

**Titulación:** Grado en Ingeniería Informática y Sistemas de Información  
**Curso:** 2018-2019. Convocatoria Ordinaria de Enero  
**Asignatura:** Bases de Datos Avanzadas – Laboratorio

## **Practica 4: Replicación e Implementación de una Base de Datos Distribuida.**

### **ALUMNO 1:**

**Nombre y Apellidos:** Marcos Barranquero

**DNI:** 51129104N

### **ALUMNO 2:**

**Nombre y Apellidos:** Eduardo Graván Serrano

**DNI:** 03212337L

**Fecha:** 16/01/2019

**Profesor Responsable:** Iván González

Mediante la entrega de este fichero los alumnos aseguran que cumplen con la normativa de autoría de trabajos de la Universidad de Alcalá, y declaran éste como un trabajo original y propio.

En caso de ser detectada copia, se puntuará **TODA** la práctica como Suspenso – Cero.

## **Plazos**

Tarea en Laboratorio: Semana 10 y 17 Diciembre.

Entrega de práctica: Día 20 de Enero de 2018. Aula Virtual

Documento a entregar: Este mismo fichero con los pasos de la implementación de la replicación y la base de datos distribuida, las pruebas realizadas de su funcionamiento; y los ficheros de configuración del maestro y del esclavo utilizados en replicación; y de la configuración de los servidores de la base de datos distribuida. Obligatorio. Se debe de entregar en un ZIP comprimido: **DNI 'sdelosAlumnos\_PECL4.zip**

