Tutorial Percona PostgreSQL HA

by Pietro Cornelio

release documento 0.1.0 Luglio 2022
release documento 0.1.1 Maggio 2023
release documento 0.1.2 Giugno 2023
release documento 0.1.3 Giugno 2023
release documento 0.1.4 Giugno 2023
release documento 0.1.5 Giugno 2023
release documento 0.1.6 Giugno 2023
release documento 0.1.7 Agosto 2023

Prefazione

In questo tutorial vedremo come installare correttamente **un impianto cluster in alta disponibilità HA (High Avalaibility) dell'RDBMS PostgreSQL** insieme al framework **Patroni**, un **archivio ETCD** e un **bilanciatore** basato su **HA Proxy** più altre tecnologie che rendono questo impianto altamente resiliente, funzionale e potente nel gestire connessioni client SQL concorrenti.

Patroni è un framework gestore di cluster in grado di personalizzare e automatizzare l'implementazione e la manutenzione dei cluster PostgreSQL HA (High Availability). Supporta il failover automatico del database e la replica in streaming. E' scritto in Python.

HAProxy è un sistema di bilanciamento del carico e proxy per applicazioni basate su TCP e HTTP.

ETCD Cluster (DCS) è un archivio chiave-valore distribuito fortemente coerente che fornisce un modo affidabile per archiviare i dati a cui un sistema distribuito o un cluster di macchine deve accedere. Usa Etcd per archiviare lo stato del cluster PostgreSQL per mantenere attivo e funzionante il cluster Postgres.

Keepalived serve per fornire un IP virtuale VIP ai 2 nodi HAProxy per garantire l'HA di connessione.

3 o più nodi dedicati a PostgreSQL che hanno a bordo anche l'agent Patroni.

Introduzione

Siccome la configurazione di un cluster PostgreSQL in HA basato sul framework Patroni, ETCD, PgBouncer ed altre eventuali tecnologie non è una implementazione banale e PostgreSQL deve essere predisposto per funzionare correttamente con Patroni con una serie di riconfigurazioni, si è scelto di utilizzare la distribuzione **PostgreSQL di Percona** che ci permetterà di disporre di ogni pacchetto software perfettamente allineato con la distribuzione GNU/Linux (Debian/Ubuntu o Red-Hat like) per funzionare con il framework Patroni, evitando così settaggi di tuning e messe a punto per risolvere alcune fastidiose frammentazioni di compatibilità delle varie distribuzioni GNU/Linux.

Inoltre Percona permette di scaricare e installare altri comodi pacchetti per aggiungere con relativa facilità interessanti funzionalità a PostgreSQL come ad esempio **audit**, **analisi**, **backup e restore**, etc., etc. un elenco delle tecnologie addon che installeremo e relativa descrizione verrà esposta di seguito.

Per maggiori dettagli sulla distribuzione Percona per PostgreSQL in HA si veda:
https://www.percona.com/ha-for-postgresql

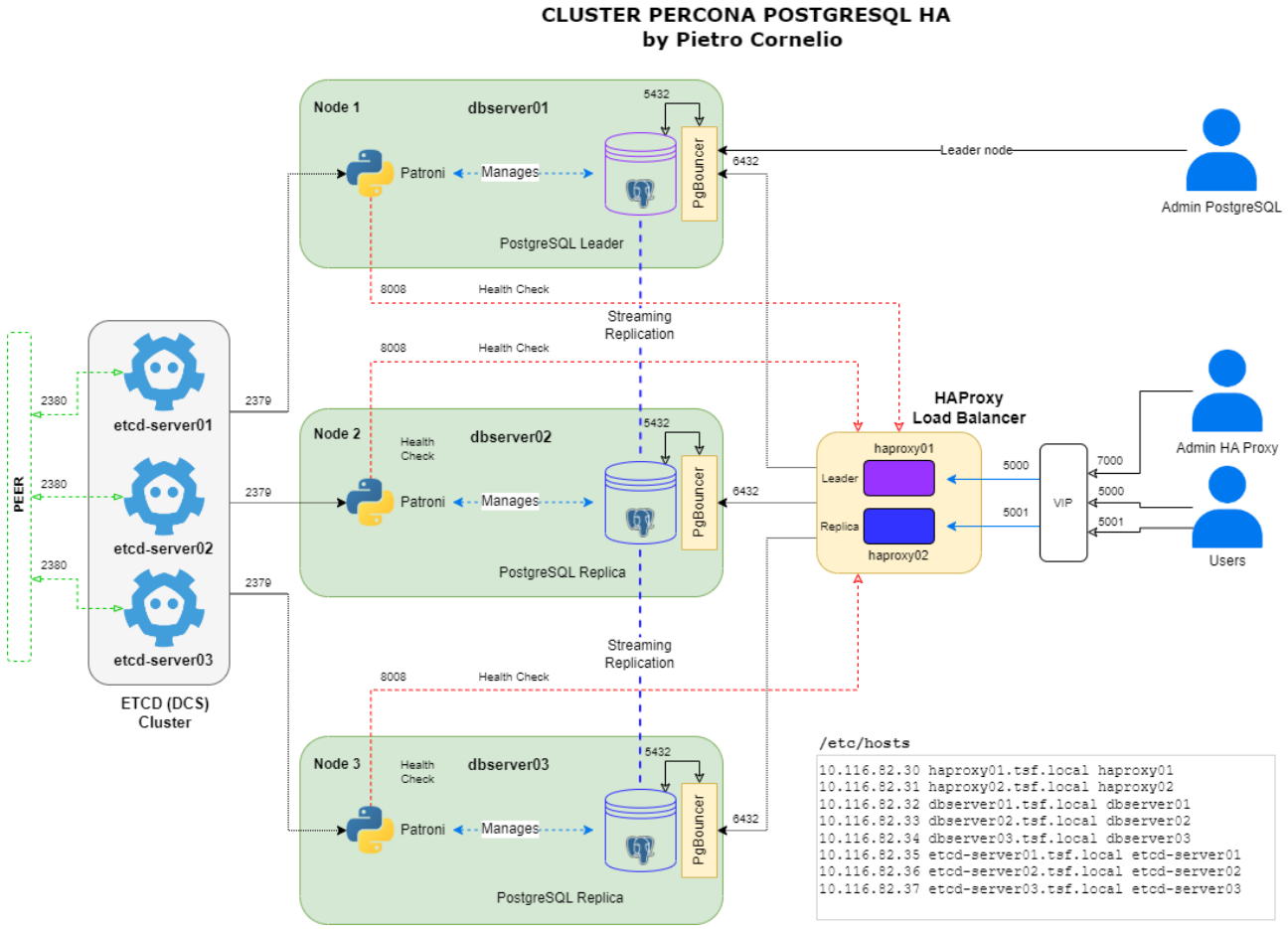


Figura 1 – Architettura Cluster PostgreSQL in HA ambienti di produzione

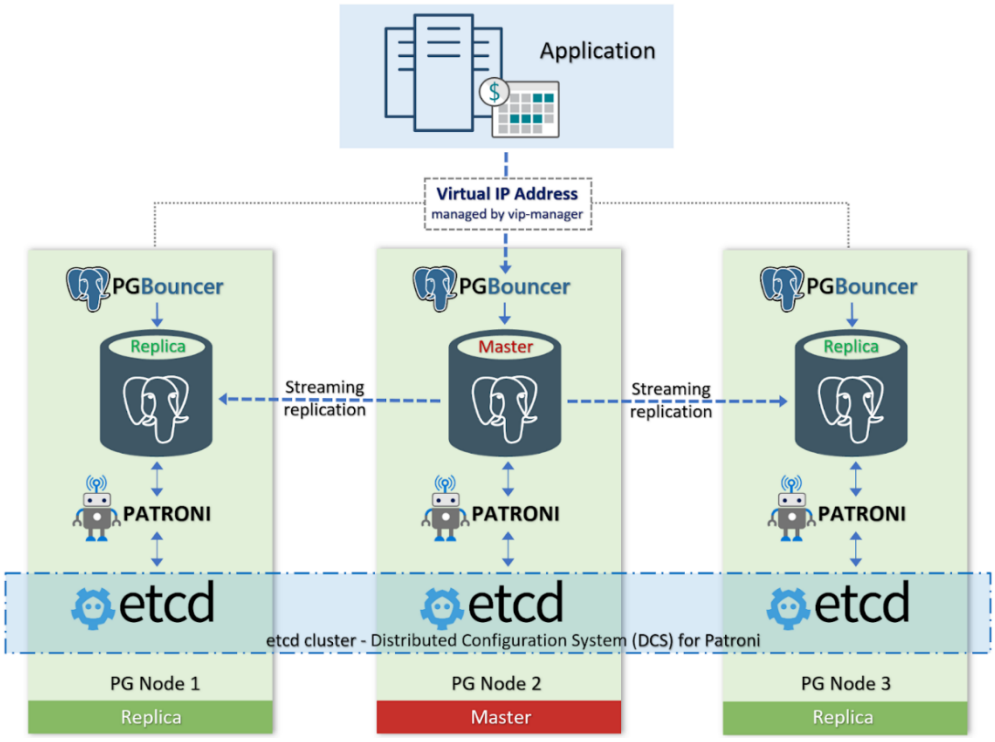


Figura 2 - Infrastruttura PostgreSQL in HA ambienti di sviluppo

Le porte di rete di default relative ai servizi delle tecnologie usate sono:

5432 PostgreSQL database standard port;
6432 PgBouncer standard port;
8008 patroni rest api port required by HAProxy to check the nodes status;
2379 etcd client port required by any client including patroni to communicate with etcd;
2380 etcd peer urls port required by the etcd members communication;
5000 HAProxy front-end listening port for back-end leader database server;
5001 HAProxy front-end listening port for back-end replica database servers;
7000 HAProxy HTTP port to access stats dashboard;

Nel nostro tutorial abbiamo usato **Ubuntu 22.04.x LTS per i nodi del cluster PostgreSQL e del cluster ETCD**
Mentre per i **2 nodi del cluster HAProxy abbiamo usato Debian 11.7** su cui abbiamo installato il kernel linux 6.1.x più performante abilitando il repo backports:

- echo "deb http://deb.debian.org/debian bullseye-backports main" | tee -a /etc/apt/sources.list
- apt-get update
- apt -t bullseye-backports upgrade

Naturalmente è possibile utilizzare Ubuntu 22.04.x LTS per HA Proxy senza alcun problema

Su tutti i nodi configurare /etc/hosts come segue
Naturalmente cambiare IP e dominio relativi al proprio ambiente

```
10.116.82.30    haproxy01.tsf.local     haproxy01
10.116.82.31    haproxy02.tsf.local     haproxy02
10.116.82.32    dbserver01.tsf.local    dbserver01
10.116.82.33    dbserver02.tsf.local    dbserver02
10.116.82.34    dbserver02.tsf.local    dbserver03
10.116.82.35    etcd-server01.tsflocal  etcd-server01
10.116.82.36    etcd-server02.tsflocal  etcd-server02
10.116.82.37    etcd-server03.tsflocal  etcd-server03
```

Importante:

Sincronizzare la data e l'orario su tutti i nodi: sudo timedatectl set-timezone Europe/Rome

Tecnologie software (estensioni aggiuntive) utilizzate

Percona pg_repack

è uno dei progetti di estensione più vecchi e ampiamente utilizzati per PostgreSQL. È così popolare che persino i fornitori di servizi DBaaS non possono evitarlo. È uno "strumento potente" nelle mani di un DBA per gestire tabelle gonfie/frammentate. Nessun progettista di sistemi IT immagina un'implementazione di produzione seria senza di esso. Sostituisce magicamente le tabelle gonfie e frammentate con una nuova tabella completamente costruita senza tenere un lock esclusivo sulla tabella durante la sua lavorazione. Questa estensione ha reso quasi inutili i comandi integrati di PostgreSQL come VACUUM FULL e CLUSTER.

PostgreSQL pgAudit

fornisce una registrazione dettagliata della sessione e/o dell'oggetto tramite la funzione di registrazione standard fornita da PostgreSQL. L'obiettivo di PostgreSQL Audit è fornire gli strumenti necessari per produrre i registri di controllo necessari per superare determinati controlli di certificazione governativi, finanziari o ISO. Il termine "audit" si riferisce a un'ispezione ufficiale dei conti di un individuo o di un'organizzazione, tipicamente da parte di un organismo indipendente. Il progetto pgaudit si basa sul lavoro svolto dai membri della PostgreSQL Development Community a supporto di una capacità di audit open source per PostgreSQL. Il codice sorgente associato a pgaudit è disponibile con la licenza PostgreSQL.

pgBackRest

è un tool che mira a essere una soluzione di backup e ripristino affidabile e facile da usare in grado di scalare senza problemi i database e i carichi di lavoro più grandi utilizzando algoritmi ottimizzati per requisiti specifici del database che si sta backuppando o restorando.

pgBouncer

è un potente, leggero e altamente configurabile software di pool-connection che permette di realizzare sistemi di connessione multiutente, concorrente e multi database per PostgreSQL alleggerendo molto il carico sull'engine di PostgreSQL. Nei capitoli successivi sarà apprpfondito in quanto è un componente importante del nostro cluster.

pgAudit-set_user

fornisce un livello aggiuntivo di registrazione e controllo quando gli utenti senza privilegi devono passare a ruoli di superutente o proprietario dell'oggetto di un database per eseguire le attività di manutenzione.

è un analizzatore delle prestazioni di PostgreSQL, creato per la velocità con rapporti completamente dettagliati basati sui file di registro di PostgreSQL.

pgBadger

è un potente, veloce e versatile SQL Log Analyzer per PostgreSQL fornendo in automatico grafici e report utilissimi per un DBA per scovare anomali in tabelle, buffer, WAL, etc., etc.

wal2json

wal2json è un'estensione di decodifica logica. Con questa estensione è possibile accedere alle tuple generate da INSERT e UPDATE e analizzare il contenuto in WAL; wal2json genererà un oggetto JSON in ogni transazione. Tutte le tuple nuove/vecchie sono fornite nell'oggetto JSON e le opzioni aggiuntive possono includere anche attributi come timestamp della transazione, struttura limitata, tipo di dati e ID transazione.

Facciamo un minimo di studio riguardo le tecnologie vitali utilizzate

Cosa è il software Patroni e cosa fa

Patroni è un sistema di gestione di alta disponibilità per database PostgreSQL. Una volta installato e configurato correttamente, Patroni si occuperà di gestire in completa autonomia l'avvio, l'arresto e la gestione di un cluster di database PostgreSQL.
Quando viene avviato il servizio Patroni su tutti i nodi del cluster PostgreSQL, esso si prenderà cura di avviare automaticamente le istanze di PostgreSQL sui nodi configurati come membri del cluster. Patroni monitorerà lo stato del cluster e, se uno dei nodi dovesse fallire, Patroni prenderà provvedimenti per mantenere la disponibilità del database, ad esempio promuovendo un nodo di standby a leader.
In sintesi, dopo aver installato Patroni, non è necessario avviare manualmente PostgreSQL. Patroni si occuperà di gestire l'avvio e l'arresto del database PostgreSQL come parte del suo processo di gestione dell'alta disponibilità.

