBS with Provisioned IOPS volumes in a RAID set can support very I/O-intensive applications.

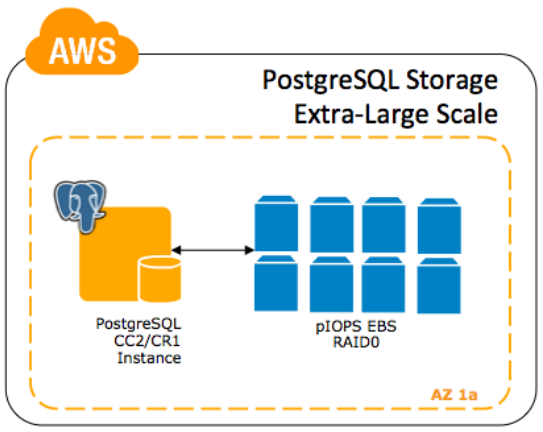

Figure 4: Extra-Large Scale: Cluster Compute and pIOPS RAID0

Alternately, you can use a hi1.4xlarge instance with two ephemeral SSD volumes. These are well suited for replicas, where you can rebuild from backups or using pg_basebackup online.

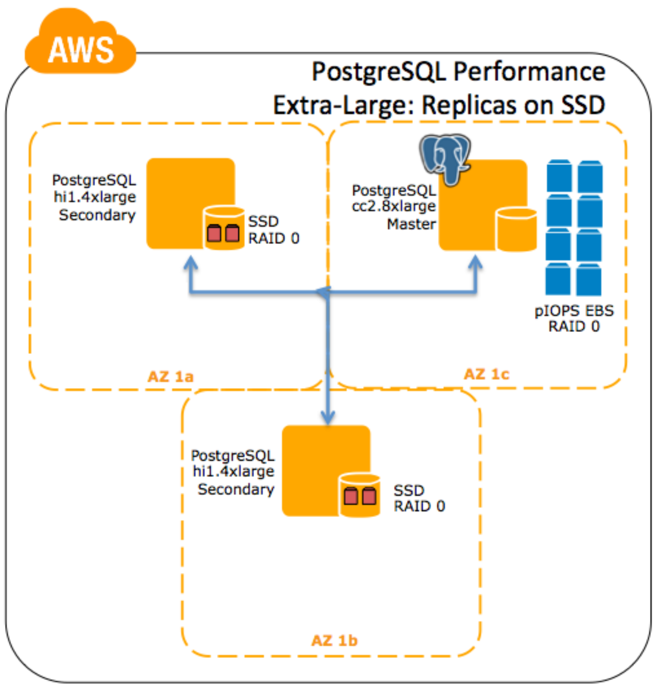

Figure 5: Extra-Large Scale: Replicas on SSD

- RAID configuration could be improved from the PostgreSQL configuration setting `effective_io_concurrency`, which needs to be set to the number of stripes of your RAID.
- If you have replicas running on SSDs, you can disable all the integrity features like fsync and `full_page_writes` on those hosts to improve the performance. The throughput on the COMMIT rate improves, as shown in the following graph (Figure 6: the orange rectangle represents the commit graph with the integrity features enabled and the green rectangle represents the graph with those variables disabled).