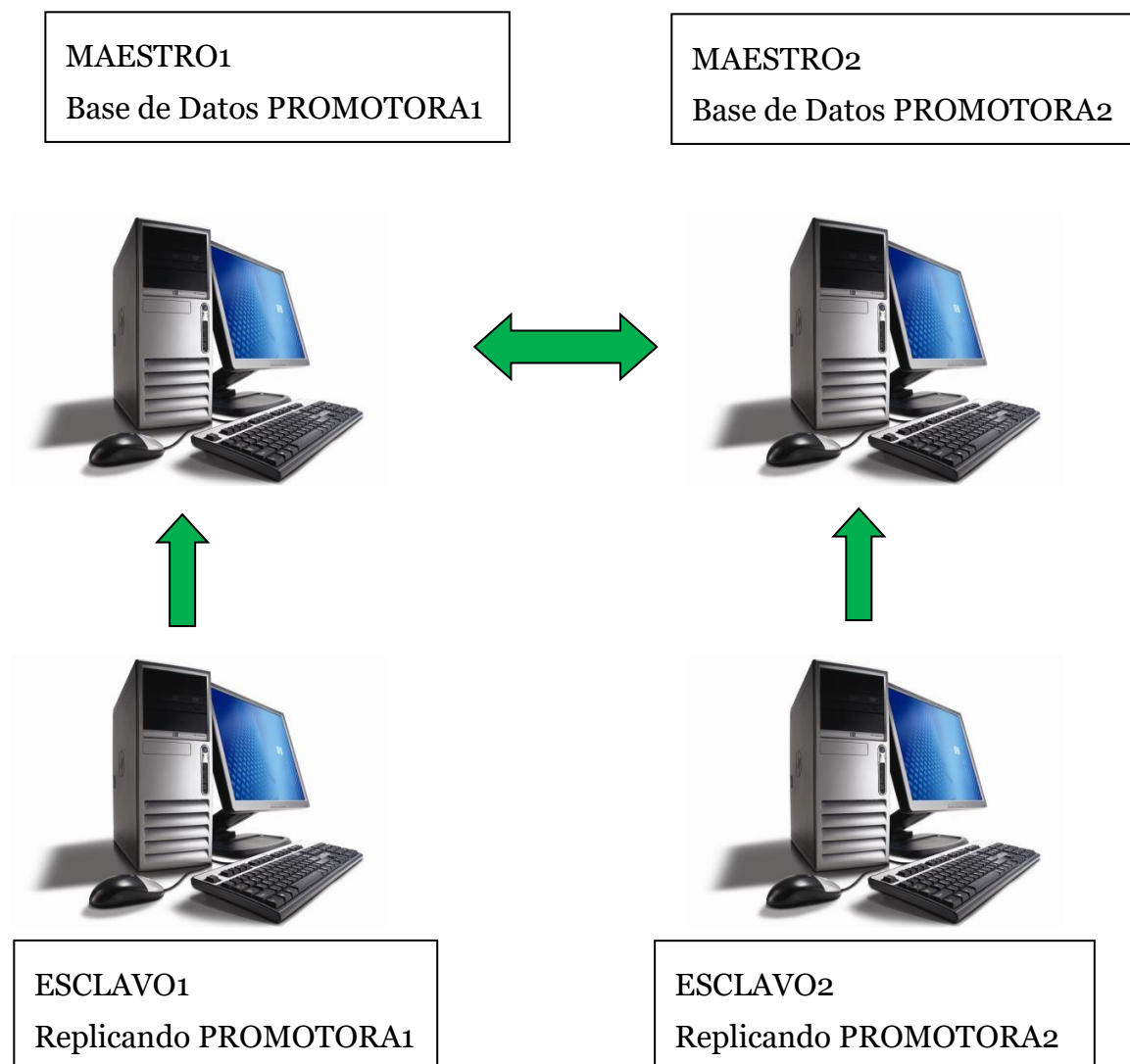
**AMBOS ALUMNOS DEBEN ENTREGAR EL FICHERO EN LA PLATAFORMA.**

## Introducción

El contenido de esta práctica versa sobre la Replicación de Bases de Datos con PostgreSQL e introducción a las bases de datos distribuidas. Concretamente se va a utilizar los servicios de replicación de bases de datos que tiene PostgreSQL. Para ello se utilizará PostgreSQL 10.x con soporte para replicación. **Se prohíbe el uso de cualquier otro programa externo a PostgreSQL para realizar la replicación, como puede ser Slony.**

También se va a diseñar e implementar una pequeña base de datos distribuida. Una base de datos distribuida es una base de datos lógica compuesta por varios nodos (equipos) situados en un lugar determinado, cuyos datos almacenados son diferentes; pero que todos ellos forman una base de datos lógica. Generalmente, los datos se reparten entre los nodos dependiendo de donde se utilizan más frecuentemente.

El escenario que se pretende realizar se muestra en el siguiente esquema:



Se van a necesitar 4 máquinas: 2 maestros y 2 esclavos. Cada maestro puede ser un ordenador de cada miembro del grupo con una base de datos de unos grupos

musicales en concreto (PROMOTORA1 y PROMOTORA2). Dentro de cada maestro se puede instalar una máquina virtual, que se corresponderá con el esclavo que se encarga de replicar la base de datos que tiene cada maestro. Es decir, hay una base de datos MUSICOS1 y otra base de datos MUSICOS2 que corresponden con una base de datos que almacena los grupos, músicos, discos, canciones, conciertos y entradas de la Promotora 1 y de la Promotora2 respectivamente.

Se debe de entregar una memoria descriptiva detallada que posea como mínimo los siguientes puntos:

**¿Qué soluciones dispone PostgreSQL para poder suministrar alta disponibilidad de las bases de datos y replicación? Comentar brevemente cada uno de los métodos. Elegir uno de los métodos propuestos.**

Tal y como se indica en <https://www.postgresql.org/docs/10/different-replication-solutions.html>, disponemos de los siguientes métodos:

- **Recuperación frente a fallos por disco compartido:** la BBDD solo está copiada en un único array de discos compartido por varios servidores. Si el servidor principal falla, el servidor secundario puede montar la BBDD como si estuviese recuperando la BBDD de un crasheo.  
Es bastante rápido y no pierdes datos.
- **Replicación del sistema de archivos:** se mantiene una copia constante del sistema de archivos en el servidor secundario. Debe escribirse y actualizarse en el mismo orden que el servidor principal.
- **Envío de WAL's:** las técnicas de WAL o Write-ahead logging permiten atomicidad y durabilidad. Consisten en escribir primero un log de lo que se va a hacer antes de escribir cambios en la base de datos. El servidor secundario puede ir leyendo este log. Si el servidor principal falla, el secundario tiene prácticamente todos los datos del principal y puede ser el nuevo servidor principal.
- **Replicación lógica:** partiendo del WAL, se envía un flujo de modificaciones de un servidor a otro. Permite cambios en las tablas individuales.
- **Replicación Principal-Secundario por Trigger:** el servidor secundario envía todas las peticiones que modifiquen datos al servidor maestro. EL servidor maestro envía asíncronamente los cambios de datos al servidor secundario.
- **Replicación por queries en medio:** un programa intercepta las queries SQL y las envía a los servidores. Cada servidor trabaja de forma independiente. Puede ser que la query tenga valores apropiados para un único servidor.
- **Replicación multimaster asíncrona:** para los servidores que no se encuentran permanentemente conectados y trabajan

independientemente, se pueden hacer replicasiones asíncronas periódicamente.

- **Replicación multimaster síncrona:** cada servidor acepta peticiones y los datos modificados son transmitidos antes de cada transacción. Se pueden enviar peticiones de read a cualquier servidor. Muchas modificaciones de escritura pueden causar un mal rendimiento.

Hemos decidido utilizar el envío de WALs debido a que hemos encontrado mayor documentación y tutoriales para este mecanismo, y nos resulta más sencillo.

**¿Qué soluciones dispone PostgreSQL para poder realizar consultas en servidores externos de PostgreSQL? Comentar brevemente cada uno de los métodos. Elegir uno de los métodos propuestos.**

Disponemos de dos principales opciones:

- **Dblink:** es una extensión que permite realizar cualquier consulta SQL que devuelva registros en una base de datos remota. Para realizar la consulta, el primer argumento debe ser la dirección de la BBDD (host, user, pass, dbname, [...]), y el segundo la consulta a realizar. Ambos en una string.
- **Postgres\_fdw:** es otra extensión, más moderna que la anterior, que permite realizar consultas externas a otra base de datos.

Para realizar la práctica, se ha elegido utilizar Dblink debido a la existencia de mayor documentación, ejemplos y tutoriales en internet, gracias a ser más antiguo.

## Configuración de cada uno de los nodos maestros de la base de datos de los grupos musicales de tal manera que se puedan recibir y realizar consultas sobre las bases externas que no tienen implementadas.

Lo primero que hemos realizado es generar datos aleatorios de músicos. Hemos reducido significativamente la cantidad de registros a generar.

Para realizar la práctica se han lanzado 4 máquinas virtuales: dos maestros y dos esclavos. En cada una de ellas se han instalado PostgreSQL y configurado adecuadamente la red para que se “vean” entre sí.

También hemos desactivado el firewall para evitarnos problemas de permisos. Sabemos que se podría configurar adecuadamente, pero es más rápido de cara a la práctica desactivarlo.

Resultado:

Nodo	IP
MAESTRO1	192.168.182.130
MAESTRO2	192.168.182.131

En ambas hemos incluido el esquema de músicos gracias al pgModeler, e introducido los datos generados previamente.

Tras esto, hemos configurado los respectivos archivos postgresql.conf y pg\_hba.conf para que puedan aceptar conexiones externas:

C:\Program Files\PostgreSQL\10\data\postgresql.conf:

```
# - Connection Settings -

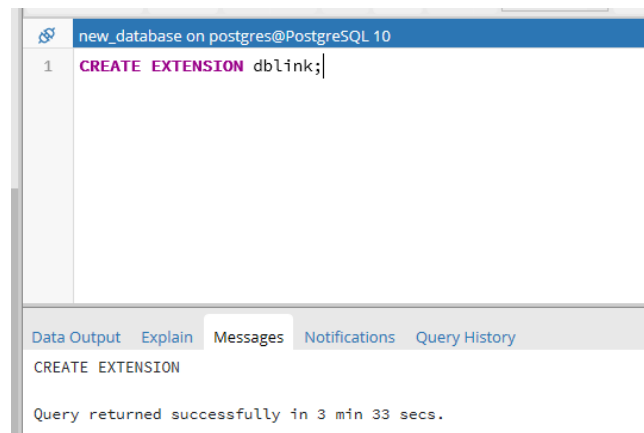
listen_addresses = '*'          # what IP address(es) to listen on;
                                # comma-separated list of addresses;
                                # defaults to 'localhost'; use '*' for all
                                # (change requires restart)
port = 5432                     # (change requires restart)
max_connections = 100           # (change requires restart)
```