Come gestisce le connessioni PostgreSQL

PostgreSQL ha un'architettura di gestione delle connessioni piuttosto pesante. Per ogni connessione in entrata, il postmaster (il demone principale di Postgres) crea un nuovo processo (chiamato convenzionalmente backend) per gestirlo. Sebbene questo design offra stabilità e isolamento migliori, non lo rende particolarmente efficiente nella gestione di connessioni di breve durata. Una nuova connessione client Postgres implica la configurazione del TCP, la creazione del processo e l'inizializzazione del back-end, tutte operazioni costose in termini di tempo e risorse di sistema.
PostgreSQL è stato creato per gestire un numero limitato di connessioni di lunga durata.
Ecco perché **ha un limite predefinito relativamente piccolo di connessioni** e non è raro che gli applicativi ricevano l'errore "limite di connessione superato":
ERROR: remaining connection slots are reserved for non-replication superuser connections
Si potrebbe essere tentati di aumentare semplicemente tale limite, ma è sbagliato farlo, ci sono modi migliori. PostgreSQL di solito completa 10.000 transazioni più velocemente se le dividiamo a 5, 10 o 20 alla volta piuttosto che eseguendole 500 alla volta. Determinare esattamente quante transazioni dovrebbero essere eseguite contemporaneamente varia in base al carico di lavoro e richiede una messa a punto (tuning) del proprio ambiente di DB. Un altro aspetto da tenere in considerazione con le connessioni al DB è che la gestione per la loro creazione e distruzione può essere responsabile della metà del tempo di funzionamento e dell'utilizzo delle risorse. Per questo motivo, sarebbe consigliabile disporre di connessioni persistenti. Però questo, a sua volta, può far esacerbare ancora di più il problema del limite precedente.
Allora come possiamo risolvere questo problema?
La risposta è usare un pool di connessioni. A volte il framework che si sta utilizzando per sviluppare un'applicazioni lo offre già (come SQLAlchemy), ma in altri "use case" potresti aver bisogno di uno strumento specializzato. Però se stai utilizzando più macchine DB per servire la tua API ti imbatterai nuovamente nel problema precedente poiché il pooler del framework è responsabile solo della gestione dell'istanza su cui è in esecuzione. Quando hai più server, quindi, una buona soluzione è usare **PGBouncer.** Questo fantastico strumento ti consentirà di connettere più server ad esso invece di connetterti direttamente a Postgres. Accoda le connessioni in entrata quando la quantità simultanea è maggiore di quella specificata dall'utente.

PgBouncer quando usarlo ed in quali contesti

L'utilizzo di PgBouncer non è strettamente necessario per creare un cluster HA (High Availability) di PostgreSQL con Patroni e HAProxy. Tuttavia, PgBouncer può fornire ottimi vantaggi aggiuntivi in termini di scalabilità e prestazioni. PgBouncer è un proxy di connessione al database che può essere utilizzato per gestire le connessioni tra le applicazioni client e il cluster PostgreSQL fornendo un pool di connessioni e può migliorare drasticamente le prestazioni dell'applicazione riducendo l'impatto delle connessioni al database. Sebbene l'uso di PgBouncer non sia obbligatorio, può essere utile in situazioni in cui ci sia un elevato numero di connessioni simultanee al database (quindi **riduce l'overhead delle connessioni**) o quando si

desidera ottimizzare l'utilizzo delle risorse di sistema. Tuttavia, l'implementazione di PgBouncer richiede un'ulteriore configurazione e manutenzione. In definitiva, possiamo creare un cluster HA PostgreSQL con Patroni e HAProxy senza l'utilizzo di PgBouncer, ma bisogna considerare l'introduzione di PgBouncer se si desidera ottenere benefici aggiuntivi come la scalabilità delle connessioni e le prestazioni ottimizzate.

Infine, ricordo che…

OLTP o OLAP ? (perplessità Amlettiana)

PostgreSQL viene solitamente utilizzato negli scenari **OLTP** con un poco di **OLAP**.
Ad esempio, se un endpoint rivolto al pubblico contiene una query molto complessa che carica molti dati ed esegue operazioni complesse è probabile dovrebbe essere trattato come OLAP quindi l'endpoint deve essere sottoposto a un refactoring del codice per adottare una strategia algoritmica diversa.
Ricordo/riporto il mantra: *Mantieni le tue query snelle e utilizza le pipeline di dati per alimentare un data lake che può essere utilizzato per l'analisi…*

Ritorniamo a PgBouncer

PgBouncer ha tre tipi di modalità di pooling. Questo è importante da capire perché offre un grande potenziale per adattare il comportamento dello strumento alle nostre esatte esigenze di campo.

POOL DI SESSIONI
Session pooling significa che PgBouncer mantiene aperta una serie di connessioni al server. I client ne sceglieranno una e le query verranno indirizzate al database.
È anche possibile utilizzare pgpool per "restringere" il numero di connessioni realmente necessarie sul lato database. Questo può essere utile nel caso in cui si disponga di un pool di connessioni lato applicazione pazzoide che necessita di un numero insolitamente elevato di connessioni aperte che non dovrebbero arrivare al database di back-end in primo luogo.
Nel caso in cui tutte le connessioni funzionino, alcune di esse devono attendere fino a quando non è disponibile uno slot nel pool. Spesso molte applicazioni che utilizzano un pool di connessioni accedono allo stesso database. Molte app che eseguono pool troppo grandi potrebbero finire con troppe connessioni nel back-end, il che a sua volta può causare problemi. Ogni client eseguirà l'intera transazione e persino l'intera connessione sulla stessa connessione al database "reale".

POOL DI TRANSAZIONI
A volte una connessione completa è troppo. E per quanto riguarda il pool di transazioni? Invece di mappare un'intera connessione client a una vera connessione al database, è sufficiente garantire che la stessa transazione finisca sullo stesso host. PgBouncer quindi mapperà molte connessioni client alla stessa connessione fisica e le separerà per transazione.

POOL DI ISTRUZIONI
Il raggruppamento di istruzioni è di gran lunga il metodo più aggressivo. Spesso non hai bisogno di grandi blocchi di transazioni. Supponiamo di voler cercare nomi in una rubrica telefonica 1 milione di volte al secondo. Chiaramente si tratta di query rapide e di piccole dimensioni e non ci sono transazioni di grandi dimensioni che coprono più istruzioni. Pertanto, non dobbiamo preoccuparci della visibilità transazionale. Possiamo semplicemente bilanciare il carico di tutte quelle istruzioni su qualsiasi connessione e trasmettere il risultato al client.
Il caso d'uso è il seguente: pompare milioni di brevi istruzioni attraverso il sistema che non sono correlate a una logica aziendale di grandi dimensioni che richiede un blocco pesante (ad esempio SELEZIONA PER AGGIORNARE) o qualcosa del genere.

Quindi si sceglierà il metodo di pooling in funzione della specificità delle applicazioni che insisteranno sul Database PostgreSQL cioè se sono OLAP, OLTP o miste.

http://www.pgbouncer.org/

Perché è necessario un cluster etcd?

ETCD è un Distributed Consensus Store (DCS) ed è necessario per fornire un meccanismo decisionale distribuito per consentire a Patroni di determinare con rigore ed efficienza quale istanza (nodo) PostgreSQL deve essere il leader e quali nodi devono essere le repliche.
Nel nostro esempio, creeremo un **cluster etcd a 3 nodi**: vedi nodi etcd-server01-02-03.
È possibile utilizzare anche Zookeeper, ma non verrà trattato in questo articolo. Il numero di nodi etcd non dipende dal numero di nodi Patroni nel cluster. Tuttavia, bisogna averne più di uno per prevenire un singolo punto di errore ed è consigliabile avere almeno 3 nodi per consentire la maggioranza durante la votazione del leader in caso di errore.

Su quali nodi conviene installare PgBouncer?

La decisione dove installare PgBouncer dipende dalle esigenze specifiche e dall'architettura del sistema. Di seguito alcune considerazioni generali:

1. **Installazione su nodi HAProxy**: Se scegli di installare PgBouncer sui nodi HAProxy, puoi configurare il tuo stack in questo modo: le applicazioni client si connettono a PgBouncer, che agisce come proxy per le connessioni al database. PgBouncer quindi distribuisce le connessioni ai nodi PostgreSQL del cluster. Questo approccio può semplificare la configurazione, in quanto hai un punto centrale per il routing delle connessioni e il bilanciamento del carico. Inoltre, se hai un'architettura a più livelli, con più client che si connettono tramite HAProxy, l'installazione di PgBouncer su questi nodi può semplificare il routing delle connessioni.
2. **Installazione sui nodi PostgreSQL**: Se decidi di installare PgBouncer sui nodi PostgreSQL, ogni nodo del cluster avrà la sua istanza di PgBouncer locale. In questo caso, ogni nodo funzionerà come proxy per le connessioni dirette al database locale. Questo approccio può fornire un maggiore controllo sulla gestione delle connessioni in quanto ogni nodo del cluster ha il proprio pool di connessioni. Tuttavia, richiede la configurazione e la gestione di PgBouncer su ogni nodo PostgreSQL, il che può comportare un po' più di lavoro. **Questa è la configurazione utilizzata nel nostro cluster.**

Come funziona Patroni con il DCS di ETCD ?

Circa ogni 10 secondi, Patroni controlla ogni nodo PostgreSQL del cluster riguardo la presenza di un primario nel Distributed Configuration Store (DCS). Se non viene trovata alcuna posizione primaria (leader), inserisce una chiave nel DCS per richiedere la posizione primaria. Quindi se questo server risulta essere il primario, informa HAProxy di usarlo come nuovo master/leader. D'altra parte, se viene trovato un server primario/leader, esegue una serie di controlli e tenta di trasformare il server corrente in una versione di replica. I meccanismi software algoritmici dietro la coppia Patroni+ETCD è stata strutturata seguendo criteri di progettazione ingegneristica semplici per rendere questa tecnologia estremamente affidabile e funzionale.

Connessione da un nodo client esterno verso il cluster PostgreSQL attraverso HAProxy

Prefazione doverosa di connessione client PostgreSQL attraverso HAProxy

Un'implementazione comune dell'alta disponibilità in un ambiente PostgreSQL fa uso di un proxy: invece di connettersi direttamente al server del database, l'applicazione si connetterà invece al proxy, che inoltrerà la richiesta a PostgreSQL. Quando HAproxy viene utilizzato per questo, è anche possibile instradare le richieste di lettura a una o più repliche, per il bilanciamento del carico. Tuttavia, questo non è un processo trasparente: l'applicazione client deve esserne consapevole e suddividere la sola lettura dal traffico di lettura-scrittura stesso. Con HAproxy, questo viene fatto fornendo due diverse porte per la connessione dell'applicazione:

Query di scrittura porta 5000
Query di lettura porta 5001

Cosa fa Keepalived in questo progetto?

Fornisce un indirizzo IP virtuale legato al **cluster a 2 nodi di HAProxy** (uno primario e l'altro in standby), siccome abbiamo 2 nodi HAProxy per fare un HA, utilizzeremo un VIP fornito da Keepalived installato sui 2 nodi HAProxy. Bisogna assicurarsi che la configurazione di Keepalived abbia una regola per il rilevamento dello stato dei nodi HAProxy, ad esempio tramite il controllo dello stato delle porte di ascolto o tramite uno script personalizzato. In questo modo, Keepalived può monitorare l'integrità dei nodi HAProxy e assegnare il VIP solo a al nodo attivo.

Nota:
In questo progetto, un 2° HD (sdb1) è montato sotto la directory **/var/lib/postgresql**
Sotto la quale c'è la directory **../14/main** dedicata a PostgreSQL :

```
pietro@dbserver02:~$ df -h
Filesystem      Size  Used Avail Use% Mounted on
tmpfs           393M  1.2M  392M   1% /run
/dev/sda2        16G  7.9G  7.0G  54% /
tmpfs           2.0G     0  2.0G   0% /dev/shm
tmpfs           5.0M     0  5.0M   0% /run/lock
/dev/sdb1        32G  261M   32G   1% /var/lib/postgresql
tmpfs           393M  4.0K  393M   1% /run/user/1000
```

Preparazione della directory dedicata a PostgreSQL su HD dedicato

sudo mkdir -p /var/lib/postgresql/14/main/

sudo chown postgres:postgres -R /var/lib/postgresql

Installazione del cluster Percona PostgreSQL

Prima di tutto aggiorniamo il sistema: cosa santa e giusta…
sudo apt update
sudo apt upgrade

Configuriamo il Percona repository

wget https://repo.percona.com/apt/percona-release_latest.$(lsb_release -sc)_all.deb
Installiamo il package appena scaricato che configurerà il repository
sudo dpkg -i percona-release_latest.$(lsb_release -sc)_all.deb
Aggiorniamo la cache dei pacchetti
sudo apt update
Abilitiamo il repository specifico (major release)
Nel nostro caso usiamo la release 14
PostgreSQL 15 : sudo percona-release setup ppg-15
PostgreSQL 14 : sudo percona-release setup ppg-14
Installiamo PostgreSQL 14
sudo apt install percona-postgresql-14

Ora installiamo i componenti/tecnologie aggiuntivi :
pg_repack: sudo apt install percona-postgresql-14-repack
pgAudit: sudo apt install percona-postgresql-14-pgaudit
pgBackRest: sudo apt install percona-pgbackrest
pgBouncer: sudo apt install percona-pgbouncer
pgAudit-set_user: sudo apt install percona-pgaudit14-set-user
pgBadger: sudo apt install percona-pgbadger
wal2json: sudo apt install percona-postgresql-14-wal2json

Install PostgreSQL contrib extensions: sudo apt install percona-postgresql-contrib

Il processo di installazione inizializza e avvia automaticamente il database predefinito.
È possibile controllare lo stato del database utilizzando il seguente comando:
sudo systemctl status postgresql.service
Test di connessione al server PostgreSQL
Per impostazione predefinita, l'utente postgres e il database postgres vengono creati in PostgreSQL al momento dell'installazione e dell'inizializzazione. Ciò consente di connettersi al database come utente postgres.
sudo su postgres
psql
oppure direttamente: sudo su postgres psql
per uscire dal client psql : \q
postgres=# \q

Il servizio PostregreSQL 14 deve essere spento e disabilitato perché sarà cura di Patroni gestire PostgreSQL accedendo direttamente ai binari di PostgreSQL
Prima di spegnere aspetta qualche minuto per evitare ci siano ancora in corsa delle attività interne di PostgreSQL…
sudo systemctl stop postgresql
sudo systemctl disable postgresql
Rimuovere la directory dei dati.
Patroni richiede un ambiente pulito per inizializzare un nuovo cluster, quindi rimuovere tutto nella dir "main":
sudo rm -rf /var/lib/postgresql/14/main/*

Installiamo e configuriamo un ETCD distributed store

Su tutti e 3 i nodi **etcd-serverXX**
sudo apt install etcd
systemctl status etcd
sudo systemctl stop etcd

Facciamo una copia del file di configurazione originale:

sudo mv /etc/default/etcd /etc/default/etcd.orig
sudo vim /etc/default/etcd
Configurare il servizio etcd sui 3 nodi DBServerXX

File : /etc/default/etcd

```
ETCD_NAME=etcd-server01
ETCD_INITIAL_CLUSTER="etcd-server01=http://10.116.82.35:2380,etcd-
server02=http://10.116.82.36:2380,etcd-server03=http://10.116.82.37:2380"
ETCD_INITIAL_CLUSTER_TOKEN="etcd-cluster"
ETCD_INITIAL_CLUSTER_STATE="new"
ETCD_INITIAL_ADVERTISE_PEER_URLS="http://10.116.82.35:2380"
ETCD_DATA_DIR="/var/lib/etcd/postgresql"
ETCD_LISTEN_PEER_URLS="http://10.116.82.35:2380"
ETCD_LISTEN_CLIENT_URLS="http://10.116.82.35:2379,http://localhost:2379"
ETCD_ADVERTISE_CLIENT_URLS=http://10.116.82.35:2379
ETCD_ENABLE_V2="true"

ETCD_NAME=etcd-server02
ETCD_INITIAL_CLUSTER="etcd-server01=http://10.116.82.35:2380,etcd-
server02=http://10.116.82.36:2380,etcd-server03=http://10.116.82.37:2380"
ETCD_INITIAL_CLUSTER_TOKEN="etcd-cluster"
ETCD_INITIAL_CLUSTER_STATE="existing"
ETCD_INITIAL_ADVERTISE_PEER_URLS="http://10.116.82.36:2380"
ETCD_DATA_DIR="/var/lib/etcd/postgresql"
ETCD_LISTEN_PEER_URLS="http://10.116.82.36:2380"
ETCD_LISTEN_CLIENT_URLS="http://10.116.82.36:2379,http://localhost:2379"
ETCD_ADVERTISE_CLIENT_URLS=http://10.116.82.36:2379
ETCD_ENABLE_V2="true"

ETCD_NAME=etcd-server03
ETCD_INITIAL_CLUSTER="etcd-server01=http://10.116.82.35:2380,etcd-
server02=http://10.116.82.36:2380,etcd-server03=http://10.116.82.37:2380"
ETCD_INITIAL_CLUSTER_TOKEN="etcd-cluster"
ETCD_INITIAL_CLUSTER_STATE="existing"
ETCD_INITIAL_ADVERTISE_PEER_URLS="http://10.116.82.37:2380"
ETCD_DATA_DIR="/var/lib/etcd/postgresql"
ETCD_LISTEN_PEER_URLS="http://10.116.82.37:2380"
ETCD_LISTEN_CLIENT_URLS="http://10.116.82.37:2379,http://localhost:2379"
ETCD_ADVERTISE_CLIENT_URLS="http://10.116.82.37:2379"
ETCD_ENABLE_V2="true"
```