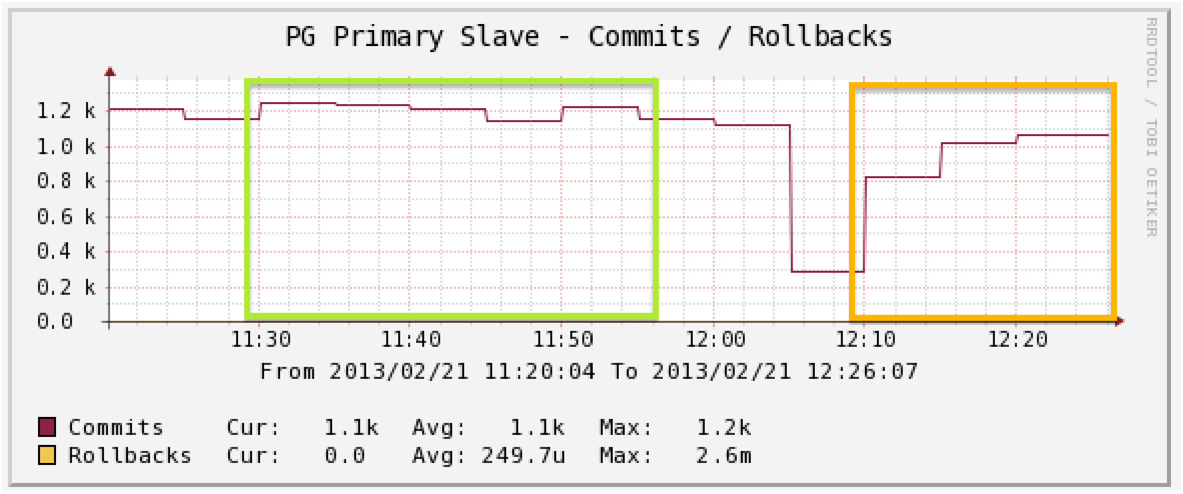

Figure 6 Improved throughput with integrity features enabled (orange rectangle)

- Consider using PIOPS (http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/EBSPerformance.html).
- How to calculate the IOPS performance? The basic formula is:

  *(NUM_IOPS * BLOCK_SIZE) /1024 = Megabytes/Sec*

  That is, 400 IOPS is only 3 MB/s at 8K block size.

Amazon EBS Details

Amazon EBS has important details worth noting for your production deployments:

- There are default account limits that you can increase by filling out a request at http://aws.amazon.com/contact-us/ebs_volume_limit_request/.
- The maximum IOPS per Provisioned IOPS volume is 4000, which requires a minimum of 400 GB provisioned storage and the use of the EBS-optimized flag. These can be aggregated up to the expected throughput maximum for the instance you've selected.
- Non cluster-compute EC2 instances using the EBS-optimized flag are connected to Amazon EBS through a 1000mbps or 500mbps link, so the theoretical maximum performance will be ~120 MB/s or ~60 MB/s respectively.

Benchmarking AWS Storage

If you are using Amazon EBS, you are using a networked device. We strongly suggest that you benchmark your storage after it is created. If you are migrating from a system in your own datacenter, this is a great chance to compare performance and confirm that you are setting up Amazon EBS correctly to meet your throughput needs.

To do basic disk tests, you can use dd (sequential) and sysbench.

- Sequential test example command:

  ```
  dd if=/dev/zero of=<location in the disk> bs=8192 count=10000 oflag=direct
  ```

- Seek test example command:

  ```
  sysbench --num-threads=16 --test=fileio --file-total-size=3G --file-test-
  mode=rndrw prepare
  sysbench --num-threads=16 --test=fileio --file-total-size=3G --file-test-
  mode=rndrw --file-fsync-all run
  sysbench --num-threads=16 --test=fileio --file-total-size=3G --file-test-
  mode=rndrw cleanup
  ```

NOTE: For more aggressive testing, you can add the `--file-fsync-all` option. We recommend this if you want to benchmark and compare different filesystem types (EXT4 vs. XFS, for instance).

To test performance through PostgreSQL, you can use pgbench. When you understand whether constraints exist in your storage layer or in your RDBMS layer, you can configure your system better.

- Pgbench command example:
  - Install the set with the respective scale:

    ```
    pgbench -i -s1000 -Upostgres database
    ```

  - Run a simple test with 20 clients with 100 transactions each against the master:

    ```
    pgbench -c 20 -t 100 -Upostgres database
    ```

  - Also, you can run a “only-read/no vacuum” test against the slave:

    ```
    pgbench -S -n -c 20 -t 1000 -h slave -Upostgres database
    ```

If you are planning to use a query balancer (such as pgpool), remember to test against the balancer instead of directly against the databases.

Operations

Maintenance and Vacuuming

One of the main maintenance tasks of a PostgreSQL instance is the vacuum. Vacuuming recovers free space available on the blocks from old versions or deleted rows.

Autovacuum is enabled by default; we recommend not turning it off. We also suggest scheduled vacuum tasks on off-hours. The primary tunable variable for the autovacuum process is the number of concurrent workers. By default, `autovacuum_workers` is set as 3. You can set the value for `autovacuum_workers` to between 5 and 8 in most production environments. Also, decreasing the thresholds/factors per table basis can help for the most updated tables. It is fine to change the thresholds during the day according to your system's load. Remember that vacuuming is expensive, so you may not want run this process in the middle of a high workload. The ANALYZE command on its own is less expensive but can also consume important resources.

