

PostgreSQL HA | Monitoring | Replication Hands-on

UNDER REVIEW

Context

- Introduction
 - What is Postgres
 - What is Grafana and InfluxDB
 - What is HA in PostgreSQL
 - What is Replication in PostgreSQL
 - What Is Automatic Failover in PostgreSQL?
 - High Availability and Failover Replication
 - Why Use PostgreSQL Replication?
 - What Are the Models of PostgreSQL Database Replication (Single-Master & Multi-Master)?
 - What Are the Classes of PostgreSQL Replication?
 - What Are the Replication Modes in PostgreSQL Database?
 - What is Pgwatch2
 - What is Pgwatch2
 - Introduction Grafana and InfluxDB ?
 - What is Grafana
- Architecture
 - PostgreSQL HA Architecture **ADDED**
- Installation
 - Guide for Installing the PostgreSQL & Repmgr
- Monitoring
 - Monitoring **UPDATED**
- Failover test cases
- Incremental backup

Information & Updates /stat

⚠ Monitoring Stack for PostgreSQL yet to add

✅ Loki Logs added

INTRODUCTION

What is PostgreSQL ?

PostgreSQL is an object-relational database management system (ORDBMS) based on POSTGRES, Version 4.2, developed at the University of California at Berkeley Computer Science Department.

What is PostgreSQL HA ?

Master-Slave

This may be the most basic HA architecture we can set up, and oftentimes, the more easy to set and maintain. It is based on one master database with one or more standby servers. These standby databases will remain synchronized (or almost synchronized) with the master, depending on whether the replication is synchronous or asynchronous. If the main server fails, the standby contains almost all of the data of the main server, and can quickly be turned into the new master database server.

We can have two categories of standby databases, based on the nature of the replication

- Logical standbys

The replication between the master and the slaves is made via SQL statements.

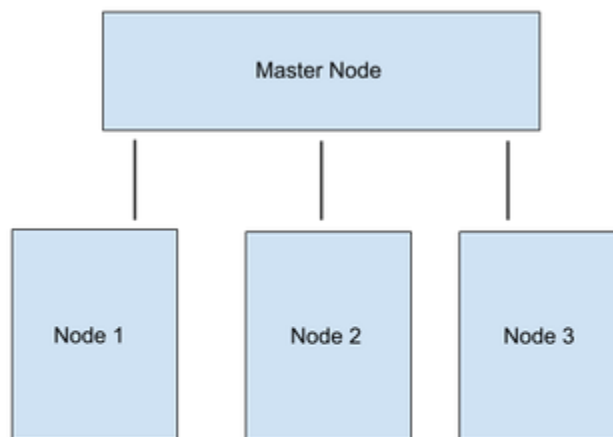
- Physical standbys
The replication between the master and the slaves is made via the internal data structure modifications

In the case of PostgreSQL, a stream of write-ahead log (WAL) records is used to keep the standby databases synchronised. This can be synchronous or asynchronous, and the entire database server is replicated.

From version 10, PostgreSQL includes a built in option to setup logical replication which is based on constructing a stream of logical data modifications from the information in the WAL. This replication method allows the data changes from individual tables to be replicated without the need of designating a master server. It also allows data to flow in multiple directions.

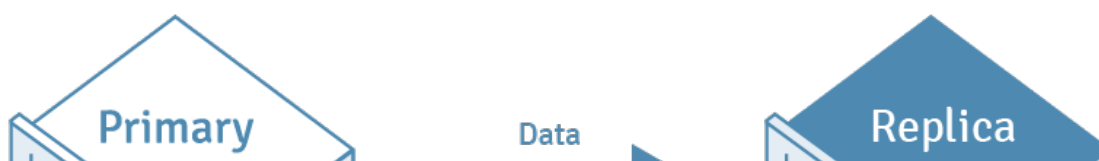
But a master-slave setup is not enough to effectively ensure high availability, as we also need to handle failures. To handle failures, we need to be able to detect them. Once we know there is a failure, e.g. errors on the master or the master is not responding, we can then select a slave and failover to it with the smaller delay possible. It is important that this process is as efficient as possible, in order to restore full functionality so the applications can start functioning again. PostgreSQL itself does not include an automatic failover mechanism, so that will require some custom script or third party tools for this automation.

After a failover happens, the application(s) need to be notified accordingly, so they can start using the new master. Also, we need to evaluate the state of our architecture after a failover, because we can run in a situation where we only have the new master running (i.e., we had a master and only one slave before the issue). In that case, we will need to add a slave somehow so as to re-create the master-slave setup we originally had for HA.



What Is PostgreSQL Replication?

The process of copying data from a PostgreSQL database server to another server is called PostgreSQL Replication. The source database server is usually called the Master server, whereas the database server receiving the copied data is called the Replica server.





What Is Automatic Failover in PostgreSQL?

Once physical streaming replication has been set up and configured in PostgreSQL, failover can take place if the primary server for the database fails. Failover is the term to describe the recovery process, which in PostgreSQL, can take some time, particularly as PostgreSQL itself does not provide built-in tools for detecting server failures. Fortunately, there are tools available that allow for Automatic Failover, which can help detect failures and automatically switch to the standby, minimizing database downtime.

EnterpriseDB's EDB Postgres Failover Manager lets you automatically detect database failures and promotes the most current standby server as the new master, helping to avoid costly database downtime. EDB Failover Manager even provides fast, automatic failure detection.

High Availability and Failover Replication

High Availability refers to database systems that are set up so that standby servers can take over quickly when the master or primary server fails. To achieve high availability, a database system should meet some key requirements: it should have redundancy to prevent single points of failure, reliable switchover mechanisms, and active monitoring to detect any failures that may occur. Setting up failover replication provides the needed redundancy to allow for high availability by ensuring that standbys are available if the master or primary server ever goes down.

Why Use PostgreSQL Replication?

Replication of data can have many uses:

- OLTP Performance
- Fault Tolerance
- Data Migration
- Testing Systems in Parallel

OLTP performance: Removing reporting query load from the online transaction processing (OLTP) system improves both reporting query time and transaction processing performance.

Fault tolerance: In the event of master database server failure, the replica server can take over, since it already contains the master server's data. In this configuration the replica server is also called the standby server. This configuration can also be used for regular maintenance of the primary server.

Data migration: To upgrade database server hardware, or to deploy the same system for another customer.

Testing systems in parallel: When porting the application from one DBMS to another, the results on the same data from both the old and new systems must be compared to ensure that the new system works as expected.

What Are the Models of PostgreSQL Database Replication (Single-Master & Multi-Master)?

- Single-Master Replication (SMR)
- Multi-Master Replication (MMR)

In **Single-Master Replication (SMR)**, changes to table rows in a designated master database server are replicated to one or more replica servers. The replicated tables in the replica database are not permitted to accept any changes (except from the master). But even if they do, changes are not replicated back to the master server.