Abilitiamo il servizio e poi lo avviamo su tutti i nodi

sudo systemctl enable etcd

sudo systemctl start etcd

Verifichiamo che il servizio sia partito correttamente

systemctl status etcd

Controlliamo i membri del cluster ETCD

sudo etcdctl member list

pietro@dbserver01:~$ sudo etcdctl member list

pietro@dbserver01:~$ sudo etcdctl member list

pietro@etcd-server01:~$ etcdctl member list

```
894a289dfb971b63: name=etcd-server01 peerURLs=http://10.116.82.35:2380 clientURLs=http://10.116.82.35:2379 isLeader=true
f3ab697e21457137: name=etcd-server03 peerURLs=http://10.116.82.37:2380 clientURLs=http://10.116.82.37:2379 isLeader=false
fba96b4b3359c41f: name=etcd-server02 peerURLs=http://10.116.82.36:2380 clientURLs=http://10.116.82.36:2379 isLeader=false
```

Facciamo una chiamata REST al primo nodo del cluster ETCD

```
pietro@dbserver01:~$ curl -s http://10.116.82.35:2380/members |jq -r
[
  {
    "id": 9892764190468676000,
    "peerURLs": [
      "http://10.116.82.35:2380"
    ],
    "name": "etcd-server01",
    "clientURLs": [
      "http://10.116.82.35:2379"
    ]
  },
  {
    "id": 17558243562679005000,
    "peerURLs": [
      "http://10.116.82.37:2380"
    ],
    "name": "etcd-server03",
    "clientURLs": [
      "http://10.116.82.37:2379"
    ]
  },
  {
    "id": 18134143345312254000,
    "peerURLs": [
      "http://10.116.82.36:2380"
    ],
    "name": "etcd-server02",
    "clientURLs": [
      "http://10.116.82.36:2379"
    ]
  }
]
```

Per promuovere un nodo a leader posizionarsi sul nodo che è leader e dare il comando

```
pietro@dbserver03:~$ etcdctl member list

etcdctl move-leader 25159af76f8c0e33
Leadership transferred from 1611461150f17ef7 to 25159af76f8c0e33
```

Importante:

In caso si volesse cancellare l'archivio DCS di ETCD basta lanciare il comando (spegnere prima il servizio):

```
sudo rm -rf /var/lib/etcd/postgresql/*
```

Servizio watchdog per Patroni-PostgreSQL

Il kernel Linux utilizza l'utilità chiamata watchdog per proteggersi da un sistema che non risponde. Il watchdog monitora un sistema cercando **errori irreversibili** dell'applicazione: risorse di sistema esaurite, blocco di processi su device in hang, ecc. quindi avvia un riavvio del nodo per riportare in modo sicuro il sistema a uno stato funzionante. La funzionalità watchdog è utile per i server destinati a funzionare senza intervento umano per lungo tempo in ambienti critici.
I dispositivi watchdog sono meccanismi software o hardware che ripristineranno l'intero sistema quando non ottengono un battito cardiaco keepalive entro un periodo di tempo specificato. Ciò aggiunge un ulteriore livello di sicurezza nel caso in cui i normali meccanismi di protezione Patroni **split-brain** falliscano. Sebbene l'uso di un meccanismo di watchdog con Patroni sia facoltativo, non si dovrebbe mai prendere in considerazione un sistema HA di PostgreSQL in produzione senza watchdog. Dipende però se l'ambiente è "mission critical" altrimenti se l'ambiente è un classico gestionale per servizi di pubblica utilità si potrebbe trascurare avere un watchdog.
In questo tutorial faremo riferimento all'implementazione **Softdog**, un'implementazione software standard per watchdog fornita con GNU/Linux

Il pericoloso fenomeno dello "Split Brain"
Avere più server PostgreSQL in esecuzione come primari può comportare la perdita di transazioni a causa di tempistiche divergenti. Questa situazione si chiama problema del cervello diviso (split brain). Per evitare lo "**split brain**", Patroni deve assicurarsi che PostgreSQL non accetterà alcun commit di transazione dopo la scadenza della chiave leader nel DCS. In circostanze normali Patroni funziona correttamente: esso proverà a raggiungere questo obiettivo arrestando PostgreSQL quando l'aggiornamento del blocco leader fallisce per qualsiasi motivo. Tuttavia, ciò potrebbe non accadere per vari motivi:

1. Patroni si è arrestato in modo anomalo a causa di un bug, di una condizione di memoria insufficiente o per essere stato ucciso accidentalmente da un amministratore di sistema.
2. L'arresto di PostgreSQL è troppo lento.
3. Patroni non riesce a funzionare a causa dell'elevato carico sul sistema, della VM messa in pausa dall'hypervisor o di altri problemi di infrastruttura.

Per garantire un comportamento corretto in queste condizioni di stallo, **Patroni supporta i dispositivi watchdog**. Patroni quindi proverà ad attivare il watchdog prima di promuovere PostgreSQL a primary.
Se l'attivazione del watchdog fallisce ed è richiesta la modalità watchdog, il nodo rifiuterà di diventare leader. Quando decide di partecipare all'elezione del leader, Patroni verificherà anche che la configurazione del watchdog gli consenta di diventare leader. Dopo aver retrocesso PostgreSQL (ad esempio a causa di un failover manuale) Patroni disabiliterà nuovamente il watchdog. Il watchdog sarà disabilitato anche mentre Patroni è in stato di pausa.
Attiviamo il watchdog su Linux
sudo modprobe softdog
Settiamo i permessi corretti:
sudo chown postgres:postgres /dev/watchdog*
Per rendere le modifiche attive al boot di Linux con i corretti permessi:

```
sudo sh -c 'echo "softdog" >> /etc/modules'
sudo sh -c 'echo "KERNEL=="watchdog", OWNER="postgres", GROUP="postgres"" >>
/etc/udev/rules.d/61-watchdog.rules'
```

Commentare nel file del kernel in uso la voce "blacklist softdog"

```
pietro@dbserver01:~$ uname -a
Linux dbserver01 5.15.0-73-generic
pietro@dbserver01:~$ grep blacklist /lib/modprobe.d/* /etc/modprobe.d/* |grep softdog
/lib/modprobe.d/blacklist_linux_5.15.0-73-generic.conf:# blacklist softdog
/lib/modprobe.d/blacklist_linux_5.4.0-150-generic.conf:blacklist softdog
sudo vim /lib/modprobe.d/blacklist_linux_5.15.0-73-generic.conf
```

Quando si aggiorna il kernel Linux va controllata la blacklist…

Premessa a un uso corretto della tecnologia Patroni

Per impostazione predefinita, Patroni configura il database PostgreSQL per la replica **asincrona**.
La scelta dello schema di replica dipende dall'ambiente di produzione.

Replica sincrona o asincrona?

PostgreSQL supporta la replica sincrona e asincrona per l'alta disponibilità e il ripristino di emergenza.
La replica sincrona significa che un'operazione di scrittura viene considerata completa solo dopo che essa è stata confermata come scritta sul server master e su uno o più server di standby sincroni. Ciò fornisce il massimo livello di durabilità e coerenza dei dati, poiché ai client viene garantito che le loro scritture siano state replicate su almeno un altro server prima che la scrittura venga riconosciuta come riuscita.
Tuttavia, il compromesso è che le prestazioni del server principale sono influenzate dal tempo necessario per confermare la scrittura sui server di standby sincroni, il che può rappresentare un collo di bottiglia negli ambienti con un elevato numero di scritture. Questo si risolve con nodi adeguatamente potenti in termini di CPU e RAM e reti di interconnessione tra i nodi componenti il cluster ad alta velocità a 10 Gbit/sec.
La replica asincrona, d'altra parte, significa che un'operazione di scrittura viene riconosciuta come riuscita non appena viene scritta sul server master, senza attendere alcuna conferma dai server in standby. Ciò fornisce prestazioni migliori in quanto il server master non è bloccato dal processo di replica, ma il compromesso è che **esiste il rischio di perdita di dati se il server master si guasta prima che la scrittura venga replicata sui server di standby.**
PostgreSQL ti consente di configurare una o più repliche sincrone e un numero qualsiasi di repliche asincrone, in modo da poter bilanciare i compromessi tra prestazioni e durabilità dei dati in base alle esigenze. Per ulteriori informazioni sulle repliche asincrone e sincrone, consultare la documentazione di Postgres e la documentazione di Patroni per determinare quale soluzione di replica è la migliore per le tue esigenze di produzione.
Per questa guida utilizzeremo la modalità di **replica asincrona predefinita di Patroni**.

Installiamo Percona Patroni

sudo apt install percona-patroni
Ora fermiamo il servizio e lo disabilitiamo (necessario per la configurazione)
sudo systemctl stop patroni
sudo systemctl disable patroni
il servizio viene creato in **/usr/lib/systemd/system/patroni.service**
Creare il file di configurazione **patroni.yml** nella directory **/etc/patroni**.
Il file contiene i valori di configurazione predefiniti per un cluster PostgreSQL.

Prendiamoci un momento per capire il contenuto del file patroni.yml.

La prima sezione fornisce i dettagli del primo nodo e delle sue porte di connessione. Successivamente, abbiamo il **servizio etcd** e i dettagli della sua porta.
A seguire, c'è l'importante sezione **bootstrap** che contiene le configurazioni di PostgreSQL dei passaggi/istruzioni da eseguire una volta inizializzato il database. Le voci **pg_hba.conf** specificano tutti gli altri nodi che possono connettersi al nodo e il loro meccanismo di autenticazione.

IMPORTANTE:

Un errore comune è riscontrare che l'agent Patroni si lamenta della mancanza di voci corrette nel file **pg_hba.conf**. Se vedi errori di quel tipo, devi aggiungere o correggere manualmente le voci in quel file e quindi riavviare il servizio. Una seconda modifica del file **patroni.yml** dopo il primo avvio di Patroni al riavvio del servizio Patroni, le modifiche non avranno alcun effetto perché la sezione bootstrap specifica la configurazione da applicare quando PostgreSQL viene avviato per la prima volta sul nodo.
Pertanto, la sezione bootstrap non ripeterà il processo anche se il file di configurazione Patroni viene modificato e il servizio viene riavviato.

Configurazione del cluster Patroni sui 3 nodi DBServerXX

In giallo evidenziato le modifiche da fare

Sul nodo dbserver01 : sudo vim /etc/patroni/patroni.yml

```
scope: postgres
namespace: /pg_cluster/
name: dbserver01

restapi:
  listen: 0.0.0.0:8008
  connect_address: 10.116.82.32:8008

etcd:
  hosts: 10.116.82.35:2379, 10.116.82.36:2379, 10.116.82.37:2379

bootstrap:
  # this section will be written into Etcd:/<namespace>/<scope>/config after initializing new cluster
  dcs:
    ttl: 30
    loop_wait: 10
    retry_timeout: 10
    maximum_lag_on_failover: 1048576
    synchronous_mode: false
    postgresql:
      use_pg_rewind: true
      use_slots: true
      parameters:
        wal_level: replica
        hot_standby: "on"
        logging_collector: 'on'
        max_wal_senders: 5
        max_replication_slots: 5
        wal_log_hints: "on"
        # Tuning Produzione by Pietro Cornelio
        # DBServer Node: 16 CPU – 64 Gb RAM
        # shared_buffers: 16GB
        # work_mem: 16MB
        # maintenance_work_mem: 2GB
        # max_worker_processes: 16
        # wal_buffers: 64MB
        # max_wal_size: 2GB
        # min_wal_size: 1GB
        # effective_cache_size: 64GB
        # fsync: on
        # default_statistics_target = 300
        # checkpoint_completion_target: 0.9
        # log_rotation_size: 100MB
        # max_connections: 1000
        # temp_buffers: 4MB

  # some desired options for 'initdb'
  initdb:  # Note: It needs to be a list (some options need values, others are switches)
  - encoding: UTF8
  - data-checksums

  pg_hba:  # Add following lines to pg_hba.conf after running 'initdb'
  #  - hostssl all all 0.0.0.0/0 md5
  - host replication replicator 127.0.0.1/32 md5
  - host all replicator 127.0.0.1/32 md5
  - host replication replicator 0.0.0.0/0 md5
  - host all replicator 0.0.0.0/0 md5
  - host all all 0.0.0.0/0 md5

  # Additional script to be launched after initial cluster creation
  # post_init: /usr/local/bin/setup_cluster.sh
  # Some additional users users which needs to be created after initializing new cluster

  users:
    admin:
      password: admin
      options:
        - createrole
        - createdb
    replicator:
```

```
      password: password
      options:
        - replication
postgresql:
  listen: 0.0.0.0:5432
  connect_address: 10.116.82.32:5432
  data_dir: "/var/lib/postgresql/14/main/"
  bin_dir: "/usr/lib/postgresql/14/bin"
  # config_dir:
  pgpass: /tmp/pgpass0
  authentication:
    replication:
      username: replicator
      password: replicator
    superuser:
      username: postgres
      password: postgres
    rewind:
      username: pgrewind
      password: pgrewind
  parameters:
    unix_socket_directories: '/var/run/postgresql'

watchdog:
  mode: automatic # Allowed values: off, automatic, required
  device: /dev/watchdog
  safety_margin: 5

tags:
    nofailover: false
    noloadbalance: false
    clonefrom: false
    nosync: false
```

Verifichiamo la correttezza di configurazione del file patroni.yml

sudo patroni --validate-config /etc/patroni/patroni.yml

Verifichiamo i permessi alla directory /var/lib/postgresql/14/main/

Directory **/var/lib/posgresql** montata sul 2° HD

```
ls -ls /var/lib/
```

In caso ci siamo dimenticati di settare i permessi facciamolo :

```
sudo chown postgres:postgres -R /var/lib/postgresql
```

Avviamo Patroni sul 1° nodo e se tutto è OK avviamolo anche sul 2° e 3° nodo

sudo systemctl start patroni && sudo journalctl -u patroni.service -n 100 -f

sudo systemctl status patroni

sudo journalctl -u patroni.service -n 100 -f

Se tutto funziona, possiamo abilitare il servizio su tutti i nodi

systemctl enable patroni

Verifichiamo lo stato del cluster PostgreSQL interrogando patroni

```
pietro@dbserver03:~$ patronictl -c /etc/patroni/patroni.yml list
+ Cluster: postgres --------+---------+---------+----+-----------+
| Member     | Host         | Role    | State   | TL | Lag in MB |
+------------+--------------+---------+---------+----+-----------+
| dbserver01 | 10.116.82.32 | Leader  | running |  1 |           |
| dbserver02 | 10.116.82.33 | Replica | running |  1 |         0 |
| dbserver03 | 10.116.82.34 | Replica | running |  1 |         0 |
+------------+--------------+---------+---------+----+-----------+
```

Oppure `patronictl -c /etc/patroni/patroni.yml topology`

a maronna cia' ccumpagna…

Legge di Hofstadter:
Per fare una cosa ci vuole sempre più tempo di quanto si pensi, anche tenendo conto della Legge di Hofstadter.