For example, if you want to have statistics updated more frequently on a specific table, you could execute one or both of the following SQL statements:

```
ALTER TABLE stat SET (autovacuum_analyze_threshold=20);
ALTER TABLE stat SET (autovacuum_analyze_scale_factor=0.10);
```

The ANALYZE command is triggered according to the following formula:

```
autovacuum_analyze_threshold + (autovacuum_analyze_scale_factor * # rows
in table)
```

In large or busy tables, you should generally reduce the scale factor to update statistics more frequently.

If you have good maintenance windows or low-load hours, you may prefer to run vacuums during those hours and increase the `autovacuum_analyze_threshold` value during working hours.

However, autovacuum isn't enough. It was designed to run concurrently and with minimal impact against the server. Autovacuum tasks can be cancelled during increased periods of load with minimal impact. You should schedule normal vacuums during non-peak hours.

A sample line, on the crontab, to vacuum and analyze a specific table every 6 hours:

```
0 */6 * * * /opt/pg/bin/vacuumdb -z -d <database> -t <table> -
Uuser_for_maintenance
```

Read-Only Servers

For some operations or specific projects, you may need to put your server in a read-only state. There is typically no need to put the server in this state for general maintenance.

To do this, you need to add the following two variables in the postgresql.conf file and then reload the server with pg_ctl reload:

```
transaction_read_only=on

default_transaction_read_only=on
```

To start accepting writes again, uncomment or delete these variables and reload the server again.

Back Up Using an Amazon EC2 Snapshot

If you want to snapshot a mounted volume, execute the following command:

```
SELECT pg_start_backup('label',true);
```

Note that this operation is non-blocking.

The second parameter turned as “true” forces the backup to start as soon as possible, but affects the query processing due to higher I/O operations. By default, this value is "false" and backups can take a while to finish. From the official documentation (http://www.postgresql.org/docs/9.2/static/continuous-archiving.html):

> “This is because it performs a checkpoint, and the I/O required for the checkpoint will be spread out over a significant period of time, by default half your inter-checkpoint interval (see the configuration parameter checkpoint_completion_target). This is usually what you want, because it minimizes the impact on query processing.”

Perform the snapshot:

```
ec2-create-snapshot -d "postgres clon"  vol-24592c0e

SNAPSHOT    snap-219c1308    vol-24592c0e    pending    2012-12-
03T01:34:12+0000      052088341151    10    postgres clon

SELECT pg_stop_backup();
```

Note: For the duration of a snapshot (until it's listed as “completed”), there is a variable impact to Amazon EBS disk performance. If you are operating near maximum I/O capacity, we recommend that you use a replica for this purpose.

For more information about consistent snapshot backups, go to http://alestic.com/2009/09/ec2-consistent-snapshot.

Restore Using an Amazon EC2 Snapshot

To restore from a backup, follow these steps:

1. Check the available snapshots:

```
$ ec2-describe-snapshots
SNAPSHOT    snap-219c1308    vol-24592c0e    completed    2012-12-
03T01:34:12+0000    100%    052088341151    10    postgres clon
```

2. Create an Amazon EBS volume from each snapshot used to back up your data (this could be more than one volume). You can find the snapshot IDs (and create the volumes, if you like) in the AWS Management Console or using the `ec2-describe-snapshots` CLI tool.

```
$ ec2-create-volume --snapshot snap-219c1308 --availability-zone eu-west-1c
VOLUME    vol-eb1561c1    10    snap-219c1308    eu-west-1c    creating
2012-12-03T10:13:44+0000
```

3. Attach the volumes to the instance. Remember, if you are restoring a RAID set, replace them in the same order for easiest re-creation of the RAID volume in the OS.

```
$ ec2-attach-volume -i i-96ec5edd -d /dev/sdc vol-eb1561c1
ATTACHMENT    vol-eb1561c1    i-96ec5edd    /dev/sdc    attaching    2012-
12-03T10:23:37+0000
```

4. Mount the volume and assign the corresponding permissions:

```
$ dmesg | tail
[3889500.959401] blkfront: xvdc: barrier or flush: disabled
[3889500.961193]  xvdd: unknown partition table
[root@ip-10-208-8-123 ~]# mkdir -p /data
[root@ip-10-208-8-123 ~]# chown -R postgres: /data
# echo "/dev/xvdd /data auto noatime,noexec,nodiratime 0 0" >> /etc/fstab
# mount -a
```

With this restore—depending on how you configured the master—you are able to restore to a point-in-time recovery, configure a new replica, or use a restored standalone instance for testing.