In **Multi-Master Replication (MMR)**, changes to table rows in more than one designated master database are replicated to their counterpart tables in every other master database. In this model conflict resolution schemes are often employed to avoid problems like duplicate primary keys.

What Are the Classes of PostgreSQL Replication?

- Unidirectional Replication
- Bidirectional Replication

Single-Master Replication is also called unidirectional, since replication data flows in one direction only, from master to replica.

Multi-Master Replication data, on the other hand, flows in both directions, and it is therefore called bidirectional replication.

What Are the Replication Modes in PostgreSQL Database?

- Asynchronous Mode of Replication
- Synchronous Mode of Replication

In **synchronous** mode replication, transactions on the master database are declared complete only when those changes have been replicated to all the replicas. The replica servers must all be available all the time for the transactions to complete on the master.

In **asynchronous** mode, transactions on the master server can be declared complete when the changes have been done on just the master server. These changes are then replicated to the replicas later in time. The replica servers can remain out-of-sync for a certain duration, which is called a replication lag.

Synchronous and asynchronous modes both have their costs and benefits, and users will want to consider safety and performance when configuring their replication settings.

What is Pgwatch2

pgwatch2 is a flexible PostgreSQL-specific monitoring solution, relying on Grafana dashboards for the UI part. It supports monitoring of almost all metrics for Postgres versions 9.0 to 13 out of the box and can be easily extended to include custom metrics. At the core of the solution is the metrics gathering daemon written in Go, with many options to configure the details and aggressiveness of monitoring, types of metrics storage and the display the metrics.

What is Grafana?

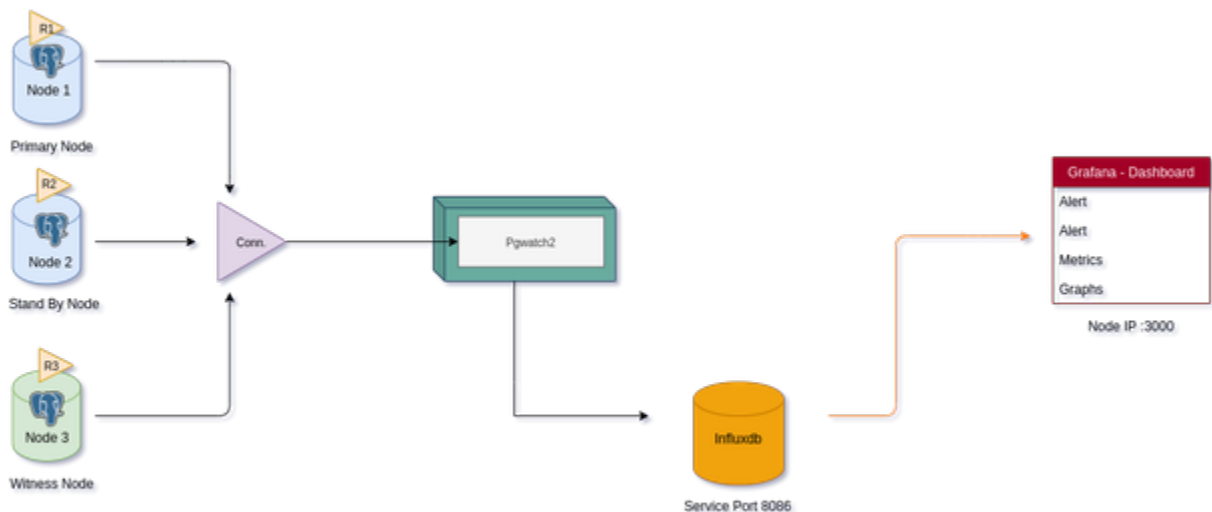
Grafana is open source visualization and analytics software. It allows you to query, visualize, alert on, and explore your metrics no matter where they are stored. In plain English, it provides you with tools to turn your time-series database (TSDB) data into beautiful graphs and visualizations.

For example, if you want to view weather data and statistics about your smart home, then you might create a playlist. If you are the administrator for a corporation and are managing Grafana for multiple teams, then you might need to set up provisioning and authentication.

InfluxDB

Architecture

Postgres HA & Repmgr with Grafana Dashboard



Notations

- Connection between Node - X using the SSH Key Exchange



- Repmgr - X



OSTGRESQL HA CLUSTER WITH REPLICATION MANAGER

Introduction

The Purpose of this document is to propose a high availability PostgreSQL solution with 2ndQuardant's Replication Manager. 2ndQuardant is a leading specialist in PostgreSQL related technologies and services. The product is used to automate, enhance, and manage PostgreSQL streaming replication. Streaming replication in PostgreSQL has been around since version 9.0. Natively setting up and managing streaming replication involves several manual steps which includes:

- Configuring replication parameters in both primary and each standby node
- Backing up primary node data with pg_basebackup from each standby node and restoring it there
- Restarting the standby node(s)

With repmgr most of these tasks can be automated, saving the DBA and operational staff both time and effort. A repmgr will also prevent the split-brain scenario in cluster.

repmgr --> repmgr is an open-source tool. this The tool is used to automate, enhance, and manage PostgreSQL streaming replication.

Throughout this Doc, we will keep referring to each these nodes, so here's a table showing their details:

Node Name	IP Address	Role	Apps
Running			
PG-Node1	10.160.0.17	Primary	PostgreSQL
12 and repmgr			
PG-Node2	10.160.0.17	Standby	
PostgreSQL 12 and repmgr			
PG-Node3	10.160.0.19	Witness	
PostgreSQL 12 and repmgr			

steps-----

1. In each node, we are running the following commands. The first command adds the PostgreSQL Global Development Group (PGDG) repo, the second one disables RHEL's built-in PostgreSQL module, and finally, the third one installs PostgreSQL 12 from the PGDG repo.

```
# dnf -y install https://download.postgresql.org/pub/repos/yum
/reporpms/EL-8-x86_64/pgdg-redhat-repo-latest.noarch.rpm
```

```
# dnf -qy module disable postgresql
```

```
# dnf install postgresql12-server postgresql12-contrib
```

2. Since PG-Node2 and PG-Node3 are standby nodes, we are leaving them and running initdb in the primary (PG-Node1) and witness (PG-Node-Witness) node only:

```
# /usr/pgsql-12/bin/postgresql-12-setup initdb
```

3. We then make the following change in the witness node's postgresql.conf file:

```
/var/lib/pgsql/12/data
```

```
listen_addresses = '*'
```

4. We also add the following lines at the end of the postgresql.conf file in the primary node:

```
/var/lib/pgsql/12/data
```

```
listen_addresses = '*'
```

```
max_wal_senders = 10
```

```
max_replication_slots = 10
```

```
wal_level = 'replica'
```

```
hot_standby = on
```

```
archive_mode = on
```

```
archive_command = '/bin/true'
```

5. With the changes made, we start PostgreSQL 12 service in both the primary and the witness node and enable the services.

```
# systemctl start postgresql-12.service
```

```
# systemctl enable postgresql-12.service
```

6. Next, we install the repmgr repo definition in the primary node (PG-Node1) and standby node (PG-Node2)

```
# wget https://download.postgresql.org/pub/repos/yum/reporpms/EL-8-
x86_64/pgdg-redhat-repo-latest.noarch.rpm
```

6.1 And then installing the repmgr package:

```
# yum install repmgr12 -y
```

7. We are running the following commands in the primary node only. Note that we have switched to the postgres user from the shell prompt before running these commands.