Configurazione di PgBouncer

PgBouncer si basa su un file di configurazione principale, generalmente archiviato come **/etc/pgbouncer/pgbouncer.ini**

Si può invocare pgbouncer come servizio systemd o semplicemente eseguirlo anche senza privilegi di superutente con il percorso di questo file di configurazione. Noi preferiamo usare systemd:

```
sudo systemctl stop pgbouncer
sudo systemctl disable pgbouncer
```

Sul primo nodo DBServer01

Colleghiamoci a PostgreSQL

```
pietro@dbserver01:~$ sudo su - postgres
postgres@dbserver01:~$ psql
psql (14.7 - Percona Distribution (Ubuntu 2:14.7-1.jammy))
Type "help" for help.
postgres=#
```

Eseguiamo le 4 query seguenti:

```
--
-- EXECUTE AS SUPERUSER postgres
--
-- execute on each database user accounts will login
--
CREATE ROLE pgbouncer LOGIN PASSWORD 'password.2023';
CREATE FUNCTION public.lookup (
       INOUT p_user     name,
       OUT   p_password text
) RETURNS record
LANGUAGE sql SECURITY DEFINER SET search_path = pg_catalog AS
$$SELECT usename, passwd FROM pg_shadow WHERE usename = p_user$$;

-- make sure only "pgbouncer" can use the function
REVOKE EXECUTE ON FUNCTION public.lookup(name) FROM PUBLIC;
GRANT EXECUTE ON FUNCTION public.lookup(name) TO pgbouncer;
```

Ora copiamo la password crittografata di pgbouncer dalla tabella del catalogo pg_shadow in un file testo per usarla dopo. Per vedere la password crittografata usiamo la query:

```
select * from pg_shadow;
```

possiamo selezionare solo la user e la pass se ci troviamo via SSH:

```
select usename, passwd from pg_shadow;
```

Aggiungere al file /etc/pgbouncer/userlist.txt la password criptata: di seguito un esempio non usarla ;-)

```
"pgbouncer" "SCRAM-SHA-
256$4096:DNb4X/kg0z+I35kdh0yVIQ==$guxMeLharILyW/NP2fdD2Ue/rT7AZ6e+jicphUS+jl4=:kvbsB1C2
NrO6wKSXH0vEgleJ85VC7oTwSVd45acUz+g="
```

Facciamo una copia del file originale:

```
sudo mv /etc/pgbouncer/pgbouncer.ini /etc/pgbouncer/pgbouncer.ini.orig
```

Creaiamo un nuovo file

```
sudo vim /etc/pgbouncer/pgbouncer.ini
```

Aggiungere al file /etc/pgbouncer/pgbouncer.ini di ogni nodo DB del cluster le seguenti voci

Cambiare l'IP del nodo DBServerXX evidenziato in giallo

```
[databases]
* = host=10.116.82.32 port=5432
[users]
# per ora lasciato vuoto...
[pgbouncer]
logfile = /var/log/postgresql/pgbouncer.log
pidfile = /var/run/postgresql/pgbouncer.pid
unix_socket_dir = /var/run/postgresql
admin_users = postgres
listen_addr = *
listen_port = 6432
;; any, trust, plain, md5, cert, hba, pam
auth_type = md5
auth_file = /etc/pgbouncer/userlist.txt
auth_user = pgbouncer
auth_query = SELECT p_user, p_password FROM public.lookup($1)
;; When server connection is released back to pool:
;;   session      - after client disconnects (default)
;;   transaction  - after transaction finishes
;;   statement    - after statement finishes
pool_mode = session
# by Pietro Cornelio
# How to use prepared statements with session pooling?
# In session pooling mode, the reset query must clean old prepared statements. This can be
achieved by
server_reset_query = DISCARD ALL;
;; Total number of clients that can connect
max_client_conn = 100
;; Default pool size.  20 is good number when transaction pooling
;; is in use, in session pooling it needs to be the number of
;; max clients you want to handle at any moment
default_pool_size = 100
# Risolve l'errore di connessione via JDBC by Pietro Cornelio
ignore_startup_parameters = extra_float_digits
```

Avviamo il servizio

```
sudo systemctl start pgbouncer
```

Passiamo all'utente postgres

```
pietro@dbserver01:~$ sudo su – postgres
```

Se facciamo modifiche a /etc/pgbouncer/pgbouncer.ini possiamo aggiornare l'agent di PgBouncer senza restartare il servizio e perdere le connessioni in produzione:

```
pgbouncer -R -d /etc/pgbouncer/pgbouncer.ini
```

Colleghiamoci a PostgreSQL dal nodo leader

```
pietro@dbserver01:~$ psql -h 127.0.0.1 -U postgres -d dvdrental -p 5432

CREATE DATABASE dvdrental;
```

Restoriamo il DB

```
pietro@dbserver01:~$ pg_restore -U postgres -d dvdrental dvdrental.tar
```

Colleghiamoci al DB appena importato

```
postgres=# \c dvdrental
You are now connected to database "dvdrental" as user "postgres".
```

Elenchiamo tutte le tabelle

```
dvdrental=# \dt
             List of relations
 Schema |     Name      | Type  |  Owner
--------+---------------+-------+----------
 public | actor         | table | postgres
 public | address       | table | postgres
 public | category      | table | postgres
 public | city          | table | postgres
 public | country       | table | postgres
 public | customer      | table | postgres
 public | film          | table | postgres
 public | film_actor    | table | postgres
 public | film_category | table | postgres
 public | inventory     | table | postgres
 public | language      | table | postgres
 public | payment       | table | postgres
 public | rental        | table | postgres
 public | staff         | table | postgres
 public | store         | table | postgres
(15 rows)
```

Facciamo una verifica di connessione al DB attraverso PgBouncer dal nodo dbserver01

```
pietro@dbserver01:~$ psql -h 127.0.0.1 -U pgbouncer -d dvdrental -p 6432
Password for user pgbouncer:
psql (14.7 - Percona Distribution (Ubuntu 2:14.7-1.jammy))
Type "help" for help.

dvdrental=> \dt
             List of relations
 Schema |     Name      | Type  |  Owner
--------+---------------+-------+----------
 public | actor         | table | postgres
 public | address       | table | postgres
 public | category      | table | postgres
 public | city          | table | postgres
 public | country       | table | postgres
 public | customer      | table | postgres
 public | film          | table | postgres
 public | film_actor    | table | postgres
 public | film_category | table | postgres
 public | inventory     | table | postgres
 public | language      | table | postgres
 public | payment       | table | postgres
 public | rental        | table | postgres
 public | staff         | table | postgres
 public | store         | table | postgres
(15 rows)

dvdrental=>
```

Possiamo abilitare il servizio di PgBouncer sui 3 nodi DBServerXX

```
sudo systemctl enable percona-pgbouncer.service
```

Sul nodo client-apps installiamo il client di PostgreSQL 14.x

```
sudo apt-get install -y postgresql-client
pietro@client-apps:~$ psql --version
psql (PostgreSQL) 14.8 (Ubuntu 14.8-0ubuntu0.22.04.1)
```

Ora proviamo la connessione al DB dvdrental da remoto dal nodo client-apps verso dbserver01 passando per PgBouncer

```
pietro@client-apps:~$ psql -h dbserver01 -U pgbouncer -d dvdrental -p 6432
Password for user pgbouncer:
psql (14.8 (Ubuntu 14.8-0ubuntu0.22.04.1), server 14.7 - Percona Distribution (Ubuntu 2:14.7-
1.jammy))
Type "help" for help.
```

Elenchiamo tutte le tabelle del DB

```
dvdrental=> \dt
             List of relations
 Schema |     Name      | Type  |  Owner
--------+---------------+-------+----------
 public | actor         | table | postgres
 public | address       | table | postgres
 public | category      | table | postgres
 public | city          | table | postgres
 public | country       | table | postgres
 public | customer      | table | postgres
 public | film          | table | postgres
 public | film_actor    | table | postgres
 public | film_category | table | postgres
 public | inventory     | table | postgres
 public | language      | table | postgres
 public | payment       | table | postgres
 public | rental        | table | postgres
 public | staff         | table | postgres
 public | store         | table | postgres
(15 rows)
```

Ora proviamo la connessione al DB dvdrental da remoto dal nodo client-apps verso dbserver02 passando per PgBouncer

```
pietro@client-apps:~$ psql -h dbserver02 -U pgbouncer -d dvdrental -p 6432
psql (14.8 (Ubuntu 14.8-0ubuntu0.22.04.1), server 14.7 - Percona Distribution (Ubuntu 2:14.7-
1.jammy))
Type "help" for help.

dvdrental=>
```

Ora proviamo la connessione al DB dvdrental da remoto dal nodo client-apps verso dbserver03 passando per PgBouncer

```
pietro@client-apps:~$ psql -h dbserver03 -U pgbouncer -d dvdrental -p 6432
psql (14.8 (Ubuntu 14.8-0ubuntu0.22.04.1), server 14.7 - Percona Distribution (Ubuntu
2:14.7-1.jammy))
Type "help" for help.

dvdrental=>
```

Installazione HAProxy sui 2 nodi

HAProxy verso PgBouncer sui 3 nodi DBServer01-02-03 porta 6432

apt install haproxy
cd /etc/haproxy/
mv haproxy.cfg haproxy.cfg.ORIG
vim /etc/haproxy/haproxy.cfg

Configurazione di HAProxy sui 2 nodi haproxy01 e haproxy02
Aggiungere le seguenti istruzioni ai 2 nodi HAProxy:

```
global
     log 127.0.0.1   local2
     log /dev/log    local0
     log /dev/log    local1 notice
     chroot /var/lib/haproxy
     stats socket /run/haproxy/admin.sock mode 660 level admin expose-fd listeners
     stats timeout 30s
     user haproxy
     group haproxy
     maxconn 4000
     daemon

defaults
    mode                    tcp
    log                     global
    option                  tcplog
    retries                 3
    timeout queue           1m
    timeout connect         10s
    timeout client          1m
    timeout server          1m
    timeout check           10s
    maxconn                 3000

listen stats
    mode http
    bind *:7000
    stats enable
    stats uri /

listen primary
    # VIP
    bind 10.116.82.38:5000
    bind *:5000
    option httpchk OPTIONS /primary
    http-check expect status 200
    default-server inter 3s fall 3 rise 2 on-marked-down shutdown-sessions
    server dbserver01 10.116.82.32:6432 maxconn 100 check port 8008
    server dbserver02 10.116.82.33:6432 maxconn 100 check port 8008
    server dbserver03 10.116.82.34:6432 maxconn 100 check port 8008

listen standby
    # VIP
    bind 10.116.82.38:5001
    bind *:5001
    balance roundrobin
    option httpchk OPTIONS /replica
    http-check expect status 200
    default-server inter 3s fall 3 rise 2 on-marked-down shutdown-sessions
    server dbserver01 10.116.82.32:6432 maxconn 100 check port 8008
    server dbserver02 10.116.82.33:6432 maxconn 100 check port 8008
    server dbserver03 10.116.82.34:6432 maxconn 100 check port 8008
```

Nota ci sono due sezioni: **primary**, che utilizza la porta 5000, e **standby**, che utilizza la porta 5001.
Tutti e tre i nodi DBServerXX sono inclusi in entrambe le sezioni: questo perché sono tutti potenziali candidati per essere primari o secondari.
Affinché HAproxy sappia quale ruolo ha attualmente ciascun nodo, invierà una richiesta HTTP alla porta 8008 del nodo quindi l'agent Patroni risponderà con lo stato del nodo.
Patroni fornisce un supporto API REST integrato per il monitoraggio del controllo dello stato che si integra perfettamente con Haproxy.
Verifichiamo la chiamata API REST con il curl e jq posizionandoci su un nodo haproxy
Nota importante: Puntare sempre al nodo leader…
```
curl -s http://dbserver01:8008 |jq -r
```

```
pietro@haproxy01:~$ curl -s http://dbserver01:8008 |jq -r
{
  "state": "running",
  "postmaster_start_time": "2023-06-15 09:58:11.666992+02:00",
  "role": "master",
  "server_version": 140007,
  "xlog": {
    "location": 90607824
  },
  "timeline": 4,
  "replication": [
    {
      "usename": "replicator",
      "application_name": "dbserver02",
      "client_addr": "10.116.82.33",
      "state": "streaming",
      "sync_state": "async",
      "sync_priority": 0
    },
    {
      "usename": "replicator",
      "application_name": "dbserver03",
      "client_addr": "10.116.82.34",
      "state": "streaming",
      "sync_state": "async",
      "sync_priority": 0
    }
  ],
  "dcs_last_seen": 1686818373,
  "database_system_identifier": "7243829706365300703",
  "patroni": {
    "version": "3.0.1",
    "scope": "postgres"
  }
}
```

Piccolo tuning:
Editare lo script di avvio del servizio haproxy /usr/lib/systemd/system/haproxy.service, aggiungendo la voce LimitNOFILE=6000 alla sezione service come segue:
```
[Service]
LimitNOFILE=6000
```
Run the following command to reload the system daemon:
systemctl daemon-reload
Avviare il servizio HAProxy
systemctl start haproxy
systemctl status haproxy
Abilitare il servizio HAProxy all'avvio:
systemctl enable haproxy

Verificare il massimo numero di files aperti dal processo HAProxy:

- cat /proc/<haproxy pid>/limits|grep open
  esempio:
- cat /proc/2893/limits|grep open
- Max open files 20037 20037 files

Verifica pre-finale

Verifichiamo che la connessione al cluster PostgreSQL Server tramite i **2 nodi HAProxy** avvenga correttamente, eseguendo i seguenti comandi su un host client DB in cui è disponibile "psql".

```
psql -h <haproxy1 ip address> -p 5000 -d postgres -U postgres
```

Connessione al DB dvdrental via PgBouncer (sul leader DBServer01) attraverso HAProxy01

```
pietro@client-apps:~$ psql -h haproxy01 -U pgbouncer -d dvdrental -p 5000
Password for user pgbouncer:
psql (14.8 (Ubuntu 14.8-0ubuntu0.22.04.1), server 14.7 - Percona Distribution (Ubuntu
2:14.7-1.jammy))
Type "help" for help.

dvdrental=> exit
```

Connessione al DB dvdrental via PgBouncer (sul leader DBServer01) attraverso HAProxy02

```
pietro@client-apps:~$ psql -h haproxy02 -U pgbouncer -d dvdrental -p 5000
Password for user pgbouncer:
psql (14.8 (Ubuntu 14.8-0ubuntu0.22.04.1), server 14.7 - Percona Distribution (Ubuntu
2:14.7-1.jammy))
Type "help" for help.
```

Listiamo le tabelle del DB dvdrental

```
dvdrental-> \dt
              List of relations
 Schema |     Name      | Type  |  Owner
--------+---------------+-------+----------
 public | actor         | table | postgres
 public | address       | table | postgres
 public | category      | table | postgres
 public | city          | table | postgres
 public | country       | table | postgres
 public | customer      | table | postgres
 public | film          | table | postgres
 public | film_actor    | table | postgres
 public | film_category | table | postgres
 public | inventory     | table | postgres
 public | language      | table | postgres
 public | payment       | table | postgres
 public | rental        | table | postgres
 public | staff         | table | postgres
 public | store         | table | postgres
(15 rows)
```

Lo ricordo ancora, ci sono due importanti sezioni di ascolto nella configurazione di haproxy:
primary utilizza la **porta 5000** per le richieste di query di lettura/scrittura (atterrano sempre sul nodo leader)
standby utilizzando la **porta 5001** solo per le richieste di query di lettura (atterrano sui nodi replica)
Tutti e tre i nodi sono inclusi in entrambe le sezioni: questo perché tutti i server di database sono potenziali candidati per essere primari o secondari. Patroni fornisce un supporto API REST integrato per il monitoraggio del controllo dello stato che funziona perfettamente con HAProxy.
HAProxy invierà una richiesta rest HTTP alla porta 8008 degli agent patroni per sapere quale ruolo ha attualmente ciascun nodo.