C:\Program Files\PostgreSQL\10\data\pg\_hba.conf:

```
77 # TYPE DATABASE USER ADDRESS METHOD
78
79 # IPv4 local connections:
80 host all all 127.0.0.1/32 md5
81 host all all 192.168.182.1/24 md5
82 # IPv6 local connections:
83 host all all ::1/128 md5
84 # Allow replication connections from localhost, by a user with the
85 # replication privilege.
86 host replication all 127.0.0.1/32 md5
87 host replication all ::1/128 md5
88
```

Se puede observar que en pg\_hba.conf hemos añadido una nueva IPv4 para poder permitir las conexiones entre las bases de datos.

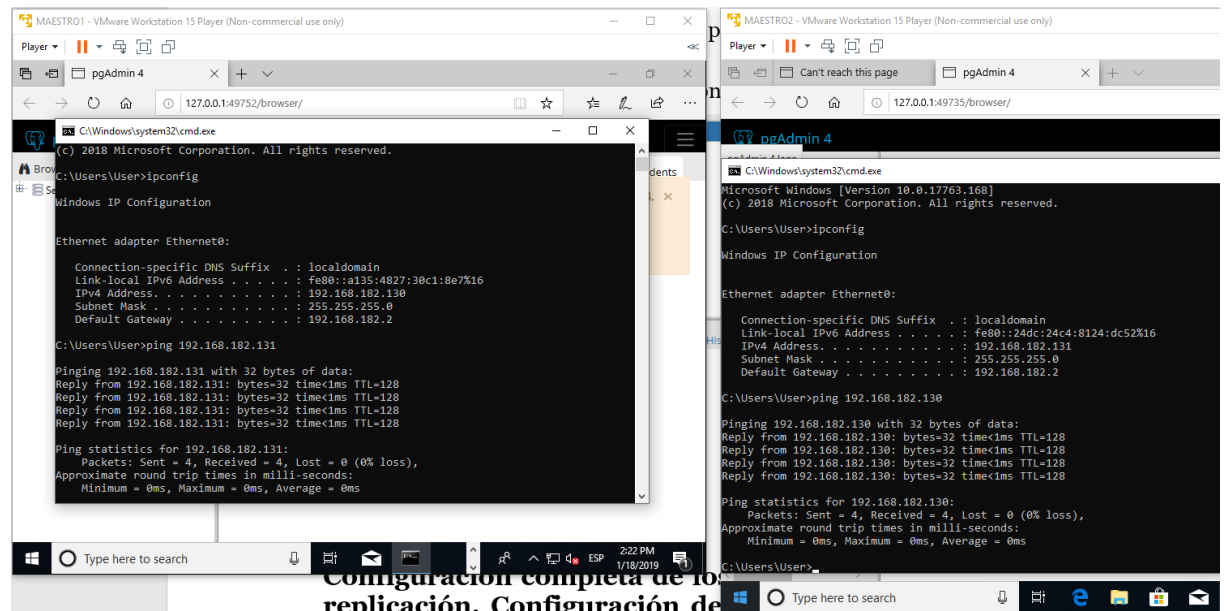
Tras esto, instalamos la extensión dblink:



```
new_database on postgres@PostgreSQL 10
1 CREATE EXTENSION dblink;

Data Output Explain Messages Notifications Query History
CREATE EXTENSION
Query returned successfully in 3 min 33 secs.
```