7.1 The first command creates the repmgr user:

```
[postgres@PG-Node1 ~]$ createuser --superuser repmgr
```

7.2 The second one creates the repmgr database, with its owner as repmgr:

```
[postgres@PG-Node1 ~]$ createdb --owner=repmgr repmgr
```

7.3 Finally, we change the repmgr user's default search path:

```
[postgres@PG-Node1 ~]$ psql -c "ALTER USER repmgr SET search_path TO repmgr, public;"
```

8. With the database and the user created, we add the following line in the primary node's postgresql.conf file:

```
NOTE -> location of postgresql.conf = (/var/lib/pgsql/12/data)
```

```
shared_preload_libraries = 'repmgr'
```

NOTE -> This will load the repmgr extension when PostgreSQL starts. By default any PostgreSQL configuration files present in the data directory will be copied when cloning a standby, so any settings configured for the primary will be copied to the standby as well.

9. We are configuring repmgr in the primary and the standby nodes below:

For PostgreSQL 12, the default location of the repmgr configuration file is /etc/repmgr/12/ and it is called repmgr.conf.

--> For primary node (PG-Node1):

```
node_id=1
node_name='PG-Node1'
conninfo='host=[ip of node 1] user=repmgr dbname=repmgr
connect_timeout=2'
data_directory='/var/lib/pgsql/12/data'
```

```
--> For standby node 1 (PG-Node2):
```

```
node_id=2
node_name='PG-Node2'
conninfo='host=[ip of node 2] user=repmgr dbname=repmgr
connect_timeout=2'
data_directory='/var/lib/pgsql/12/data'
```

10. Next, we add the following lines in the `pg_hba.conf` file in PG-Node1. As we will see later, the `pg_hba.conf` file from the primary node will be copied to the two standbys when `repmgr` sets up replication.

Note how we are using the CIDR range of the cluster instead of specifying individual IP addresses.

```
# IPv4 local connections:

trust    host    replication    all    10.160.0.17/32
trust    host    replication    all    10.160.0.18/32
trust    host    replication    all    10.160.0.19/32

# Allow replication connections from localhost, by a user with the
# replication privilege.

trust    local    replication    all
trust    host    replication    all    127.0.0.1/32
trust    host    replication    all    ::1/128
ident    host    all            all    10.160.0.17/32
trust    host    all            all    10.160.0.18/32
```

11. Once the configs are completed, we restart the PostgreSQL service in the primary node:

```
# systemctl restart postgresql-12.service
```

12. To test if the standby nodes can connect to the primary node, we are running the following command from PG-Node2

```
# psql 'host=[ip of primary] user=repmgr dbname=repmgr'
```



```
connect_timeout=2'
```

12.1 In our test case, the connectivity is working as we see the primary node's PostgreSQL prompt.at the repmgr database:

```
# repmgr>
```

13. We then run the following command in the primary node (PG-Node1) as the postgres user. This registers the primary node PostgreSQL instance with repmgr. This command installs the repmgr extension. It also adds metadata about the primary node in the repmgr database.

```
[postgres@PG-Node1 ~]$ /usr/pgsql-12/bin/repmgr -f /etc/repmgr/12/repmgr.conf primary register
```

The output looks like this:

```
INFO: connecting to primary database...
NOTICE: attempting to install extension "repmgr"
NOTICE: "repmgr" extension successfully installed
NOTICE: primary node record (ID: 1) registered
```

13.1 We can now quickly check the status of our cluster:

```
[postgres@PG-NodeX ~]$ /usr/pgsql-12/bin/repmgr -f /etc/repmgr/12/repmgr.conf cluster show
```

As expected, our cluster has only one node - the primary:

ID	Name	Role	Status	Upstream	Connection string
1	PG-Node1	primary	* running		host=PG-Node1 dbname=repmgr user=repmgr connect_timeout=2

14. Next, we will run the following command in the standby node PG-Node2 as the postgres user for a dry-run before actually cloning from the primary:

```
[postgres@PG-NodeX ~]$ /usr/pgsql-12/bin/repmgr -h [primary node ip] -U repmgr -d repmgr -f /etc/repmgr/12/repmgr.conf standby clone --dry-run
```

If the output is like the following, it tells us the cloning will succeed:

```
NOTICE: using provided configuration file "/etc/repmgr.conf"
```

```
destination directory "/var/lib/pgsql/12/data" provided
INFO: connecting to source node
NOTICE: checking for available walsenders on source node (2
required)
INFO: sufficient walsenders available on source node (2
required)
NOTICE: standby will attach to upstream node 1
HINT: consider using the -c/--fast-checkpoint option
INFO: all prerequisites for "standby clone" are met
```

15. the standby nodes show all prerequisites for standby clone are met, we can go ahead with the clone operation:

```
/usr/pgsql-12/bin/repmgr -h [primary ip] -U repmgr -d repmgr -f
/etc/repmgr/12/repmgr.conf standby clone
```

A successful clone operation shows a series of messages like this:

```
NOTICE: destination directory "/var/lib/pgsql/12/data"
provided
INFO: connecting to source node
DETAIL: connection string is: host=16.0.xxx.xxx6
user=repmgr dbname=repmgr
DETAIL: current installation size is 66 MB
NOTICE: checking for available walsenders on the source
node (2 required)
NOTICE: checking replication connections can be made to the
source server (2 required)
INFO: checking and correcting permissions on existing
directory "/var/lib/pgsql/12/data"
NOTICE: starting backup (using pg_basebackup)...
HINT: this may take some time; consider using the -c/--fast-
checkpoint option
INFO: executing:
/usr/pgsql-12/bin/pg_basebackup -l "repmgr base backup" -D
/var/lib/pgsql/12/data -h 16.0.xxx.xxx -p 5432 -U repmgr -X stream
NOTICE: standby clone (using pg_basebackup) complete
NOTICE: you can now start your PostgreSQL server
HINT: for example: pg_ctl -D /var/lib/pgsql/12/data start
HINT: after starting the server, you need to register this
standby with "repmgr standby register"
```

NOTE--> At this stage, PostgreSQL isn't running in the standby node, although the nodes have their postgres data directory copied from the primary.