Eseguiamo 4 richieste di lettura per verificare che il meccanismo "round robin" di accesso ai nodi replica PostgreSQL di HAProxy funzioni come previsto:

```
pietro@client-apps:~$ psql -U pgbouncer -h haproxy01 -p 5001 -d postgres -t -c "select
inet_server_addr()"
 10.116.82.33

pietro@client-apps:~$ psql -U pgbouncer -h haproxy01 -p 5001 -d postgres -t -c "select
inet_server_addr()"
 10.116.82.34

pietro@client-apps:~$ psql -U pgbouncer -h haproxy01 -p 5001 -d postgres -t -c "select
inet_server_addr()"
 10.116.82.33

pietro@client-apps:~$ psql -U pgbouncer -h haproxy01 -p 5001 -d postgres -t -c "select
inet_server_addr()"
 10.116.82.34
```

Round-Robin correttamente eseguito sui nodi replica…

Ora testiamo la richiesta sulla porta 5000 vedremo se sarà reindirizzata al nodo Leader del cluster PostgreSQL:

```
psql -U pgbouncer -h haproxy01 -p 5000 -d postgres -t -c "select inet_server_addr()"
Password for user pgbouncer:
 10.116.82.32

psql -U pgbouncer -h haproxy02 -p 5000 -d postgres -t -c "select inet_server_addr()"
Password for user pgbouncer:
 10.116.82.32
```

Se non si vuole inserire la password ogni volta settare una var di ambiente al volo:
pietro@client-apps:~$ export PGPASSWORD='password.2023'

VIP: Installiamo Keepalived sui nodi HAProxyXX

sudo apt -y install keepalived
sudo systemctl stop keepalived
sudo systemctl disable keepalived

Aggiungere le linee seguenti al file /etc/sysctl.conf (alla fine):
net.ipv4.ip_nonlocal_bind = 1
net.ipv4.ip_forward = 1
nano /etc/sysctl.conf
Carichiamo la configurazione del kernel
root@haproxy01:/home/pietro# /sbin/sysctl -p
net.ipv4.ip_nonlocal_bind = 1
net.ipv4.ip_forward = 1
root@haproxy02:/home/pietro# /sbin/sysctl -p
net.ipv4.ip_nonlocal_bind = 1
net.ipv4.ip_forward = 1

Configurazione VIP (ip condiviso del cluster HAProxy)
nano /etc/keepalived/keepalived.conf
Evidenziato in giallo il VIP e l'interfaccia di rete

```
vrrp_script chk_haproxy {
        script "pkill -0 haproxy"
        interval 5
        weight -4
        fall 2
        rise 1
}

vrrp_script chk_lb {
        script "pkill -0 keepalived"
        interval 1
        weight 6
        fall 2
        rise 1
}

vrrp_script chk_servers {
        script "echo 'GET /are-you-ok' | nc 127.0.0.1 7000 | grep -q '200 OK'"
        interval 2
        weight 2
        fall 2
        rise 2
}

vrrp_instance vrrp_1 {
        interface ens192
        state MASTER
        virtual_router_id 51
        priority 101
        virtual_ipaddress_excluded {
                10.116.82.38
        }
        track_interface {
                ens192 weight -2
        }
        track_script {
                chk_haproxy
                chk_lb
        }
}
```

Avviamo il servizio Keepalived su HAProxy01

```
root@haproxy01:/home/pietro# systemctl start keepalived
root@haproxy01:/home/pietro# systemctl status keepalived
● keepalived.service - Keepalive Daemon (LVS and VRRP)
     Loaded: loaded (/lib/systemd/system/keepalived.service; enabled; preset: enabled)
     Active: active (running) since Tue 2023-06-13 17:40:57 CEST; 17s ago
   Main PID: 12230 (keepalived)
      Tasks: 2 (limit: 4637)
     Memory: 3.3M
        CPU: 417ms
     CGroup: /system.slice/keepalived.service
             ├─12230 /usr/sbin/keepalived --dont-fork
             └─12231 /usr/sbin/keepalived --dont-fork

giu 13 17:40:57 haproxy01 Keepalived_vrrp[12231]: (vrrp_1) No VIP specified; at least one is sensible
giu 13 17:40:57 haproxy01 Keepalived_vrrp[12231]: (vrrp_1) Ignoring track_interface ens192 since own
interface
giu 13 17:40:57 haproxy01 Keepalived_vrrp[12231]: (vrrp_1) No VIP specified; at least one is sensible
giu 13 17:40:57 haproxy01 Keepalived_vrrp[12231]: Warning - script chk_servers is not used
giu 13 17:40:57 haproxy01 Keepalived_vrrp[12231]: Registering gratuitous ARP shared channel
giu 13 17:40:57 haproxy01 Keepalived_vrrp[12231]: (vrrp_1) Entering BACKUP STATE (init)
giu 13 17:40:57 haproxy01 Keepalived_vrrp[12231]: VRRP_Script(chk_haproxy) succeeded
giu 13 17:40:57 haproxy01 Keepalived_vrrp[12231]: VRRP_Script(chk_lb) succeeded
giu 13 17:40:57 haproxy01 Keepalived_vrrp[12231]: (vrrp_1) Changing effective priority from 101 to 107
giu 13 17:41:00 haproxy01 Keepalived_vrrp[12231]: (vrrp_1) Entering MASTER STATE
```

Avviamo il servizio Keepalived su HAProxy02

```
root@haproxy02:/home/pietro# systemctl start keepalived
root@haproxy02:/home/pietro# systemctl status keepalived
● keepalived.service - Keepalive Daemon (LVS and VRRP)
     Loaded: loaded (/lib/systemd/system/keepalived.service; enabled; preset: enabled)
     Active: active (running) since Tue 2023-06-13 17:41:07 CEST; 16s ago
   Main PID: 11300 (keepalived)
      Tasks: 2 (limit: 4637)
     Memory: 3.3M
        CPU: 387ms
     CGroup: /system.slice/keepalived.service
             ├─11300 /usr/sbin/keepalived --dont-fork
             └─11301 /usr/sbin/keepalived --dont-fork

giu 13 17:41:07 haproxy02 Keepalived_vrrp[11301]: SECURITY VIOLATION - scripts are being executed but
script_security not enabled.
giu 13 17:41:07 haproxy02 Keepalived_vrrp[11301]: (vrrp_1) No VIP specified; at least one is sensible
giu 13 17:41:07 haproxy02 Keepalived_vrrp[11301]: (vrrp_1) Ignoring track_interface ens192 since own
interface
giu 13 17:41:07 haproxy02 Keepalived_vrrp[11301]: (vrrp_1) No VIP specified; at least one is sensible
giu 13 17:41:07 haproxy02 Keepalived_vrrp[11301]: Warning - script chk_servers is not used
giu 13 17:41:07 haproxy02 Keepalived_vrrp[11301]: Registering gratuitous ARP shared channel
giu 13 17:41:07 haproxy02 Keepalived_vrrp[11301]: (vrrp_1) Entering BACKUP STATE (init)
giu 13 17:41:07 haproxy02 Keepalived_vrrp[11301]: VRRP_Script(chk_haproxy) succeeded
giu 13 17:41:07 haproxy02 Keepalived_vrrp[11301]: VRRP_Script(chk_lb) succeeded
giu 13 17:41:07 haproxy02 Keepalived_vrrp[11301]: (vrrp_1) Changing effective priority from 101 to 107
```

Verifichiamo la presenza del VIP sul primo nodo HAProxy01

```
root@haproxy01:/home/pietro# ip addr show ens192

2: ens192: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP group default qlen 1000
    link/ether 00:50:56:99:33:c5 brd ff:ff:ff:ff:ff:ff
    altname enp11s0
    inet 10.116.82.30/24 brd 10.116.82.255 scope global ens192
       valid_lft forever preferred_lft forever
    inet 10.116.82.38/32 scope global ens192
       valid_lft forever preferred_lft forever
    inet6 fe80::250:56ff:fe99:33c5/64 scope link
       valid_lft forever preferred_lft forever
```

Verifichiamo il corretto funzionamento del VIP

Connessione a uno dei nodi PostgrSQL replica (round-robin)

```
pietro@client-apps:~$ psql -U pgbouncer -h 10.116.82.38 -p 5001 -d postgres -t -c "select
inet_server_addr()"
 10.116.82.33

pietro@client-apps:~$ psql -U pgbouncer -h 10.116.82.38 -p 5001 -d postgres -t -c "select
inet_server_addr()"
 10.116.82.34

pietro@client-apps:~$ psql -U pgbouncer -h 10.116.82.38 -p 5001 -d postgres -t -c "select
inet_server_addr()"
 10.116.82.33

pietro@client-apps:~$ psql -U pgbouncer -h 10.116.82.38 -p 5001 -d postgres -t -c "select
inet_server_addr()"
 10.116.82.34
```

Come si vede dalle connessioni a HAProxy, attraverso l'IP condiviso (VIP), viene applicato l'algoritmo round-robin in modo da far connettere il client psql alternativamente (per le query in sola lettura sulla porta 5001) ai 2 nodi replica disponibili.

Connessione al nodo PostgreSQL leader

```
pietro@client-apps:~$ psql -U pgbouncer -h 10.116.82.38 -p 5000 -d postgres -t -c
"select inet_server_addr()"
 10.116.82.32
```

Naturalmente per tutte le query di lettura e scrittura se puntiamo alla porta 5000 di HAProxy atterreremo sempre sul nodo Leader. Le applicazioni client scritte bene e con i contro caxx, useranno la porta 5001 per tutte le query in sola lettura in modo che esse siano distribuite alternativamente ai nodi replica secondo l'algoritmo round-robin mentre per le query di scrittura e modifica saranno indirizzate alla porta 5000.

Creiamo un DB di esempio per i nostri test

Scarichiamo il DB dvdrental.zip portandolo sul nodo leader dal portale:
https://www.postgresqltutorial.com/postgresql-getting-started/postgresql-sample-database/
direttamente con

```
wget https://www.postgresqltutorial.com/wp-content/uploads/2019/05/dvdrental.zip
```

Installiamo il pacchetto unzip

```
sudo apt install unzip
unzip dvdrental.zip
sudo su – postgres
```

Restoriamo il db dvdrental

```
pg_restore -U postgres -d dvdrental dvdrental.tar
```

Verifichiamo il DB è stato restorato/importato correttamente

Possiamo farlo a bordo del nodo Leader:

```
pietro@dbserver01:~$ psql -h 127.0.0.1 -U pgbouncer -d dvdrental -p 6432
Password for user pgbouncer:

psql (14.7 - Percona Distribution (Ubuntu 2:14.7-1.jammy))
Type "help" for help.
```

```
dvdrental=> \dt
             List of relations
 Schema |     Name      | Type  |  Owner
--------+---------------+-------+----------
 public | actor         | table | postgres
 public | address       | table | postgres
 public | category      | table | postgres
 public | city          | table | postgres
 public | country       | table | postgres
 public | customer      | table | postgres
 public | film          | table | postgres
 public | film_actor    | table | postgres
 public | film_category | table | postgres
 public | inventory     | table | postgres
 public | language      | table | postgres
 public | payment       | table | postgres
 public | rental        | table | postgres
 public | staff         | table | postgres
 public | store         | table | postgres
(15 rows)

dvdrental=>
```

Ora proviamo dal nodo client-apps passando per il VIP sul bilanciatore HAProxy

```
pietro@client-apps:~$ psql -h 10.116.82.38 -U pgbouncer -d dvdrental -p 5000
Password for user pgbouncer:
psql (14.8 (Ubuntu 14.8-0ubuntu0.22.04.1), server 14.7 - Percona Distribution (Ubuntu 2:14.7-
1.jammy))
Type "help" for help.

dvdrental=> \dt
             List of relations
 Schema |     Name      | Type  |  Owner
--------+---------------+-------+----------
 public | actor         | table | postgres
 public | address       | table | postgres
 public | category      | table | postgres
 public | city          | table | postgres
 public | country       | table | postgres
 public | customer      | table | postgres
 public | film          | table | postgres
 public | film_actor    | table | postgres
 public | film_category | table | postgres
 public | inventory     | table | postgres
 public | language      | table | postgres
 public | payment       | table | postgres
 public | rental        | table | postgres
 public | staff         | table | postgres
 public | store         | table | postgres
(15 rows)
```

Psql

```
\dt elenca tabelle
\du elenca utenti
\dv elenca viste
\ds elenca sequenze
```

Ora diamo i privilegi all'utente pgbouncer sul DB dvdrental e i suoi oggetti

Ci colleghiamo direttamente a bordo del nodo leader dbserver01

```
pietro@dbserver01:~$ psql -h 127.0.0.1 -U postgres -d dvdrental -p 5432
Password for user postgres:
psql (14.8 (Ubuntu 14.8-0ubuntu0.22.04.1), server 14.7 - Percona Distribution (Ubuntu 2:14.7-
1.jammy))
Type "help" for help.

postgres=#
GRANT ALL PRIVILEGES ON DATABASE dvdrental TO pgbouncer;
GRANT ALL PRIVILEGES ON ALL TABLES IN SCHEMA public TO pgbouncer;
GRANT ALL PRIVILEGES ON ALL SEQUENCES IN SCHEMA public TO pgbouncer;
```

Ora su tutte le tabelle

```
dvdrental=# ALTER TABLE actor OWNER TO pgbouncer;
dvdrental=# ALTER TABLE address OWNER TO pgbouncer;
dvdrental=# ALTER TABLE category OWNER TO pgbouncer;
dvdrental=# ALTER TABLE city OWNER TO pgbouncer;
dvdrental=# ALTER TABLE country OWNER TO pgbouncer;
dvdrental=# ALTER TABLE customer OWNER TO pgbouncer;
dvdrental=# ALTER TABLE film OWNER TO pgbouncer;
dvdrental=# ALTER TABLE film_actor OWNER TO pgbouncer;
dvdrental=# ALTER TABLE film_category OWNER TO pgbouncer;
dvdrental=# ALTER TABLE inventory OWNER TO pgbouncer;
dvdrental=# ALTER TABLE language OWNER TO pgbouncer;
dvdrental=# ALTER TABLE payment OWNER TO pgbouncer;
dvdrental=# ALTER TABLE rental OWNER TO pgbouncer;
dvdrental=# ALTER TABLE staff OWNER TO pgbouncer;
dvdrental=# ALTER TABLE store OWNER TO pgbouncer;
dvdrental=# ALTER VIEW actor_info OWNER TO pgbouncer;
dvdrental=# ALTER VIEW customer_list OWNER TO pgbouncer;
dvdrental=# ALTER VIEW film_list OWNER TO pgbouncer;
dvdrental=# ALTER VIEW nicer_but_slower_film_list OWNER TO pgbouncer;
dvdrental=# ALTER VIEW sales_by_film_category OWNER TO pgbouncer;
dvdrental=# ALTER VIEW sales_by_store OWNER TO pgbouncer;
dvdrental=# ALTER VIEW staff_list OWNER TO pgbouncer;

dvdrental=# \dt
             List of relations
 Schema |     Name      | Type  |   Owner
--------+---------------+-------+-----------
 public | actor         | table | pgbouncer
 public | address       | table | pgbouncer
 public | category      | table | pgbouncer
 public | city          | table | pgbouncer
 public | country       | table | pgbouncer
 public | customer      | table | pgbouncer
 public | film          | table | pgbouncer
 public | film_actor    | table | pgbouncer
 public | film_category | table | pgbouncer
 public | inventory     | table | pgbouncer
 public | language      | table | pgbouncer
 public | payment       | table | pgbouncer
 public | rental        | table | pgbouncer
 public | staff         | table | pgbouncer
 public | store         | table | pgbouncer
(15 rows)
```