Storing Backups and WAL Files

There is a specific project to work around WAL and backups over Amazon S3:

```
https://github.com/wal-e/wal-e
```

You need to install the dependencies and run `setup.py install`. You'll need to have the AWS_ACCESS_KEY_ID and the AWS_SECRET_ACCESS_KEY values loaded in your user environment.

The following commands are available:

- **backup-push**—Stores the full backup on the Amazon S3 buckets.
- **backup-fetch**—Gets the full backup for restore purposes from Amazon S3.
- **wal-push**—Stores the WAL files using `archive_command`.
- **wal-fetch**—Gets the WAL files using `restore_command`.
- **backup-list**—Gets the backup list.
- **delete**—Deletes pieces prior to a specified date.

This tool supports encryption, GPG-based, load measurement, and compression.

These types of backups aren't Amazon EBS snapshots so they can take some time to complete. The advantage is that they allow restoration on non-AWS machines.

PostgreSQL Replication

In this section, you see how to configure a master, replica, and cascading replica. The cascading server is configured to run as a Multi-AZ deployment. You also walk through using replicas as backup targets.

Basic Streaming Setup

You set up a streaming replication-based slave by restoring a backup to a separate instance before configuring replication.

When you have attached and mounted the snapshot volume, you can configure streaming replication between the hosts. This can be broken down into five steps: create the user, give the account grants to run replication processes, grant access to the server, configure the slave to point to the primary server, and start replication.

On the new database server, create a minimal set of configuration files:

```
/data/pg/recovery.conf
standby_mode = on
primary_conninfo = 'host=10.250.227.15 port=5432 user=repl password=repl'

postgresql.conf
host_standby = on
```

If you are planning to have all your nodes as hot standby nodes, you can enable this variable on the master, so that the variable maintains its value when you run a snapshot of the data.

There are variables that must be the same across the entire cluster, such as `max_connections` or `wal_level`.

You can use a replica for backups. The following backup tools are available:

- **snapshot**—This technique is fast, but requires some additional steps.
- **pg_basebackup**—You can use it for direct backups or for store them for PITR.
- **wal-e**—It allows you to store backups into Amazon S3 buckets.
- **pg_dump / pg_dumpall**—These tools are not for replication setup purposes.

An example of running pg_basebackup online:

```
pg_basebackup -D new_data -U replication_user -h source_host -p
source_port
```

Minimal Master Configuration

On the master, we need to assure that the WAL verbosity is set high enough for streaming replication and that the replication user is created (by default, there is no replication user created).

Minimal WAL configuration setup:

```
wal_level=hot_standby
wal_keep_segments=<recommended to start >800 >
max_wal_senders=<number of servers you are planning to set up is the
minimal>
```

User creation and authentication configuration:

```
$ psql -Upostgres
postgres=# CREATE USER repl WITH PASSWORD 'repl';
```

```
CREATE ROLE

postgres=# ALTER USER repl REPLICATION;
ALTER ROLE

# echo "host    replication       repl   10.208.8.123/32         md5"  >>
pg_hba.conf
# service postgresql reload
```

Tunables

The behavior of PostgreSQL is significantly impacted by the settings of internal configuration values, or tunables. Some significant values include:

- Swappiness, vm, kernel tuning
  - By default, shmmax and shmall have really small values. Those values are linked to `shared_buffers` in postgresql.conf; if this value is higher than the kernel parameters, PostgreSQL won't start.
  - We recommend setting up vm.swappiness with a value under 5. This avoids using swap space unless it is necessary.
- Filesystem tuning
  - XFS (`nobarrier`,`noatime`,`noexec`,`nodiratime`)
  - ext3 or ext4
    - You can use ext3 or nonjournaled filesystems for logs.
- Memory tuning
  - `shared_buffers` is the most important and difficult memory variable to tune up. A starting recommendation could be to start with a quarter of your RAM.
- WAL
  - We strongly recommend separating the data from the pg_xlog (WAL) folder. For the WAL files, we strongly recommend the XFS filesystem, due to the high amount of fsync generated.
  - The value of the `checkpoint_segments` variable depends strictly on the amount of data modified on the instance. At the beginning, you can start with a moderate value and monitor the logs looking for HINTS, which are logged as follows:

    ```
    pg_log/postgresql-2012-12-02_134148.log:LOG:  checkpoints are
    occurring too frequently (5 seconds apart)
    pg_log/postgresql-2012-12-02_134148.log:HINT:  Consider increasing
    the configuration parameter "checkpoint_segments".
    ```

  - File segments are 16 MB each so it will be easy to fill them if you have a batch of processes adding or modifying data. You could easily need more than 30 file segments on a busy server.

  - We recommend not using the ext3 filesystem if you plan to have the WALs in the same directory as the data. This filesystem handles fsync calls inefficiently.

- Pgtune
  - Pgtune is a Python script that recommends a configuration according to the hardware on your server. For basic installation and execution:

```
wget https://github.com/gregs1104/pgtune/archive/master.zip
unzip master.zip
./pgtune -i /opt/pg/data/postgresql.conf -o test.conf -TOLTP
```

  - Pgtune supports three interesting options:
    - -M TOTALMEMORY, --memory=TOTALMEMORY

      Total system memory, will attempt to detect if unspecified
    - -T DBTYPE, --type=DBTYPE

      Database type, defaults to Mixed, valid options are DW, OLTP, Web, Mixed, Desktop
    - -c CONNECTIONS, --connections=CONNECTIONS

      Maximum number of expected connections, default depends on the database type
  - The new lines are followed by the “CUSTOMIZED OPTIONS” label:

```
# egrep 'CUSTOMIZED' -A 40 test.conf
# CUSTOMIZED OPTIONS
#---------------------------------------------------------------------
----------
# Add settings for extensions here
#---------------------------------------------------------------------
----------
# pgtune wizard run on 2013-02-21
# Based on 1696956 KB RAM in the server
#---------------------------------------------------------------------
----------

default_statistics_target = 100
maintenance_work_mem = 96MB
checkpoint_completion_target = 0.9
effective_cache_size = 1152MB
work_mem = 5632kB
wal_buffers = 8MB
checkpoint_segments = 16
shared_buffers = 384MB
max_connections = 300
```

  - Pgtune advises about the most common variables in the postgresql.conf file; however, there are some factors and variables that depend directly on usage. The `max_connections` value is a perfect example of this; 300 could be considered high for some environments.

- Pgtune currently supports PostgreSQL version 9.1. You need to be careful with the advice if you use it on a newer version.

Monitoring

AWS provides robust monitoring of EC2 instances, Amazon EBS volumes, and other services using the Amazon CloudWatch service. CloudWatch can alarm, via SMS or email, upon user-defined thresholds for individual AWS services. One example would be triggering an alarm based on excessive storage throughput, as shown here. Another approach would be to create a custom metric in CloudWatch—current free memory, for instance—and to alarm or trigger automatic responses from those measures.

The main things you should monitor on a PostgreSQL cluster:

- `checkpoint_segments` warnings: if an instance runs out of checkpoint segments, it raises a warning in the log. You can detect if your instance needs to increase the `checkpoint_segments` value by monitoring the logfile for that error.

- Number of connections.

- Memory usage and load average.

- Slow queries: execute the following query:

```
SELECT pid, QUERY FROM pg_stat_activity WHERE (query_start - now()) > '30
seconds'::interval;
```

- Replication lag: On the replica, execute the following query:

```
SELECT ( extract('epoch' from now()) - extract('epoch' from
pg_last_xact_replay_timestamp())) AS result;
```

Using Amazon CloudWatch Custom Metrics

You need first to download and configure the tool environment. For more information, see http://docs.aws.amazon.com/AmazonCloudWatch/latest/DeveloperGuide/choosing_your_cloudwatch_interface.html#tools-download.

The latest version is: http://ec2-downloads.s3.amazonaws.com/CloudWatch-2010-08-01.zip.