16. We start the postgresql service in the nodes and enable the service:

```
# systemctl start postgresql-12.service
# systemctl enable postgresql-12.service
```

17. We then run the following command in standby node as the postgres user to register it with repmgr:

```
[postgres@PG-NodeX ~]$ /usr/pgsql-12/bin/repmgr -f /etc/repmgr/12
/repmgr.conf standby register
```

A successful registration shows a series of messages like the following:

```
INFO: connecting to local node "PG-NodeX" (ID: 3)
INFO: connecting to primary database
WARNING: --upstream-node-id not supplied, assuming upstream
node is primary (node ID 1)
INFO: standby registration complete
NOTICE: standby node "PG-NodeX" (ID: X) successfully
registered
```

18. we run the following repmgr command from any of the nodes as the postgres user:

ID	Name	Role	Status
Upstream	Location	Prio.	TLI
-----+-----+-----+-----			
+-----+-----+-----			
1	PG-Node1	primary	* running
default	100	1	
2	PG-Node2	standby	* running
default	100	2	

HOW TO AUTOMATE POSTGRESQL
REPLICATION AND FAILOVER

NOTE --> Setting up streaming replication with repmgr is very simple. What we need to do next is to ensure the cluster will function even when the primary becomes unavailable.

In PostgreSQL replication, a primary can become unavailable for a few reasons. For example:

1. The operating system of the primary node can crash, or become unresponsive
2. The primary node can lose its network connectivity
3. The PostgreSQL service in the primary node can crash, stop, or become unavailable unexpectedly
4. The PostgreSQL service in the primary node can be stopped intentionally or accidentally

Whenever a primary becomes unavailable, a standby does not automatically promote itself to the primary role. A standby still continues to serve read-only queries - although the data will be current up to the last LSN received from the primary. Any attempt for a write operation will fail.

There are two ways to mitigate this:

1. The standby is manually upgraded to a primary role. This is usually the case for a planned failover or "switchover"
2. The standby is automatically promoted to a primary role. This is the case with non-native tools that continuously monitor replication and take recovery action when the primary is unavailable. repmgr is one such tool.

We will consider the second scenario here. This situation has some extra challenges though:

When the primary is unavailable - it is the witness node's job to help the standbys reach a quorum if one of them should be promoted to a primary role. The standbys reach this quorum by determining if the primary node is actually offline or only temporarily unavailable. The witness node should be located in the same data centre/network segment /subnet as the primary node, but must NEVER run on the same physical host as the primary node.

The second component of the solution is the repmgr daemon (repmgrd) running in all nodes of the cluster and the witness node.

The daemon comes as part of the repmgr package - when enabled, it runs as a regular service and continuously monitors the cluster's health. It initiates a failover when a quorum is reached about the primary being offline. Not only can it automatically promote a standby, it can also reinitiate other standbys in a multi-node cluster to follow the new

primary.

The Quorum Process -->

When a standby realizes it cannot see the primary, it consults with other standbys. All the standbys running in the cluster reach a quorum to choose a new primary using a series of checks:

- * Each standby interrogates other standbys about the time it last "saw" the primary. If a standby's last replicated LSN or the time of last communication with the primary is more recent than the current node's last replicated LSN or the time of last communication, the node does not do anything and waits for the communication with the primary to be restored

- * If none of the standbys can see the primary, they check if the witness node is available. If the witness node cannot be reached either, the standbys assume there is a network outage on the primary side and do not proceed to choose a new primary

- * If the witness can be reached, the standbys assume the primary is down and proceed to choose a primary

- * The node that was configured as the "preferred" primary will then be promoted. Each standby will have its replication reinitialized to follow the new primary.

We will now configure the cluster and the witness node for automatic failover.

steps

1. Install and Configure repmgr in Witness

```
# wget https://download.postgresql.org/pub/repos/yum/reporpms/EL-8-x86_64/pgdg-redhat-repo-latest.noarch.rpm
```

And then:

```
# yum install repmgr12 -y
```

2. Next, we add the following lines in the witness node's postgresql.conf file:

```
/var/lib/pgsql/12/data
```

```
listen_addresses = '*'
shared_preload_libraries = 'repmgr'
```

3. We also add the following lines in the `pg_hba.conf` file in the witness node. Note how we are using the CIDR range of the cluster instead of specifying individual IP addresses.

```
# IPv4 local connections:

host      all             all             127.0.0.1/32          trust
host      replication     all             10.160.0.17/32        trust
host      replication     all             10.160.0.18/32        trust
host      replication     all             10.160.0.19/32        trust

# replication privilege.

local     replication     all             trust
host      replication     all             127.0.0.1/32          trust
host      replication     all             ::1/128               trust
host      repmgr          all             10.160.0.17/32        trust
host      repmgr          all             10.160.0.18/32        trust
host      repmgr          all             10.160.0.19/32        trust
```

4. With `postgresql.conf` and `pg_hba.conf` changes done, we create the `repmgr` user and the `repmgr` database in the witness, and change the `repmgr` user's default search path:

```
[postgres@PG-Node-Witness ~]$ createuser --superuser repmgr

[postgres@PG-Node-Witness ~]$ createdb --owner=repmgr repmgr

[postgres@PG-Node-Witness ~]$ psql -c "ALTER USER repmgr SET
search_path TO repmgr, public;"
```

5. Finally, we add the following lines to the `repmgr.conf` file, located under `/etc/repmgr/12/`

```
node_id=4
node_name='PG-Node-Witness'
conninfo='host=16.0.1.37 user=repmgr dbname=repmgr
connect_timeout=2'
data_directory='/var/lib/pgsql/12/data'
```

6. Once the config parameters are set, we restart the PostgreSQL

service in the witness node:

```
# systemctl restart postgresql-12.service
```

7. To test the connectivity to witness node repmgr, we can run this command from the primary node:

```
[postgres@PG-Node1 ~]$ psql 'host=16.0.1.37 user=repmgr
dbname=repmgr connect_timeout=2'
```

8. Next, we register the witness node with repmgr by running the "repmgr witness register" command as the postgres user. Note how we are using the address of the primary node, and NOT the witness node in the command below:

```
[postgres@PG-Node-Witness ~]$ /usr/pgsql-12/bin/repmgr -f /etc
/repmgr/12/repmgr.conf witness register -h [primary node ip]
```

```
ex: [postgres@PG-Node-Witness ~]$ /usr/pgsql-12/bin/repmgr -f /etc
/repmgr/12/repmgr.conf witness register -h 16.0.1.156
```

8.1 This is because the "repmgr witness register" command adds the witness node's metadata to primary node's repmgr database, and if necessary, initialises the witness node by installing the repmgr extension and copying the repmgr metadata to the witness node.