Accesso alla dashboard di HAProxy via web

`http://shared-ip:7000/`

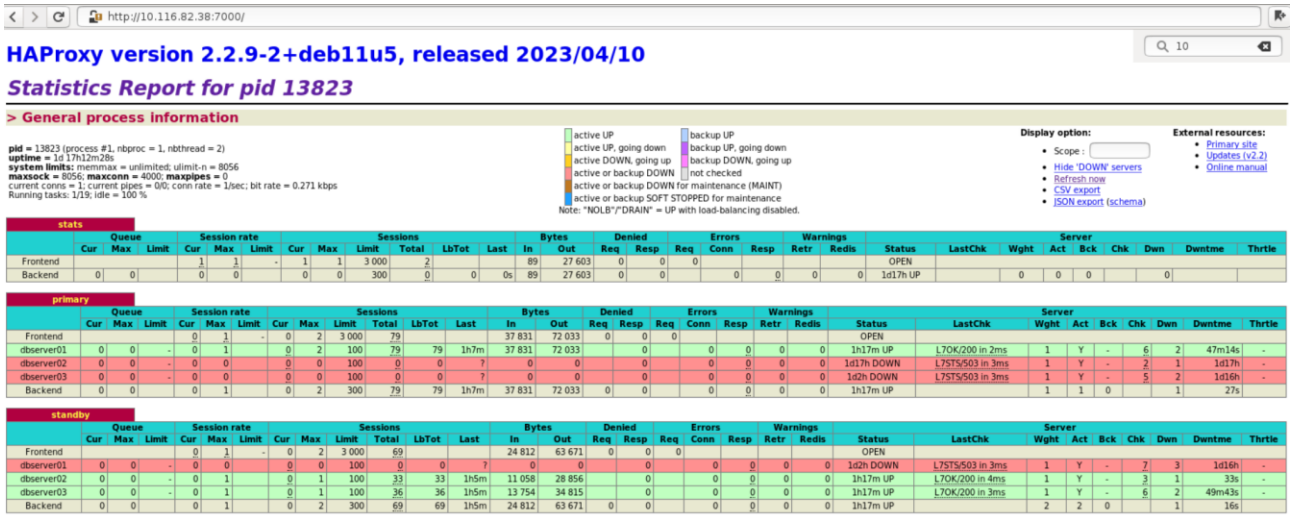
http://10.116.82.38:7000/

Sul nodo client-apps installiamo il browser web leggero Midori in quanto la remotizzazione X11 con Firefox è molto pesante.

```
sudo snap install midori
```

Lanciamo il browser Midori e poi connettiamoci al VIP di HAProxy su porta 7000

```
XAUTHORITY=$HOME/.Xauthority /snap/bin/midori
```



Installiamo un ottimo client di amministrazione grafica di PostgreSQL

```
pietro@client-apps:~$ sudo snap install dbeaver-ce
```

Esecuzione

```
pietro@client-apps:~$ XAUTHORITY=$HOME/.Xauthority /snap/bin/dbeaver-ce
```

Siccome il software dbeaver è pesante usarlo da remoto su X11 vis ssh usiamo l'ottimo e leggero SQLWorkbench scritto in Java:

sudo apt install openjdk-11-jre

wget https://www.sql-workbench.eu/Workbench-Build129-with-optional-libs.zip

mkdir SQLworkbench

mv Workbench-Build129-with-optional-libs.zip ./SQLworkbench

cd SQLworkbench/

unzip Workbench-Build129-with-optional-libs.zip

./sqlworkbench.sh

Naturalmente se abbiamo un nodo desktop che può accedere direttamente al bilanciatore HAProxy al VIP, installiamo i software su questo nodo desktop senza preoccuparci di remotizzare su protocollo X11 via SSH i software.

Testiamo l'HA del cluster PostgreSQL

Passaggio (switch) manuale
Ad esempio, se bisogna manutenere il nodo leader e quindi servono tempi di inattività (naturalmente pianificati) per eseguire l'attività di manutenzione. Patroni fornisce il comando comando di switchover per passare manualmente dal nodo leader.
Prima elenchiamo lo stato del cluster

```
pietro@dbserver01:~$ patronictl -c /etc/patroni/patroni.yml list

+ Cluster: postgres --------+---------+---------+----+-----------+
| Member     | Host         | Role    | State   | TL | Lag in MB |
+------------+--------------+---------+---------+----+-----------+
| dbserver01 | 10.116.82.32 | Leader  | running |  1 |           |
| dbserver02 | 10.116.82.33 | Replica | running |  1 |         0 |
| dbserver03 | 10.116.82.34 | Replica | running |  1 |         0 |
+------------+--------------+---------+---------+----+-----------+
```

Il nodo **dbserver01** è attualmente il leader
Eseguire il seguente comando sul nodo leader corrente:
`sudo patronictl -c /etc/patroni/patroni.yml switchover`
Patroni chiede il nome dell'attuale nodo primario e quindi il nodo che dovrebbe assumere il ruolo di primario commutato. È inoltre possibile specificare l'ora in cui deve avvenire il passaggio. Per attivare immediatamente il processo, specificare ora il valore.
Di seguito la procedura completa:

```
pietro@dbserver01:~$ sudo patronictl -c /etc/patroni/patroni.yml switchover
Current cluster topology
+ Cluster: postgres --------+---------+---------+----+-----------+
| Member     | Host         | Role    | State   | TL | Lag in MB |
+------------+--------------+---------+---------+----+-----------+
| dbserver01 | 10.116.82.32 | Leader  | running |  1 |           |
| dbserver02 | 10.116.82.33 | Replica | running |  1 |         0 |
| dbserver03 | 10.116.82.34 | Replica | running |  1 |         0 |
+------------+--------------+---------+---------+----+-----------+
Primary [dbserver01]: dbserver01
Candidate ['dbserver02', 'dbserver03'] []: dbserver03
When should the switchover take place (e.g. 2023-06-14T09:09 )  [now]: now
Are you sure you want to switchover cluster postgres, demoting current leader
dbserver01? [y/N]: y
2023-06-14 08:09:31.83191 Successfully switched over to "dbserver03"
+ Cluster: postgres --------+---------+---------+----+-----------+
| Member     | Host         | Role    | State   | TL | Lag in MB |
+------------+--------------+---------+---------+----+-----------+
| dbserver01 | 10.116.82.32 | Replica | stopped |    |   unknown |
| dbserver02 | 10.116.82.33 | Replica | running |  1 |         0 |
| dbserver03 | 10.116.82.34 | Leader  | running |  1 |           |
+------------+--------------+---------+---------+----+-----------+
pietro@dbserver01:~$ patronictl -c /etc/patroni/patroni.yml list
+ Cluster: postgres --------+---------+---------+----+-----------+
| Member     | Host         | Role    | State   | TL | Lag in MB |
+------------+--------------+---------+---------+----+-----------+
| dbserver01 | 10.116.82.32 | Replica | running |  2 |         0 |
| dbserver02 | 10.116.82.33 | Replica | running |  2 |         0 |
| dbserver03 | 10.116.82.34 | Leader  | running |  2 |           |
+------------+--------------+---------+---------+----+-----------+
```

Come si può vedere chiaramente (in rosso le risposte) il cluster ha obbedito correttamente elevando il nodo dbserver3 a leader.
Il nodo DBServer01 in manutenzione quindi spegniamo il servizio patroni in modo che esso come membro del cluster vada in stato di "stopped" vedi di seguito:

```
pietro@dbserver01:~$ sudo systemctl stop patroni.service
[sudo] password for pietro:
pietro@dbserver01:~$ patronictl -c /etc/patroni/patroni.yml list
+ Cluster: postgres --------+---------+---------+----+-----------+
| Member     | Host         | Role    | State   | TL | Lag in MB |
+------------+--------------+---------+---------+----+-----------+
| dbserver01 | 10.116.82.32 | Replica | stopped |    |   unknown |
| dbserver02 | 10.116.82.33 | Replica | running |  2 |         0 |
| dbserver03 | 10.116.82.34 | Leader  | running |  2 |           |
+------------+--------------+---------+---------+----+-----------+
```

Dopo l'attività di manutenzione al nodo dbserver01 (ex leader) per riportare il ruolo di leader al nodo dbserver01 basta semplicemente riavviare il servizio Patroni del nodo dbserver01, il nodo si ricongiungerà in automatico al cluster come nodo secondario.

```
pietro@dbserver01:~$ sudo systemctl start patroni.service
pietro@dbserver01:~$ patronictl -c /etc/patroni/patroni.yml list
+ Cluster: postgres --------+---------+---------+----+-----------+
| Member     | Host         | Role    | State   | TL | Lag in MB |
+------------+--------------+---------+---------+----+-----------+
| dbserver01 | 10.116.82.32 | Replica | running |  2 |         0 |
| dbserver02 | 10.116.82.33 | Replica | running |  2 |         0 |
| dbserver03 | 10.116.82.34 | Leader  | running |  2 |           |
+------------+--------------+---------+---------+----+-----------+
```

Naturalmente, possiamo lasciare come nodo leader dbserver03 rispetto alla configurazione iniziale in quanto ogni nodo del cluster è paritetico.

Passaggio (switch) automatico

Allora, provochiamo un guasto a un nodo e verifichiamo che lo switch automatico avvenga correttamente. Posizioniamoci sul nodo dbserver03 attualmente il leader e troviamo il pid del processo patroni:

```
pietro@dbserver03:~$ ps aux | grep -i patroni
postgres   18092  0.1  1.1 501252 45256 ?        Ssl  Jun12   4:06 /usr/bin/python3
/usr/bin/patroni /etc/patroni/patroni.yml
```

killiamo il processo di patroni sul nodo dbserver03 ed osserviamo cosa accade

```
sudo kill -9 18092 && sudo journalctl -u patroni.service -n 100 -f
```

```
Jun 14 08:45:26 dbserver03 systemd[1]: patroni.service: Main process exited, code=killed, status=9/KILL
Jun 14 08:45:26 dbserver03 systemd[1]: patroni.service: Failed with result 'signal'.
Jun 14 08:45:26 dbserver03 systemd[1]: patroni.service: Unit process 18105 (postgres) remains running after unit stopped.
Jun 14 08:45:26 dbserver03 systemd[1]: patroni.service: Unit process 18107 (postgres) remains running after unit stopped.
Jun 14 08:45:26 dbserver03 systemd[1]: patroni.service: Unit process 18109 (postgres) remains running after unit stopped.
Jun 14 08:45:26 dbserver03 systemd[1]: patroni.service: Unit process 18110 (postgres) remains running after unit stopped.
Jun 14 08:45:26 dbserver03 systemd[1]: patroni.service: Unit process 18111 (postgres) remains running after unit stopped.
Jun 14 08:45:26 dbserver03 systemd[1]: patroni.service: Unit process 90771 (postgres) remains running after unit stopped.
Jun 14 08:45:26 dbserver03 systemd[1]: patroni.service: Unit process 90772 (postgres) remains running after unit stopped.
Jun 14 08:45:26 dbserver03 systemd[1]: patroni.service: Unit process 90773 (postgres) remains running after unit stopped.
Jun 14 08:45:26 dbserver03 systemd[1]: patroni.service: Unit process 90784 (postgres) remains running after unit stopped.
Jun 14 08:45:26 dbserver03 systemd[1]: patroni.service: Unit process 93064 (postgres) remains running after unit stopped.
Jun 14 08:45:26 dbserver03 systemd[1]: patroni.service: Consumed 8min 15.956s CPU time.
Jun 14 08:45:26 dbserver03 systemd[1]: patroni.service: Scheduled restart job, restart counter is at 1.
Jun 14 08:45:26 dbserver03 systemd[1]: Stopped PostgreSQL high-availability manager.
Jun 14 08:45:26 dbserver03 systemd[1]: patroni.service: Consumed 8min 15.956s CPU time.
Jun 14 08:45:26 dbserver03 systemd[1]: patroni.service: Found left-over process 18105 (postgres) in control group while
starting unit. Ignoring.
Jun 14 08:45:26 dbserver03 systemd[1]: This usually indicates unclean termination of a previous run, or service
implementation deficiencies.
Jun 14 08:45:26 dbserver03 systemd[1]: patroni.service: Found left-over process 18107 (postgres) in control group while
starting unit. Ignoring.
Jun 14 08:45:26 dbserver03 systemd[1]: This usually indicates unclean termination of a previous run, or service
implementation deficiencies.
Jun 14 08:45:26 dbserver03 systemd[1]: patroni.service: Found left-over process 18109 (postgres) in control group while
starting unit. Ignoring.
Jun 14 08:45:26 dbserver03 systemd[1]: This usually indicates unclean termination of a previous run, or service
implementation deficiencies.
Jun 14 08:45:26 dbserver03 systemd[1]: patroni.service: Found left-over process 18110 (postgres) in control group while
starting unit. Ignoring.
Jun 14 08:45:26 dbserver03 systemd[1]: This usually indicates unclean termination of a previous run, or service
implementation deficiencies.
```

```
Jun 14 08:45:26 dbserver03 systemd[1]: patroni.service: Found left-over process 18111 (postgres) in control group while
starting unit. Ignoring.
Jun 14 08:45:26 dbserver03 systemd[1]: This usually indicates unclean termination of a previous run, or service
implementation deficiencies.
Jun 14 08:45:26 dbserver03 systemd[1]: patroni.service: Found left-over process 90771 (postgres) in control group while
starting unit. Ignoring.
Jun 14 08:45:26 dbserver03 systemd[1]: This usually indicates unclean termination of a previous run, or service
implementation deficiencies.
Jun 14 08:45:26 dbserver03 systemd[1]: patroni.service: Found left-over process 90772 (postgres) in control group while
starting unit. Ignoring.
Jun 14 08:45:26 dbserver03 systemd[1]: This usually indicates unclean termination of a previous run, or service
implementation deficiencies.
Jun 14 08:45:26 dbserver03 systemd[1]: patroni.service: Found left-over process 90773 (postgres) in control group while
starting unit. Ignoring.
Jun 14 08:45:26 dbserver03 systemd[1]: This usually indicates unclean termination of a previous run, or service
implementation deficiencies.
Jun 14 08:45:26 dbserver03 systemd[1]: patroni.service: Found left-over process 90784 (postgres) in control group while
starting unit. Ignoring.
Jun 14 08:45:26 dbserver03 systemd[1]: This usually indicates unclean termination of a previous run, or service
implementation deficiencies.
Jun 14 08:45:26 dbserver03 systemd[1]: patroni.service: Found left-over process 93064 (postgres) in control group while
starting unit. Ignoring.
Jun 14 08:45:26 dbserver03 systemd[1]: This usually indicates unclean termination of a previous run, or service
implementation deficiencies.
Jun 14 08:45:26 dbserver03 systemd[1]: Started PostgreSQL high-availability manager.
Jun 14 08:45:27 dbserver03 patroni[93911]: 2023-06-14 08:45:27,096 INFO: Selected new etcd server http://10.116.82.37:2379
Jun 14 08:45:27 dbserver03 patroni[93911]: 2023-06-14 08:45:27,103 INFO: No PostgreSQL configuration items changed, nothing
to reload.
Jun 14 08:45:27 dbserver03 patroni[93911]: 2023-06-14 08:45:27,108 INFO: establishing a new patroni connection to the
postgres cluster
Jun 14 08:45:27 dbserver03 patroni[93911]: 2023-06-14 08:45:27,124 INFO: Lock owner: dbserver03; I am dbserver03
Jun 14 08:45:27 dbserver03 patroni[93911]: 2023-06-14 08:45:27,133 INFO: Software Watchdog activated with 25 second timeout,
timing slack 15 seconds
Jun 14 08:45:27 dbserver03 patroni[93911]: 2023-06-14 08:45:27,139 INFO: no action. I am (dbserver03), the leader with the
lock
Jun 14 08:45:37 dbserver03 patroni[93911]: 2023-06-14 08:45:37,132 INFO: no action. I am (dbserver03), the leader with the
lock
```

Siccome abbiamo attivato il **watchdog** (software) il servizio patroni è stato riportato in vita senza creare danni al cluster.