For more information on CloudWatch metrics, see http://docs.aws.amazon.com/AmazonCloudWatch/latest/DeveloperGuide/publishingMetrics.html.

The following example turns on instance monitoring, pushes some metrics, and gets the statistics:

```
$ ec2-monitor-instances i-08fe4e43
i-08fe4e43    monitoring-pending

# while true ; do CloudWatch-1.0.13.4/bin/mon-put-data --metric-name
backends --namespace Postgres --dimensions "InstanceId=i-08fe4e43" --
value `psql -Upostgres -Atc 'SELECT sum(numbackends) FROM
```

```
pg_stat_database'` --timestamp `date +%Y-%m-%dT%H:%M:%S.000Z` ; sleep 60
; done
# CloudWatch-1.0.13.4/bin/mon-list-metrics  | grep -i backends
backends                                                        Postgres
{InstanceId=i-08fe4e43}

# CloudWatch-1.0.13.4/bin/mon-get-stats backends --namespace Postgres --
statistics "Average,Maximum" --dimensions "InstanceId=i-08fe4e43" --
start-time 2013-03-04T23:00:00.000Z
2013-03-05 13:15:00  1.0  1.0  None
2013-03-05 13:16:00  1.0  1.0  None
2013-03-05 13:17:00  1.0  1.0  None
2013-03-05 13:22:00  1.0  1.0  None
2013-03-05 13:23:00  1.0  1.0  None
2013-03-05 13:24:00  1.0  1.0  None
…
```

The previous example was a simple test to store and retrieve the statistics on the number of backends. Some statistics require collection in scripts before you push a value to CloudWatch.

Security

This is a large topic. The only aspect we cover in some detail in this whitepaper is configuring SSL communication between the servers.

Disk Encryption

PostgreSQL has no native support for encrypting the data folder, but PostgreSQL should work with any tool that encrypts data at the filesystem or operating system level.

Row-Level Encryption

Pgcrypto is a tool which is included on the source distribution in the /contrib directory. For more information, go to the PostgreSQL documentation at http://www.postgresql.org/docs/current/static/pgcrypto.html.

SSL

If you install PostgreSQL using a package, this option is normally included and enabled. You can check this setting by running the following command on your servers:

```
postgres=# show ssl;
 ssl
-----
 on
```

If you are compiling from source tarball, you need to include the option `--with-openssl` when you run the configurecommand.

The following configuration options are available:

```
#ssl_ciphers = 'ALL:!ADH:!LOW:!EXP:!MD5:@STRENGTH'    # allowed SSL
ciphers
```

```
#ssl_renegotiation_limit = 512MB    # amount of data between
renegotiations
```

On the data directory, you should point to the server certificates, as follows:

```
lrwxrwxrwx 1 root     root        36 Jan 28 16:56 server.crt ->
/etc/ssl/certs/ssl-cert.pem
lrwxrwxrwx 1 root     root        38 Jan 28 16:56 server.key ->
etc/ssl/private/ssl-cert.key
```

Authentication and Network

Although the main authentication configuration file is pg_hba.conf, there is a variable called `listen_addresses` in postgresql.conf. Changing this variable requires a restart. A common practice is to set this variable wide open with a wildcard ('*') and then specify the host in pg_hba.conf.

PostgreSQL uses port 5432 by default. You can have several instances running on the same server on different ports and data directories. We do not recommend this for production servers due to resource consumption.

The best practice in a multi-tier architecture is to permit operational access to the PostgreSQL tier only from servers in the security groups that require access, and control access only from known administrative IP addresses.

The authentication methods supported are: "trust", "reject", "md5", "password", "gss", "sspi", "krb5", "ident", "peer", "pam", "ldap", "radius" or "cert".

We recommend "md5" over "password" as the latter sends the password in the clear.

For storing your PostgreSQL login, you can configure your ~/.pgpass file. For more information, see http://www.postgresql.org/docs/current/static/libpq-pgpass.html.

Conclusion

The AWS cloud provides a unique platform for any RDBMS, including PostgreSQL. With capabilities that can meet dynamic needs, cost based on usage, and easy integration with other AWS products such as Amazon CloudWatch, the AWS cloud enables you to run a variety of applications without having to manage the hardware yourself.