The output will look like this:

```
INFO: connecting to witness node "PG-Node-Witness" (ID: 4)
INFO: connecting to primary node
NOTICE: attempting to install extension "repmgr"
NOTICE: "repmgr" extension successfully installed
INFO: witness registration complete
NOTICE: witness node "PG-Node-Witness" (ID: 4) successfully
registered
```

9. Finally, we check the status of the overall setup from any node:

```
[postgres@PG-Node-Witness ~]$ /usr/pgsql-12/bin/repmgr -f /etc
/repmgr/12/repmgr.conf cluster show --compact
```

The output looks like this:

ID	Name	Role	Status	Upstream
	Location	Prio.	TLI	

```

-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
      1 | PG-Node1          | primary | * running          |
| default | 100      | 1
      2 | PG-Node2          | standby | * running          |
| default | 100      | 2
      3 | PG-Node-Witness    | witness | * running          | ! PG-Node2
| default | 0        | n/a

```

10. Modifying the sudoers File

With the cluster and the witness running, we add the following lines in the sudoers file In each node of the cluster and the witness node:

```

* Defaults:postgres !requiretty

* postgres ALL = NOPASSWD: /usr/bin/systemctl stop postgresql-12.
service, /usr/ bin/systemctl start postgresql-12.service, /usr/bin
/systemctl restart postgresql-12.service, /usr/bin/systemctl reload
postgresql-12.service, /usr/bin/systemctl start repmgr12.service, /usr
/bin/systemctl stop repmgr12.service

```

11. Configuring repmgrd Parameters

We have already added four parameters in the repmgr.conf file in each node. The parameters added are the basic ones needed for repmgr operation. To enable the repmgr daemon and automatic failover, a number of other parameters need to be enabled/added. In the following subsections, we will describe each parameter and the value they will be set to in each node.

11.1 failover -->

The failover parameter is one of the mandatory parameters for the repmgr daemon. This parameter tells the daemon if it should initiate an automatic failover when a failover situation is detected. It can have either of two values: "manual" or "automatic". NOTE [We will set this to automatic in each node:]

```
/etc/repmgr/12/repmgr.conf --> repmgr.conf
```

```
failover='automatic'
```

11.2 promote_command -->

This is another mandatory parameter for the repmgr daemon. This parameter tells the repmgr daemon what command it should run to promote a standby. The value of this parameter will be typically the "repmgr standby promote" command, or the path to a shell script that calls the command. For our use case, NOTE [we set this to the following in each node:]

```
/etc/repmgr/12/repmgr.conf --> repmgr.conf
```

```
promote_command='/usr/pgsql-12/bin/repmgr standby promote -f /etc/repmgr/12/repmgr.conf --log-to-file'
```

11.3 follow_command -->

This is the third mandatory parameter for the repmgr daemon. This parameter tells a standby node to follow the new primary. The repmgr daemon replaces the %n placeholder with the node ID of the new primary at run time:

```
/etc/repmgr/12/repmgr.conf --> repmgr.conf
```

```
follow_command='/usr/pgsql-12/bin/repmgr standby follow -f /etc/repmgr/12/repmgr.conf --log-to-file --upstream-node-id=%n'
```

11.4 priority -->

The priority parameter adds weight to a node's eligibility to become a primary. Setting this parameter to a higher value gives a node greater eligibility to become the primary node. Also, setting this value to zero for a node will ensure the node is never promoted as primary.

NOTE --> In our case, we have one standby: PG-NODE2. We want to promote PG-NODE2 as the new primary. We set the parameter to the following value in the two standby node:

```
/etc/repmgr/12/repmgr.conf --> repmgr.conf
```

```
PG-Node2          priority=60
```

11.5 monitor_interval_secs -->

This parameter tells the repmgr daemon how often (in number of seconds) it should check the availability of the upstream node. In our case, there is only one upstream node: the primary node. The default value is 2 seconds, but we will explicitly set this anyhow in each node:

```
/etc/repmgr/12/repmgr.conf --> repmgr.conf
```

```
monitor_interval_secs=2
```

11.6 connection_check_type -->

The `connection_check_type` parameter dictates the protocol repmgr daemon will use to reach out to the upstream node. This parameter can take three values:

- * `ping`: repmgr uses the `PQPing()` method

- * `connection`: repmgr tries to create a new connection to the upstream node

- * `query`: repmgr tries to run a SQL query on the upstream node using the existing connection

11.7 Again, we will set this parameter to the default value of `ping` in each node:

```
/etc/repmgr/12/repmgr.conf --> repmgr.conf
```

```
connection_check_type='ping'
```

11.8 reconnect_attempts and reconnect_interval -->

When the primary becomes unavailable, the repmgr daemon in the standby nodes will try to reconnect to the primary for `reconnect_attempts` times. The default value for this parameter is 6. Between each reconnect attempt, it will wait for `reconnect_interval` seconds, which has a default value of 10. For demonstration purposes, we will use a short interval and fewer reconnect attempts. NOTE [We set this parameter in every node:]

```
/etc/repmgr/12/repmgr.conf --> repmgr.conf
```

```
reconnect_attempts=4
```

```
reconnect_interval=8
```

11.9 primary_visibility_consensus -->

When the primary becomes unavailable in a multi-node cluster, the standbys can consult each other to build a quorum about a failover. This is done by asking each standby about the time it last saw the

primary. If a node's last communication was very recent and later than the time the local node saw the primary, the local node assumes the primary is still available, and does not go ahead with a failover decision.

NOTE --> o enable this consensus model, the `primary_visibility_consensus` parameter needs to be set to "true" in each node - including the witness:

```
/etc/repmgr/12/repmgr.conf --> repmgr.conf
```

```
primary_visibility_consensus=true
```

11.10 `standby_disconnect_on_failover` -->

When the `standby_disconnect_on_failover` parameter is set to "true" in a standby node, the `repmgr` daemon will ensure its WAL receiver is disconnected from the primary and not receiving any WAL segments. It will also wait for the WAL receivers of other standby nodes to stop before making a failover decision. This parameter should be set to the same value in each node. We are setting this to "true".

```
/etc/repmgr/12/repmgr.conf --> repmgr.conf
```

```
standby_disconnect_on_failover=true
```

Setting this parameter to true means every standby node has stopped receiving data from the primary as the failover happens. The process will have a delay of 5 seconds plus the time it takes the WAL receiver to stop before a failover decision is made. By default, the `repmgr` daemon will wait for 30 seconds to confirm all sibling nodes have stopped receiving WAL segments before the failover happens.

11.11 `repmgrd_service_start_command` and `repmgrd_service_stop_command`

These two parameters specify how to start and stop the `repmgr` daemon using the "repmgr daemon start" and "repmgr daemon stop" commands.