Verifichiamo di nuovo il processo di patroni

```
pietro@dbserver03:~$ ps aux | grep -i patroni
postgres   93911  0.7  1.1 427412 44936 ?        Ssl  08:45   0:00 /usr/bin/python3
/usr/bin/patroni /etc/patroni/patroni.yml

pietro@dbserver01:~$ patronictl -c /etc/patroni/patroni.yml list
+ Cluster: postgres --------+---------+---------+----+-----------+
| Member     | Host         | Role    | State   | TL | Lag in MB |
+------------+--------------+---------+---------+----+-----------+
| dbserver01 | 10.116.82.32 | Replica | running |  2 |         0 |
| dbserver02 | 10.116.82.33 | Replica | running |  2 |         0 |
| dbserver03 | 10.116.82.34 | Leader  | running |  2 |           |
+------------+--------------+---------+---------+----+-----------+
```

Test di spegnimento/riavvio di un nodo del cluster

Proviamo a riavviare completamente il nodo **DBServer03** e vediamo cosa accade, dopo ci posizioniamo su DBServer01 per dare il comndando di stato del cluster:
come vediamo di seguito, immediatamente patroni elegge dbserver01 a leader:
Nota: ho dato il comando seguente immediatamente dopo aver riavviato dbserver03

```
pietro@dbserver01:~$ patronictl -c /etc/patroni/patroni.yml list
+ Cluster: postgres --------+---------+---------+----+-----------+
| Member     | Host         | Role    | State   | TL | Lag in MB |
+------------+--------------+---------+---------+----+-----------+
| dbserver01 | 10.116.82.32 | Leader  | running |  3 |           |
| dbserver02 | 10.116.82.33 | Replica | running |  2 |         0 |
| dbserver03 | 10.116.82.34 | Replica | stopped |    |   unknown |
+------------+--------------+---------+---------+----+-----------+
pietro@dbserver01:~$ patronictl -c /etc/patroni/patroni.yml list
+ Cluster: postgres --------+---------+---------+----+-----------+
| Member     | Host         | Role    | State   | TL | Lag in MB |
+------------+--------------+---------+---------+----+-----------+
| dbserver01 | 10.116.82.32 | Leader  | running |  3 |           |
| dbserver02 | 10.116.82.33 | Replica | running |  2 |         0 |
| dbserver03 | 10.116.82.34 | Replica | stopped |    |   unknown |
+------------+--------------+---------+---------+----+-----------+
pietro@dbserver01:~$ patronictl -c /etc/patroni/patroni.yml list
+ Cluster: postgres --------+---------+---------+----+-----------+
| Member     | Host         | Role    | State   | TL | Lag in MB |
+------------+--------------+---------+---------+----+-----------+
| dbserver01 | 10.116.82.32 | Leader  | running |  3 |           |
| dbserver02 | 10.116.82.33 | Replica | running |  3 |         0 |
+------------+--------------+---------+---------+----+-----------+
```

Ecco ricomparire correttamente DBServer03 come membro del cluster con ruolo replica

```
pietro@dbserver01:~$ patronictl -c /etc/patroni/patroni.yml list
+ Cluster: postgres --------+---------+---------+----+-----------+
| Member     | Host         | Role    | State   | TL | Lag in MB |
+------------+--------------+---------+---------+----+-----------+
| dbserver01 | 10.116.82.32 | Leader  | running |  3 |           |
| dbserver02 | 10.116.82.33 | Replica | running |  3 |         0 |
| dbserver03 | 10.116.82.34 | Replica | running |  2 |         0 |
+------------+--------------+---------+---------+----+-----------+
```

Come si è potuto vedere, l'azione e la gestione di elevazione di un nodo replica a leader di un cluster PostgreSQL con il framework Patroni + ETCD di fronte a un evento di fault è istantanea.

Nel prossimo capitolo vedremo come si comporta un cluster di questo tipo di fronte a un fault mentre ci sono attività concorrenti di query di lettura e scrittura da un client applicativo...

Testing di failover del cluster PostgreSQL durante letture e scritture

```
sudo apt install python3-psycopg2
git clone https://github.com/manwerjalil/pgscripts.git
ls -ls
chmod +x ~/pgscripts/pgsqlhatest.py
```

Configuriamo lo script python con i nostri parametri (VIP bilanciatore, user, etc.)

```
vim ~/pgscripts/pgsqlhatest.py
       host = "10.116.82.38"
       dbname = "postgres"
       user = "postgres"
       password = "postgres"
```

Creiamo gli oggetti DB che ci servono per fare il test su nodo leader

```
pietro@dbserver01:~$ psql -h 127.0.0.1 -U postgres -d postgres -p 5432
psql (14.7 - Percona Distribution (Ubuntu 2:14.7-1.jammy))
Type "help" for help.

postgres=# CREATE TABLE PGSQLHATEST (TM TIMESTAMP);
CREATE TABLE
postgres=# CREATE UNIQUE INDEX idx_pgsqlhatext ON pgsqlhatest (tm desc);
CREATE INDEX
postgres=#
```

Esecuzione del test per verificare le query di lettura e di scrittura

Apriamo 2 sessioni SSH al nodo client-apps

Lanciando i 2 comandi seguenti su 2 terminali separati

pietro@client-apps:~/pgscripts$ ./pgsqlhatest.py 5000 (scrittura)

pietro@client-apps:~/pgscripts$ ./pgsqlhatest.py 5001 (lettura con round-robin)

```
pietro@client-apps:~/pgscripts$ ./pgsqlhatest.py 5000
 Working with:   MASTER - 10.116.82.32
     Inserted: 2023-06-14 16:16:17.390071

 Working with:   MASTER - 10.116.82.32
     Inserted: 2023-06-14 16:16:18.399162

 Working with:   MASTER - 10.116.82.32
     Inserted: 2023-06-14 16:16:19.405285

 Working with:   MASTER - 10.116.82.32
     Inserted: 2023-06-14 16:16:20.410209

 Working with:   MASTER - 10.116.82.32
     Inserted: 2023-06-14 16:16:21.415365

pietro@client-apps:~/pgscripts$ ./pgsqlhatest.py 5001
 Working with:    REPLICA - 10.116.82.34
     Retrived: 2023-06-14 16:16:22.420410

 Working with:    REPLICA - 10.116.82.34
     Retrived: 2023-06-14 16:16:23.425579

 Working with:    REPLICA - 10.116.82.34
     Retrived: 2023-06-14 16:16:24.431371

 Working with:    REPLICA - 10.116.82.34
     Retrived: 2023-06-14 16:16:25.436172

 Working with:    REPLICA - 10.116.82.34
     Retrived: 2023-06-14 16:16:26.441315
```

Sul nodo DB Leader possiamo verificare le scritture, eseguendo ogni 1 secondo la query seguente

Riporto per brevità l'insert fatto dal programma python in questione, poi letta sotto sul nodo leader:

```
Working with:   MASTER - 10.116.82.32
     Inserted: 2023-06-14 16:17:06.642313

postgres@dbserver01:~$ watch -n 1 'psql -h localhost -p 5432 -c "SELECT tm FROM pgsqlhatest ORDER
BY tm DESC LIMIT 1;" '

Every 1.0s: psql -h localhost -p 5432 -c "SELECT tm FROM pgsqlhatest ORDER BY tm DESC LIMIT 1;"
dbserver01: Wed Jun 14 16:20:49 2023

             tm
----------------------------
 2023-06-14 16:17:06.642313
(1 row)
```

Allora, con questo meccanismo di lettura e scrittura possiamo provare a effettuare dei failover vedendo come il cluster, grazie a Patroni, l'archivio DCS e PgBouncer e il bilanciatore HAProxy perfettamente integrato in PostgreSQL, provvederà in completa autonomia a non far perdere nessuna query di scrittura in streaming sui nodi replica.
Ce ne accorgiamo dal timestamp scritto, facciamo una query dopo il fault per vedere tutte le voci inserite se coincidono con quelle a video della sessione su porta 5000:
postgres=# select * from pgsqlhatest;

Backup e Restore con l'utility pgBackRest

pgBackRest affronta molte delle funzionalità indispensabili che si vorrebbe in una soluzione di backup PostgreSQL. pgBackRest storicamente è stato scritto in Perl ma negli ultimi anni il progetto sta facendo progressi costanti convertendosi in codice C nativo.

Configurazione

Le direcotory seguenti sono state già create e settare dall'installer Percona, però per chiarezza espongo comunque i comandi che solitamente si danno dopo una installazione di pgBackRest classica:

```
sudo mkdir /var/lib/pgbackrest
sudo chown postgres:postgres /var/lib/pgbackrest
sudo mkdir -p /var/log/pgbackrest
sudo chown postgres:postgres /var/log/pgbackrest
```

Il primo passo è editare il file di configurazione **/etc/pgbackrest.conf** per la stanza su tutti i nodi DBServerXX. Una stanza definisce la configurazione di backup per uno specifico cluster di database PostgreSQL. Qualsiasi sezione di configurazione globale può essere sovrascritta per definire impostazioni specifiche della stanza. Editiamo il file di configurazione come segue:

```
postgres@dbserver01:~$ sudo vim /etc/pgbackrest.conf
[global]
#repo1-path=/var/lib/pgbackrest
repo1-path=/data/postgresql/backup
repo1-retention-full=2
process-max=2
log-level-console=info
log-level-file=debug
start-fast=y
stop-auto=y

[main]
pg1-path=/var/lib/postgresql/14/main
# pg1-host=pg-primary
pg1-host-user=postgres
pg1-port = 5432
```

IMPORTANTE: la variabile **repo1-path** dovrà essere valorizzata col path (condiviso tra i 3 nodi del cluster ad esempio una share NFS montata localmente sui 3 nodi) dove vogliamo vadano i nostri backup, nel nostro caso la settiamo per scrivere sul 2° HD sdb1 (per fare i test non avendo una share NFS) :

`repo1-path=/data/postgresql/backup`

Ora possiamo creare la stanza

Importante: posizionarsi sul nodo Leader:

```
postgres@dbserver03:~$ pgbackrest stanza-create --stanza=main --log-level-console=info
2023-06-19 15:20:01.837 P00   INFO: stanza-create command begin 2.43: --exec-id=485218-3a73989d --log-level-
console=info --log-level-file=debug --pg1-path=/var/lib/postgresql/14/main --pg1-port=5432 --repo1-
path=/data/postgresql/backup --stanza=main
2023-06-19 15:20:02.446 P00   INFO: stanza-create for stanza 'main' on repo1
2023-06-19 15:20:02.462 P00   INFO: stanza-create command end: completed successfully (636ms)
```

Ora dobbiamo impostare i parametri del database per usare **pgbackrest**.

In particolare, dovremmo usare pgbackrest per il comando classico di PostgreSQL "archive_command".

Una delle maggiori preoccupazioni per l'utilizzo di cp come utilità in "archive_command" è che sono scrittori pigri: cioè non assicurano che tutto sia scritto correttamente e che fsync sia invocato.

Questo è un potenziale buco in molte configurazioni di backup mentre **pgbackrest** risolve questo problema.

Ecco le modifiche ai parametri di sistema di PostgreSQL per adattarlo a pgbackrest:

```
ALTER SYSTEM SET wal_level = 'replica';
ALTER SYSTEM SET archive_mode = 'on';
ALTER SYSTEM SET archive_command = 'pgbackrest --stanza=main archive-push %p';
ALTER SYSTEM SET max_wal_senders = '10';
ALTER SYSTEM SET hot_standby = 'on';
```

Dopo le modifiche di alter system bisogna riavviare il cluster attraverso Patroni

```
pietro@dbserver01:~$ patronictl -c /etc/patroni/patroni.yml reload postgres
+ Cluster: postgres --------+---------+---------+----+-----------+
| Member     | Host         | Role    | State   | TL | Lag in MB |
+------------+--------------+---------+---------+----+-----------+
| dbserver01 | 10.116.82.32 | Leader  | running |  7 |           |
| dbserver02 | 10.116.82.33 | Replica | running |  7 |         0 |
| dbserver03 | 10.116.82.34 | Replica | running |  7 |         0 |
+------------+--------------+---------+---------+----+-----------+
Are you sure you want to reload members dbserver01, dbserver02, dbserver03? [y/N]: y
Reload request received for member dbserver01 and will be processed within 10 seconds
Reload request received for member dbserver02 and will be processed within 10 seconds
Reload request received for member dbserver03 and will be processed within 10 seconds

pietro@dbserver01:~$ patronictl -c /etc/patroni/patroni.yml list
+ Cluster: postgres --------+---------+---------+----+-----------+-----------------+
| Member     | Host         | Role    | State   | TL | Lag in MB | Pending restart |
+------------+--------------+---------+---------+----+-----------+-----------------+
| dbserver01 | 10.116.82.32 | Leader  | running |  7 |           | *               |
| dbserver02 | 10.116.82.33 | Replica | running |  7 |         0 |                 |
| dbserver03 | 10.116.82.34 | Replica | running |  7 |         0 |                 |
+------------+--------------+---------+---------+----+-----------+-----------------+
```

Riavviamo il primo nodo dbserver01 (attualmente il leader)

```
pietro@dbserver01:~$ patronictl -c /etc/patroni/patroni.yml restart postgres dbserver01

+ Cluster: postgres --------+---------+---------+----+-----------+-----------------+
| Member     | Host         | Role    | State   | TL | Lag in MB | Pending restart |
+------------+--------------+---------+---------+----+-----------+-----------------+
| dbserver01 | 10.116.82.32 | Leader  | running |  7 |           | *               |
| dbserver02 | 10.116.82.33 | Replica | running |  7 |         0 |                 |
| dbserver03 | 10.116.82.34 | Replica | running |  7 |         0 |                 |
+------------+--------------+---------+---------+----+-----------+-----------------+
When should the restart take place (e.g. 2023-06-19T17:26)  [now]:
Are you sure you want to restart members dbserver01? [y/N]: y
Restart if the PostgreSQL version is less than provided (e.g. 9.5.2)  []:
Success: restart on member dbserver01

pietro@dbserver01:~$ patronictl -c /etc/patroni/patroni.yml list
+ Cluster: postgres --------+---------+---------+----+-----------+
| Member     | Host         | Role    | State   | TL | Lag in MB |
+------------+--------------+---------+---------+----+-----------+
| dbserver01 | 10.116.82.32 | Leader  | running |  7 |           |
| dbserver02 | 10.116.82.33 | Replica | running |  7 |         0 |
| dbserver03 | 10.116.82.34 | Replica | running |  7 |         0 |
+------------+--------------+---------+---------+----+-----------+
```

Verifichiamo la stanza creata

```
pietro@dbserver01:~$ sudo su - postgres
postgres@dbserver01:~$ pgbackrest check --stanza=main --log-level-console=info
2023-06-19 16:27:56.969 P00   INFO: check command begin 2.43: --exec-id=375238-17b690e0 --log-
level-console=info --log-level-file=debug --pg1-path=/var/lib/postgresql/14/main --pg1-port=5432
--repo1-path=/data/postgresql/backup --stanza=main
2023-06-19 16:27:57.581 P00   INFO: check repo1 configuration (primary)
2023-06-19 16:27:57.784 P00   INFO: check repo1 archive for WAL (primary)
2023-06-19 16:27:57.986 P00   INFO: WAL segment 000000070000000000000009 successfully archived to
'/data/postgresql/backup/archive/main/14-1/0000000700000000/000000070000000000000009-
fb26430f3323c60b17426f7023475919a05319d0.gz' on repo1
2023-06-19 16:27:57.986 P00   INFO: check command end: completed successfully (1020ms)
```

Proviamo a fare il primo Backup full

Con utente postgres sempre dal nod leader

```
postgres@dbserver01:~$ pgbackrest backup --stanza=main --log-level-console=info

2023-06-19 16:27:56.969 P00   INFO: check command begin 2.43: --exec-id=375238-17b690e0 --log-
level-console=info --log-level-file=debug --pg1-path=/var/lib/postgresql/14/main --pg1-port=5432
--repo1-path=/data/postgresql/backup --stanza=main
2023-06-19 16:27:57.581 P00   INFO: check repo1 configuration (primary)
2023-06-19 16:27:57.784 P00   INFO: check repo1 archive for WAL (primary)
2023-06-19 16:27:57.986 P00   INFO: WAL segment 000000070000000000000009 successfully archived to
'/data/postgresql/backup/archive/main/14-1/0000000700000000/000000070000000000000009-
fb26430f3323c60b17426f7023475919a05319d0.gz' on repo1
2023-06-19 16:27:57.986 P00   INFO: check command end: completed successfully (1020ms)
postgres@dbserver01:~$ pgbackrest backup --stanza=main --log-level-console=info
2023-06-19 16:35:49.135 P00   INFO: backup command begin 2.43: --exec-id=375895-8bd5f0ad --log-
level-console=info --log-level-file=debug --pg1-path=/var/lib/postgresql/14/main --pg1-port=5432
--process-max=2 --repo1-path=/data/postgresql/backup --repo1-retention-full=2 --stanza=main --
start-fast --stop-auto
WARN: no prior backup exists, incr backup has been changed to full
2023-06-19 16:35:49.855 P00   INFO: execute non-exclusive backup start: backup begins after the
requested immediate checkpoint completes
2023-06-19 16:35:50.658 P00   INFO: backup start archive = 000000070000000000000000B, lsn =
0/B000028
2023-06-19 16:35:50.658 P00   INFO: check archive for prior segment 000000070000000000000000A
2023-06-19 16:35:55.671 P00   INFO: execute non-exclusive backup stop and wait for all WAL
segments to archive
2023-06-19 16:35:55.873 P00   INFO: backup stop archive = 000000070000000000000000B, lsn =
0/B000138
2023-06-19 16:35:55.877 P00   INFO: check archive for segment(s)
000000070000000000000000B:000000070000000000000000B
2023-06-19 16:35:56.004 P00   INFO: new backup label = 20230619-163549F
2023-06-19 16:35:56.069 P00   INFO: full backup size = 40.5MB, file total = 1347
2023-06-19 16:35:56.069 P00   INFO: backup command end: completed successfully (6939ms)
2023-06-19 16:35:56.069 P00   INFO: expire command begin 2.43: --exec-id=375895-8bd5f0ad --log-
level-console=info --log-level-file=debug --repo1-path=/data/postgresql/backup --repo1-retention-
full=2 --stanza=main
2023-06-19 16:35:56.083 P00   INFO: expire command end: completed successfully (14ms)
```

Poiché stiamo eseguendo un backup per la prima volta, pgBackRest rileva che non esiste alcun backup precedente e così passa al backup completo del database (evidenziato in giallo).