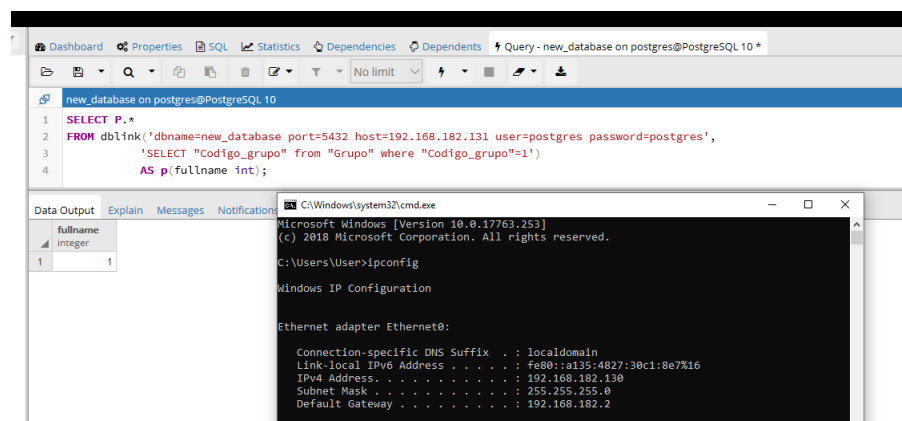
Verificamos la conectividad entre máquinas:



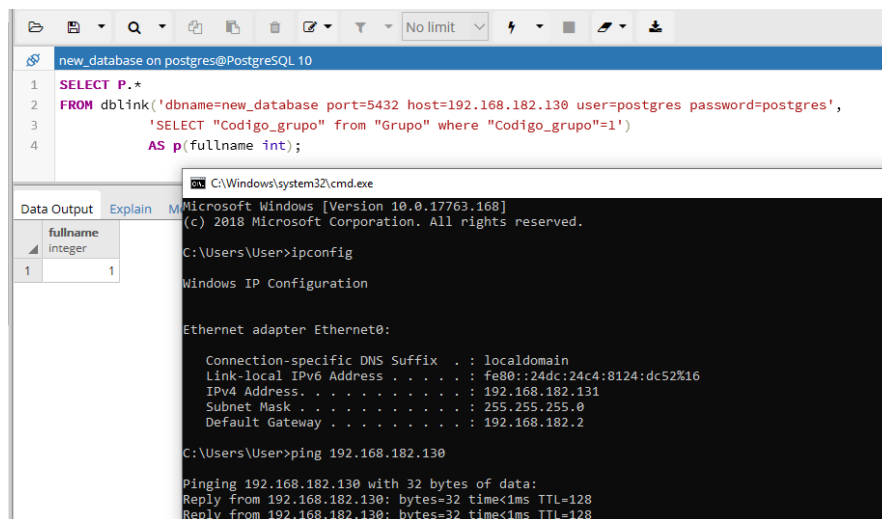
Configuración completa de 10  
replicación. Configuración de

Y ejecutamos una consulta de prueba:

MAESTRO1 preguntando a MAESTRO2:



MAESTRO2 preguntando a MAESTRO1:



**Configuración completa de los equipos para estar en modo de replicación. Configuración del nodo maestro. Tipos de nodos maestros, diferencias en el modo de funcionamiento y tipo elegido. Tipos de nodos esclavos, diferencias en el modo de funcionamiento y tipo elegido, etc.**

### *Nodo maestro (Maestro 1)*

Como se contestó en la primera pregunta, utilizaremos los envíos de WALs.

Creamos un directorio en C:\WALS\ donde guardaremos nuestros archivos WAL. Este directorio debe ser accesible por todos los nodos.