Basically, these two commands are wrappers around operating system commands for starting/stopping the service. The two parameter values map these commands to their OS-specific versions. We set these parameters to the following values in each node:

```
repmgrd_service_start_command='sudo /usr/bin/systemctl  
start repmgr12.service'
```

```
repmgrd_service_stop_command='sudo /usr/bin/systemctl stop  
repmgr12.service'
```

11.12 PostgreSQL Service Start/Stop/Restart Commands --> As part of its operation, the repmgr daemon will often need to stop, start or restart the PostgreSQL service. To ensure this happens smoothly, it is best to specify the corresponding operating system commands as parameter values in the repmgr.conf file. We will set four parameters in each node for this purpose:

```
service_start_command='sudo /usr/bin/systemctl start
postgresql-12.service'
service_stop_command='sudo /usr/bin/systemctl stop
postgresql-12.service'
service_restart_command='sudo /usr/bin/systemctl restart
postgresql-12.service'
service_reload_command='sudo /usr/bin/systemctl reload
postgresql-12.service'
```

11.13 monitoring_history --> Setting the monitoring_history parameter to "yes" will ensure repmgr is saving its cluster monitoring data. We set this to "yes" in each node:

```
monitoring_history=yes
```

11.14 log_status_interval -->

We set the parameter in each node to specify how often the repmgr daemon will log a status message. In this case, we are setting this to every 60 seconds:

```
log_status_interval=60
```

12. Starting the repmgr Daemon > With the parameters now set in the cluster and the witness node, we execute a dry run of the command to start the repmgr daemon. We test this in the primary node first, and then the two standby nodes, followed by the witness node. The command has to be executed as the postgres user:

```
[postgres@PG-NodeX ~]$ /usr/pgsql-12/bin/repmgr -f /etc/repmgr/12
/repmgr.conf daemon start --dry-run
```

The output should look like this:

```
INFO: prerequisites for starting repmgrd met
DETAIL: following command would be executed:
      sudo /usr/bin/systemctl start repmgr12.service
```

13. Next, we start the daemon in all four nodes:

```
[postgres@PG-NodeX ~]$ /usr/pgsql-12/bin/repmgr -f /etc/repmgr/12
/repmgr.conf daemon start
```

The output in each node should show the daemon has started:

```
NOTICE: executing: "sudo /usr/bin/systemctl start repmgr12.
service"
NOTICE: repmgrd was successfully started
```

14. We can also check the service startup event from the primary or standby nodes:

```
[postgres@PG-NodeX ~]$ /usr/pgsql-12/bin/repmgr -f /etc/repmgr/12
/repmgr.conf cluster event --event=repmgrd_start
```

The output should show the daemon is monitoring the connections:

Node ID	Name	Event	OK
Timestamp	Details		
3	PG-Node-Witness	repmgrd_start	t
witness monitoring connection to primary node "PG-Node1" (ID: 1)			
2	PG-Node2	repmgrd_start	t
monitoring connection to upstream node "PG-Node1" (ID: 1)			
1	PG-Node1	repmgrd_start	t
monitoring cluster primary "PG-Node1" (ID: 1)			

15. Finally, we can check the daemon output from the syslog in the standby:

```
# cat /var/log/messages | grep repmgr | less
```

Here is the output from PG-Node2:

```
[NOTICE] starting monitoring of node "PG-Node3" (ID: 3)
[INFO] "connection_check_type" set to "ping"
[INFO] monitoring connection to upstream node "PG-Node1" (ID: 1)
[INFO] node "PG-Node2" (ID: 2) monitoring upstream node "PG-Node1"
(ID: 1) in normal state
[DETAIL] last monitoring statistics update was 2 seconds ago
[INFO] node "PG-Node2" (ID: 2) monitoring upstream node "PG-Node1"
(ID: 1) in normal state
```

16. Checking the syslog in the primary node shows a different type of output:

```
[NOTICE] using provided configuration file "/etc/repmgr/12/repmgr.conf"
[NOTICE] repmgrd (repmgrd 5.0.0) starting up
[INFO] connecting to database "host=16.0.1.156 user=repmgr dbname=repmgr connect_timeout=2"
[NOTICE] starting monitoring of node "PG-Node1" (ID: 1)
[INFO] "connection_check_type" set to "ping"
[NOTICE] monitoring cluster primary "PG-Node1" (ID: 1)
[INFO] child node "PG-Node-Witness" (ID: 4) is not yet attached
[INFO] child node "PG-Node3" (ID: 3) is attached
[INFO] child node "PG-Node2" (ID: 2) is attached
[NOTICE] new witness "PG-Node-Witness" (ID: 4) has connected
[INFO] monitoring primary node "PG-Node1" (ID: 1) in normal state
[INFO] monitoring primary node "PG-Node1" (ID: 1) in normal state
```

17. Simulating a Failed Primary > Now we will simulate a failed primary by stopping the primary node (PG-Node1). From the shell prompt of the node, we run the following command:

```
# systemctl stop postgresql-12.service
```

18. The Failover Process > Once the process stops, we wait for about a minute or two, and then check the syslog file of PG-Node2. The following messages are shown.

```
[INFO] switching to primary monitoring mode
[NOTICE] monitoring cluster primary "PG-Node2" (ID: 2)
[NOTICE] new witness "PG-Node-Witness" (ID: 3) has connected
[INFO] monitoring primary node "PG-Node2" (ID: 2) in normal state
[INFO] monitoring primary node "PG-Node2" (ID: 2) in normal state
```

19. Our cluster is now back in action. We can confirm this by running the "repmgr cluster show" command:

```
[postgres@PG-Node-Witness ~]$ /usr/pgsql-12/bin/repmgr -f /etc/repmgr/12/repmgr.conf cluster show --compact
```

ID	Name	Role	Status	Upstream
Location	Prio.	TLI		
1	PG-Node1	primary	! running	

```

default | 100 | 1
        2 | PG-Node2 | primary | * running |
default | 100 | 2
        3 | PG-Node-Witness | witness | * running | PG-Node2 |
default | 0 | n/a

```

20. We can also look for the events by running the "repmgr cluster event" command:

```

[postgres@PG-Node2 ~]$ /usr/pgsql-12/bin/repmgr -f /etc/repmgr/12
/repmgr.conf cluster event