Controlliamo le info del backup appena fatto

```
postgres@dbserver01:~$ pgbackrest info
stanza: main
    status: ok
    cipher: none

    db (current)
        wal archive min/max (14): 000000010000000000000001/000000070000000000000000B

        full backup: 20230619-163549F
            timestamp start/stop: 2023-06-19 16:35:49 / 2023-06-19 16:35:55
            wal start/stop: 000000070000000000000000B / 000000070000000000000000B
            database size: 40.5MB, database backup size: 40.5MB
            repo1: backup set size: 5.8MB, backup size: 5.8MB
```

Come ripristare/restorare un backup completo di tutti i DB

Con utente postgres sempre dal nodo leader

```
pgbackrest restore --stanza=main --log-level-console=info
```

Come ripristare/restorare solo il backup di uno specifico DB

```
pgbackrest restore --stanza=main --log-level-console=info --db-include=dvdrental
```

Come ripristare solo il backup o parti di esso di uno specifico DB a uno specifico point in time

Facciamo un esempio pratico sul DB di esempio "percona" seguente:

dal nodo Leader connettiamoci al DB e poi elenchiamo tutti i DB

```
postgres=# \l
                                 List of databases
   Name    |  Owner   | Encoding | Collate |  Ctype  |   Access privileges
-----------+----------+----------+---------+---------+-----------------------
 mydb      | postgres | UTF8     | C.UTF-8 | C.UTF-8 |
 percona   | postgres | UTF8     | C.UTF-8 | C.UTF-8 |
 postgres  | postgres | UTF8     | C.UTF-8 | C.UTF-8 |
 template0 | postgres | UTF8     | C.UTF-8 | C.UTF-8 | =c/postgres          +
           |          |          |         |         | postgres=CTc/postgres
 template1 | postgres | UTF8     | C.UTF-8 | C.UTF-8 | =c/postgres          +
           |          |          |         |         | postgres=CTc/postgres
```

Cambiamo DB e passiamo al DB percona

```
postgres=# \l+ percona
                                                                 List of databases
  Name   |  Owner   | Encoding | Collate |  Ctype  | Access privileges |  Size  | Tablespace | Description
---------+----------+----------+---------+---------+-------------------+--------+------------+-------------
 percona | postgres | UTF8     | C.UTF-8 | C.UTF-8 |                   | 157 MB | pg_default |
```

Ripristino del backup a un momento specifico

Il ripristino point-in-time è possibile con pgBackRest, si consideri che una tabella o un database è stata eliminata e deve essere ripristinata. In questa situazione, abbiamo bisogno del timestamp dell'evento drop, del backup pgBackRest e degli archivi. Quindi sul nodo primario ho una tabella denominata **pitr** nel database **percona**. Questa tabella è stata eliminata al timestamp 2022-11-04 14:24:32.231309+05:30.

Elenchiamo le relazione

```
postgres=# \dt+ pitr
                              List of relations
 Schema | Name | Type  |  Owner   | Persistence |    Size    | Description
--------+------+-------+----------+-------------+------------+-------------
 public | pitr | table | postgres | permanent   | 8192 bytes |
(1 row)
```

Elenchiamo la data attuale del DB e poi cancelliamo la tabella pitr

```
postgres=# select now();drop table pitr;
              now
-----------------------------------
 2022-11-04 14:24:32.231309+05:30
(1 row)

DROP TABLE
```

Verifichiamo esista l'evento drop nel point time del WAL

```
postgres=# select pg_switch_wal();
 pg_switch_wal
---------------
 0/7B0162A0
(1 row)
```

Utilizzando l'opzione di ripristino pgBackRest con il comando recovery type possiamo ottenere il ripristino point-in-time, per impostazione predefinita questo tipo ripristina l'archivio fino alla fine del flusso wal, in questo scenario, specificheremo il timestamp esatto per ripristinare la tabella che avevamo cancellato.

Ritornare come utente postgres di OS e dare il seguente impianto di comandi:

```
pgbackrest --log-level-console=info --stanza=demo --db-include=percona --type=time "--
target=2022-11-04 14:24:31" restore
```

Appunti

Comandi plsql

Elencare tutti i database

```
postgres=> \l
                                  List of databases
   Name    |   Owner   | Encoding |   Collate   |    Ctype    |    Access privileges
-----------+-----------+----------+-------------+-------------+-----------------------
 dvdrental | pgbouncer | UTF8     | en_US.UTF-8 | en_US.UTF-8 | =Tc/pgbouncer          +
           |           |          |             |             | pgbouncer=CTc/pgbouncer
 postgres  | postgres  | UTF8     | en_US.UTF-8 | en_US.UTF-8 |
 template0 | postgres  | UTF8     | en_US.UTF-8 | en_US.UTF-8 | =c/postgres            +
           |           |          |             |             | postgres=CTc/postgres
 template1 | postgres  | UTF8     | en_US.UTF-8 | en_US.UTF-8 | =c/postgres            +
           |           |          |             |             | postgres=CTc/postgres
(4 rows)
```

Elencare le tablespace

```
postgres=# \db
       List of tablespaces
    Name    |  Owner   | Location
------------+----------+----------
 pg_default | postgres |
 pg_global  | postgres |
(2 rows)
```

Elencare le tablespaces + altre informazioni

```
postgres=# \db+
                                 List of tablespaces
    Name    |  Owner   | Location | Access privileges | Options |  Size  | Description
------------+----------+----------+-------------------+---------+--------+-------------
 pg_default | postgres |          |                   |         | 40 MB  |
 pg_global  | postgres |          |                   |         | 560 kB |
(2 rows)
```

Elencare tabelle schema e db corrente

```
postgres=> \dt
            List of relations
 Schema |    Name     | Type  |  Owner
--------+-------------+-------+----------
 public | pgsqlhatest | table | postgres
(1 row)
```

Elencare tutti gli utenti

```
postgres=> \du
                                   List of roles
 Role name  |                         Attributes                         | Member of
------------+------------------------------------------------------------+-----------
 admin      | Create role, Create DB                                     | {}
 pgbouncer  |                                                            | {}
 pgrewind   |                                                            | {}
 postgres   | Superuser, Create role, Create DB, Replication, Bypass RLS | {}
 replicator | Replication                                                | {}
```

elenca tutti i comandi digitati (history)

```
postgres=> \s
```

elenca le relazioni

```
postgres-# \dt+
                                      List of relations
 Schema |    Name     | Type  |  Owner   | Persistence | Access method | Size  | Description
--------+-------------+-------+----------+-------------+---------------+-------+-------------
 public | pgsqlhatest | table | postgres | permanent   | heap          | 32 kB |
(1 row)
```

Elenchiamo utti i nodi replica e loro stato

```
postgres@dbserver01:~$ psql
psql (14.7 - Percona Distribution (Ubuntu 2:14.7-1.jammy))
Type "help" for help.

postgres=# select usename,client_addr,sync_state,state  from pg_catalog.pg_stat_replication;
  usename   | client_addr  | sync_state |   state
------------+--------------+------------+-----------
 replicator | 10.116.82.34 | async      | streaming
 replicator | 10.116.82.33 | async      | streaming
(2 rows)
```

Comandi ETCD (DCS)

Sui nodi etcd-server01-02-03
pietro@etcd-server01:~$ etcdctl ls
/pg_cluster

pietro@etcd-server01:~$ etcdctl ls /pg_cluster
/pg_cluster/postgres

Se volessi rimuovere l'archivio DCS (pericoloso) :
etcdctl rm --recursive /pg_cluster/postgres

Comandi Patroni più importanti

patronictl edit-config

To edit postgres configuration parameters you can use edit-config command. It will open configuration file in editor, make the required changes and Patroni will validate all parameters before saving configuration file. You can also add pg_hba entries to the configuration so these will be reflected all over the cluster
When to use: If you want to change some postgres parameter, add/remove pg_hba entries - you can use patronictl edit-config command
How to use:
patronictl -c /etc/patroni/patroni.yaml edit-config

patronictl reload

This command will reload parameters from configuration file and takes required action like restart on cluster nodes
When to use: If you have changed parameters in configuration file using edit-config you can use reload command for parameters to take effect
How to use:
patronictl -c /etc/patroni/patroni.yaml reload patroni_cluster

patronictl switchover

It will make selected replica as master node basically will switch all traffic to new selected node. We can have planned switchover at particular time as well.
When to use: If you have maintenance for master node you can switchover master to another node in the cluster
How to use:
It will ask for node to switchover and also time for switchover
patronictl -c /etc/patroni/patroni.yaml switchover

patronictl pause

Patroni will stop managing postgres cluster and will turn on the maintenance mode. If you want to do some manual activities for maintenance you need to stop patroni from auto managing cluster.
When to use: If you want to put cluster in maintenance mode and manage Postgres database manually for some time, you can use pause command so that Patroni will stop managing the cluster
How to use:
patronictl -c /etc/patroni/patroni.yaml pause

patronictl resume

It will start the paused cluster management and remove the cluster from maintenance mode
When to use: If you want to turn off maintenance mode, you can use resume command and patroni will start managing the cluster
How to use:
patronictl -c /etc/patroni/patroni.yaml resume

patronictl list

List all nodes and it's role, status. You can use it for checking status of all nodes, which is the master and which all are slaves/replicas.
When to use: To check list and status of all nodes in the cluster, you can get all the information about nodes including if any restart is required for any node
How to use:
patronictl -c /etc/patroni/patroni.yaml list

patronictl restart

It will restart single node in the postgres cluster or all nodes(complete cluster). Patroni will do the rolling restart for postgres on all nodes.

When to use: Sometimes you need to restart all nodes in the cluster without downtime, you can use this command for rolling restart

How to use:

Restart particular node in cluster

patronictl -c /etc/patroni/patroni.yaml restart <CLUSTER_NAME> <NODE_NAME>

Restart whole cluster(all nodes in cluster)

patronictl -c /etc/patroni/patroni.yaml restart <CLUSTER_NAME>

patronictl reinit

It will reinitialize node in the cluster. If you want to reinitialize particular replica or slave node you can reinitialize node using reinit command.

When to use: patronictl reinit command allows you to reinitialize a specific node and can be utilized when a cluster node experiences failure in starting or displays an unknown status for the node in the cluster . It is often useful in cases where a node has corrupt data.

How to use:

patronictl -c /etc/patroni/patroni.yaml reinit <CLUSTER_NAME> <NODE_NAME>

CASI PRATICI DI DIFETTI CLUSTER

Caso 1

Dopo un riavvio dei 3 nodi a iniziare dal 1° nodo dbserver01, dopo il riavvio i membri replica rimangono in uno stato di perenne starting e in Lag unknown

```
pietro@dbserver01:~$ patronictl -c /etc/patroni/patroni.yml list

+ Cluster: postgres --------+---------+----------+----+-----------+
| Member     | Host         | Role    | State    | TL | Lag in MB |
+------------+--------------+---------+----------+----+-----------+
| dbserver01 | 10.116.82.32 | Leader  | running  |  5 |           |
| dbserver02 | 10.116.82.33 | Replica | starting |    |   unknown |
| dbserver03 | 10.116.82.34 | Replica | starting |    |   unknown |
+------------+--------------+---------+----------+----+-----------+
```

Diamo il comando di reinit membro del clustwer postgres

```
pietro@dbserver01:~$ patronictl -c /etc/patroni/patroni.yml reinit postgres
+ Cluster: postgres --------+---------+----------+----+-----------+
| Member     | Host         | Role    | State    | TL | Lag in MB |
+------------+--------------+---------+----------+----+-----------+
| dbserver01 | 10.116.82.32 | Leader  | running  |  5 |           |
| dbserver02 | 10.116.82.33 | Replica | starting |    |   unknown |
| dbserver03 | 10.116.82.34 | Replica | starting |    |   unknown |
+------------+--------------+---------+----------+----+-----------+
Which member do you want to reinitialize [dbserver03, dbserver02]? []: dbserver02
Are you sure you want to reinitialize members dbserver02? [y/N]: y
Success: reinitialize for member dbserver02
pietro@dbserver01:~$ patronictl -c /etc/patroni/patroni.yml list
+ Cluster: postgres --------+---------+----------+----+-----------+
| Member     | Host         | Role    | State    | TL | Lag in MB |
+------------+--------------+---------+----------+----+-----------+
| dbserver01 | 10.116.82.32 | Leader  | running  |  5 |           |
| dbserver02 | 10.116.82.33 | Replica | starting |    |   unknown |
| dbserver03 | 10.116.82.34 | Replica | starting |    |   unknown |
+------------+--------------+---------+----------+----+-----------+
pietro@dbserver01:~$ patronictl -c /etc/patroni/patroni.yml list
+ Cluster: postgres --------+---------+----------+----+-----------+
| Member     | Host         | Role    | State    | TL | Lag in MB |
+------------+--------------+---------+----------+----+-----------+
| dbserver01 | 10.116.82.32 | Leader  | running  |  5 |           |
| dbserver02 | 10.116.82.33 | Replica | running  |  5 |         0 |
| dbserver03 | 10.116.82.34 | Replica | starting |    |   unknown |
+------------+--------------+---------+----------+----+-----------+
pietro@dbserver01:~$ patronictl -c /etc/patroni/patroni.yml reinit postgres
+ Cluster: postgres --------+---------+----------+----+-----------+
| Member     | Host         | Role    | State    | TL | Lag in MB |
+------------+--------------+---------+----------+----+-----------+
| dbserver01 | 10.116.82.32 | Leader  | running  |  5 |           |
| dbserver02 | 10.116.82.33 | Replica | running  |  5 |         0 |
| dbserver03 | 10.116.82.34 | Replica | starting |    |   unknown |
+------------+--------------+---------+----------+----+-----------+
Which member do you want to reinitialize [dbserver02, dbserver03]? []: dbserver03
Are you sure you want to reinitialize members dbserver03? [y/N]: y
Success: reinitialize for member dbserver03
pietro@dbserver01:~$ patronictl -c /etc/patroni/patroni.yml list
+ Cluster: postgres --------+---------+---------+----+-----------+
| Member     | Host         | Role    | State   | TL | Lag in MB |
+------------+--------------+---------+---------+----+-----------+
| dbserver01 | 10.116.82.32 | Leader  | running |  5 |           |
| dbserver02 | 10.116.82.33 | Replica | running |  5 |         0 |
| dbserver03 | 10.116.82.34 | Replica | running |  5 |         0 |
+------------+--------------+---------+---------+----+-----------+
```

Cluster di nuovo perfettamente funzionante

Dimensionamento di un custer PostgreSQL

Dimenzionamento Cluster medie dimensioni

Componenti	**Numero di VMs**	**Configurazione**	Note
ETCD Cluster(DCS)	3	2 Cores/4GB	DCS numeri dispari: 3-5-etc.
HAProxy	2	4 Cores/8GB	Load Balancer in HA
Postgres + Patroni	3	8 Cores/16GB	Nodi Cluster PostgreSQL

Continua...