```

The output displays how it happened:

```

Node ID | Name | Event
| OK | Timestamp | Details
-----+-----+-----
+---+-----+
+-----+
-----
      2 | PG-Node2 | repmgrd_shutdown | t |
2021-08-30 12:48:47 | TERM signal received
      3 | PG-Node-Witness | repmgrd_shutdown | t |
2021-08-30 12:48:47 | TERM signal received
      2 | PG-Node2 | repmgrd_local_reconnect | t |
2021-08-30 11:29:28 | reconnected to primary node after 96 seconds,
resuming monitoring
      3 | PG-Node-Witness | repmgrd_upstream_reconnect | t |
2021-08-30 11:29:27 | reconnected to upstream node "PG-Node2" (ID: 2)
after 10 seconds, resuming monitoring
      2 | PG-Node2 | child_node_new_connect | t |
2021-08-30 11:14:37 | new witness "PG-Node-Witness" (ID: 3) has
connected
      3 | PG-Node-Witness | repmgrd_upstream_reconnect | t |
2021-08-30 11:14:31 | witness monitoring connection to primary node "PG-
Node2" (ID: 2)
      3 | PG-Node-Witness | repmgrd_failover_follow | t |
2021-08-30 11:14:31 | witness node "PG-Node-Witness" (ID: 3) now
following new primary node "PG-Node2" (ID: 2)
      2 | PG-Node2 | repmgrd_reload | t |
2021-08-30 11:14:06 | monitoring cluster primary "PG-Node2" (ID: 2)
      2 | PG-Node2 | repmgrd_failover_promote | t |
2021-08-30 11:14:06 | node "PG-Node2" (ID: 2) promoted to primary; old
primary "PG-Node1" (ID: 1) marked as failed
      2 | PG-Node2 | standby_promote | t |
2021-08-30 11:14:06 | server "PG-Node2" (ID: 2) was successfully
promoted to primary
      1 | PG-Node1 | child_node_new_connect | t |
2021-08-30 10:48:15 | new witness "PG-Node-Witness" (ID: 3) has

```

```

connected
      3      | PG-Node-Witness | repmgrd_start | t |
2021-08-30 10:48:09 | witness monitoring connection to primary node "PG-
Node1" (ID: 1)
      2      | PG-Node2      | repmgrd_start | t |
2021-08-30 10:48:06 | monitoring connection to upstream node "PG-Node1"
(ID: 1)
      1      | PG-Node1      | repmgrd_start | t |
2021-08-30 10:48:03 | monitoring cluster primary "PG-Node1" (ID: 1)
      3      | PG-Node-Witness | witness_register | t |
2021-08-28 19:31:35 | witness registration succeeded; upstream node ID
is 1
      2      | PG-Node2      | standby_register | t |
2021-08-28 19:07:15 | standby registration succeeded; upstream node ID
is 1
      2      | PG-Node2      | standby_clone | t |
2021-08-28 19:04:35 | cloned from host "10.160.0.17", port 5432; backup
method: pg_basebackup; --force: N
      1      | PG-Node1      | primary_register | t |
2021-08-28 18:57:00 |
      1      | PG-Node1      | cluster_created | t |
2021-08-28 18:57:00 |

```

END

Logs Aggregation Tool Loki with Grafana

WORKING

Prerequisites

- Loki Binary
- Promtail
- Install Wget & unzip

Steps for installation

1. Download the prerequisites
 - a. Download Loki binary from GitHub

```
curl -O -L "https://github.com/grafana/loki/releases/download/v2.3.0/loki-linux-amd64.zip"
```

- b. Download the Promtail from GitHub


```
wget https://github.com/grafana/loki/releases/download/v2.3.0/promtail-  
linux-amd64.zip
```

c. Unzip the packages & give them permission for execution

```
unzip "loki-linux-amd64.zip"  
chmod a+x "loki-linux-amd64"  
unzip promtail-linux-amd64.zip
```

2. Configuration

a. Loki Configuration

```
vim config.yaml
```

```
server:  
  http_listen_port: 9080  
  grpc_listen_port: 0  
  
positions:  
  filename: /tmp/positions.yaml  
  
client:  
  url: http://localhost:3100/api/prom/push  
  
scrape_configs:  
- job_name: system  
  static_configs:  
  - targets:  
    - localhost  
  labels:  
    job: varlogs  
    __path__: /var/lib/pgsql/12/data/log/*log
```

b. Promtail Configuration

```
vim promtail-local-config.yaml
```

```
server:  
  http_listen_port: 9080  
  grpc_listen_port: 0
```

```

positions:
  filename: /tmp/positions.yaml

clients:
  - url: http://localhost:3100/loki/api/v1/push

scrape_configs:
  - job_name: system
    static_configs:
      - targets:
          - localhost
        labels:
          job: varlogs
          __path__: /var/lib/pgsql/12/data/log/*log

```

Run the Promtail & Loki using the Binary

a. Run Loki binary

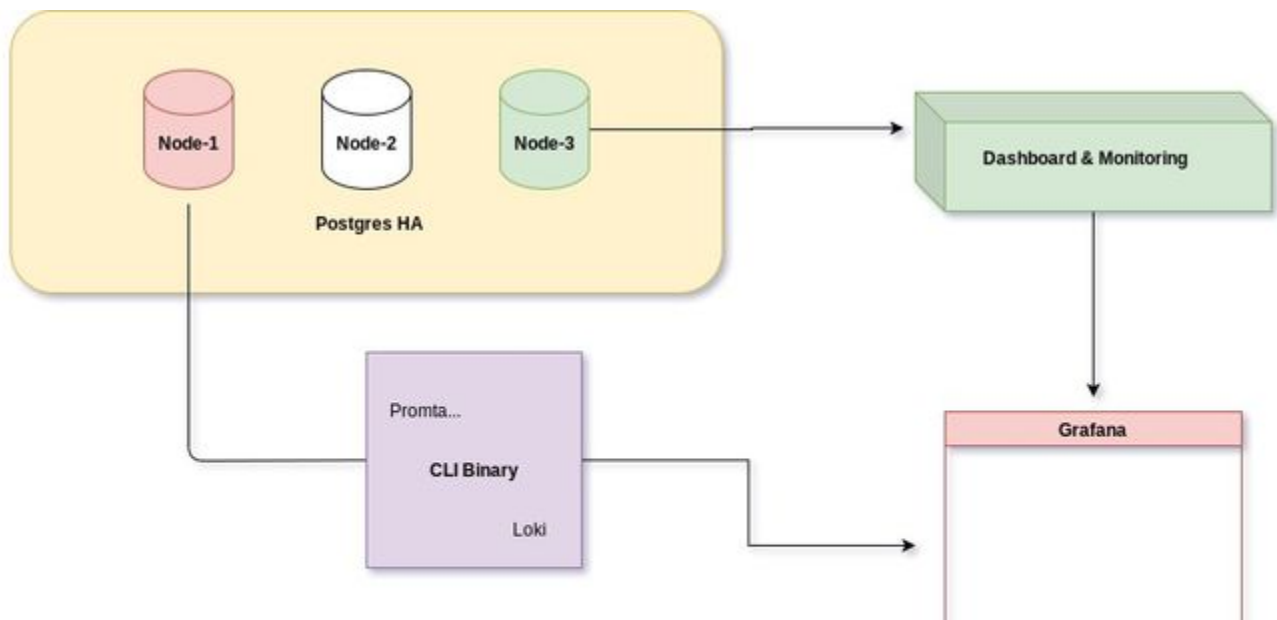
```
./loki-linux-amd64 -config.file=./config.yaml
```

b. Run the promtail binary

```
./promtail-linux-amd64 -config.file=./promtail-local-config.yaml
```

i Make sure promtail & Loki working fine

Architecture for Monitoring





Connect Loki With Grafana Dashboard

Grafana is already installed in Node-3 as a container image now we need to connect with the Promtail and Loki

Video is here

1. Connect the Grafana And the data source

a.

The screenshot shows the 'Data Sources / Loki' configuration page in Grafana. The page has a dark theme. At the top, there's a header with the Grafana logo and the title 'Data Sources / Loki' with 'Type: Loki' below it. A 'Settings' tab is selected. Below the header, there's a 'Name' field set to 'Loki' and a 'Default' toggle switch. The 'HTTP' section contains a 'URL' field set to 'http://10.160.0.17:3100' and a 'Whitelisted Cookies' field with an 'Add' button. The 'Auth' section has several toggle switches: 'Basic auth' (off), 'With Credentials' (off), 'TLS Client Auth' (off), 'With CA Cert' (off), 'Skip TLS Verify' (off), and 'Forward OAuth Identity' (off). At the bottom, there's a 'Custom HTTP Headers' section with an 'Add Header' button.

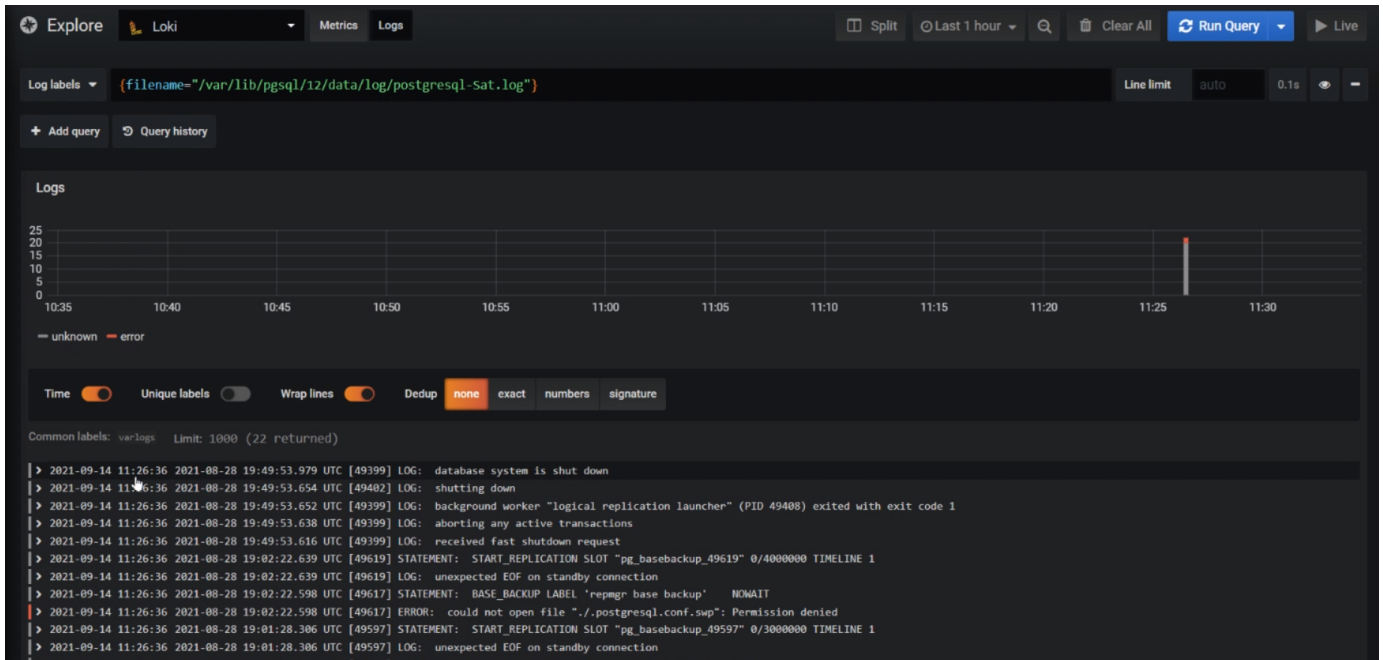
2. Lets Select the Logs

b. Loki added to the grafana there must be Dashboard Update

This screenshot shows the same 'Data Sources / Loki' configuration page, but with additional success messages. A green notification bar at the top right says '✓ Datasource updated'. Another green notification bar at the bottom says '✓ Data source connected and labels found.' The 'Custom HTTP Headers' section is visible with the 'Add Header' button. At the bottom, there are three buttons: 'Save & Test' (green), 'Delete' (red), and 'Back' (grey).

3. Final logs looks like this

c. live record



done !

PostgreSQL Pg_Backrest Backup

Incremental backup

Installation

Install pg_backrest using the dnf on CentOS 8

```
$ sudo dnf install pgbackrest -y
```

Permissions set

```
$ sudo chown postgres:postgres /var/log/pgbackrest
```

Make Directory


```
$ sudo mkdir -p /etc/pgbackrest
$ sudo mkdir -p /etc/pgbackrest/conf.d
$ sudo touch /etc/pgbackrest/pgbackrest.conf
```

Edit the file "pgbackrest"

```
--> vim /etc  
  
/pgbackrest  
[global]  
repol-path=/var/lib/pgbackrest  
#[main]  
#pg1-path=/var/lib/pgsql/12/data  
[testing]  
pg1-path=/var/lib/pgsql/12/data
```

Make sure about the postgresql.conf

```
archive_mode = on  
archive_command = 'pgbackrest --stanza=testing archive-push %p'  
max_wal_senders = 3  
wal_level = replica
```

 Restart PostgreSQL after configuration

Change the file permission and ownership

```
sudo chmod 640 /etc/pgbackrest/pgbackrest.conf  
sudo chown postgres:postgres /etc/pgbackrest/pgbackrest.conf  
sudo chmod 640 /var/log/pgbackrest/testing-stanza-create.log  
sudo chown postgres:postgres /tmp/pgbackrest/
```

Stanza creation **USER MUST BE POSTGRES**

```
pgbackrest --stanza=testing --log-level-console=info stanza-create  
pgbackrest --stanza=testing --log-level-console=info check  
pgbackrest --stanza=testing --type=diff --log-level-stderr=info backup
```

Recovery Test **AS A ROOT USER**

Removing the /data/

```
systemctl restart postgresql-12.service
rm -rf /var/lib/pgsql/12/data/
mkdir /var/lib/pgsql/12/data
chown postgres:postgres -R ./data/
```

Restoring the data

```
$ pgbackrest --stanza=testing --log-level-stderr=info restore
```

Cross checking

```
* su - postgres
* psql
* \l ->> it shows old database
```

✔ done