Tras esto, cambiamos la configuración de postgresql.conf:

```
#-----
# WRITE AHEAD LOG
#-----

# - Settings -

wal_level = hot_standby          # minimal, replica, or logical
                                # (change requires restart)
#fsync = on                     # flush data to disk for crash safety
                                # (turning this off can cause
                                # unrecoverable data corruption)
#synchronous_commit = on       # synchronization level;
                                # off, local, remote_write, remote_apply, or on
#wal_sync_method = fsync        # the default is the first option
                                # supported by the operating system:
                                #   open_datasync
                                #   fdatasync (default on Linux)
                                #   fsync
                                #   fsync_writethrough
                                #   open_sync
#full_page_writes = on         # recover from partial page writes
#wal_compression = off        # enable compression of full-page writes
#wal_log_hints = off           # also do full page writes of non-critical updates
                                # (change requires restart)
#wal_buffers = -1              # min 32kB, -1 sets based on shared_buffers
                                # (change requires restart)
#wal_writer_delay = 200ms      # 1-10000 milliseconds
#wal_writer_flush_after = 1MB  # measured in pages, 0 disables
```

```

#commit_delay = 0          # range 0-100000, in microseconds
#commit_siblings = 5       # range 1-1000

# - Checkpoints -

#checkpoint_timeout = 5min  # range 30s-1d
max_wal_size = 1GB
#min_wal_size = 80MB
#checkpoint_completion_target = 0.5 # checkpoint target duration, 0.0 - 1.0
#checkpoint_flush_after = 0      # measured in pages, 0 disables
#checkpoint_warning = 30s       # 0 disables

# - Archiving -

archive_mode = on           # enables archiving; off, on, or always
                           # (change requires restart)
#archive_command = 'copy "%p" "C:\\WALS\\%f"'

#archive_timeout = 0        # force a logfile segment switch after this
                           # number of seconds; 0 disables

#-----
# REPLICATION
#-----

# - Sending Server(s) -

# Set these on the master and on any standby that will send replication data.

max_wal_senders = 5         # max number of walsender processes
                           # (change requires restart)
wal_keep_segments = 10     # in logfile segments, 16MB each; 0 disables
#wal_sender_timeout = 60s   # in milliseconds; 0 disables

max_replication_slots = 5   # max number of replication slots
                           # (change requires restart)
#track_commit_timestamp = off # collect timestamp of transaction commit
                           # (change requires restart)

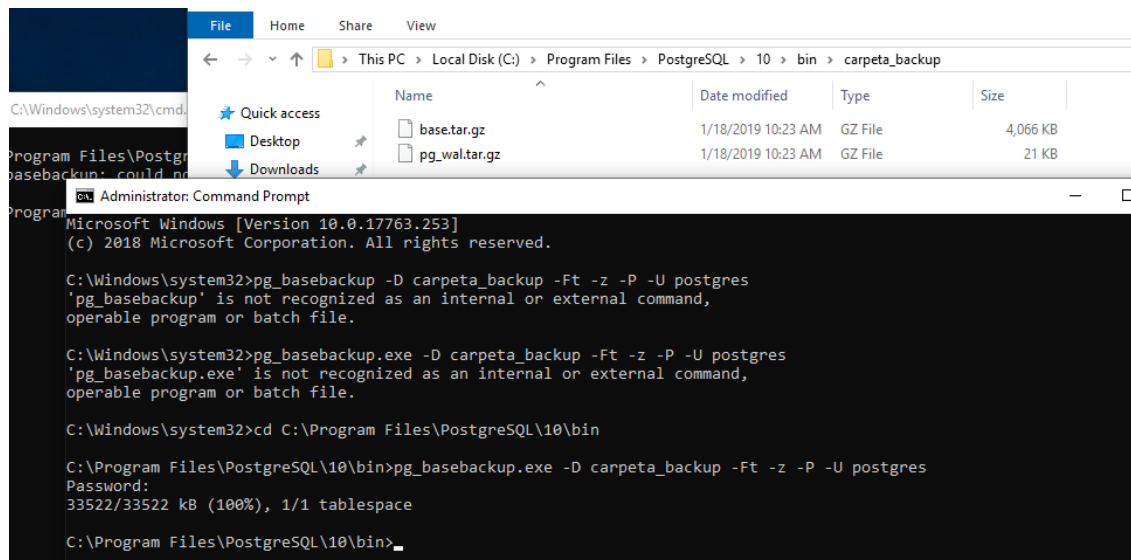
```

Después, añadimos al archivo C:\Program Files\PostgreSQL\10\data\pg\_hba.conf el nodo esclavo:

#	TYPE	DATABASE	USER	ADDRESS	METHOD
# IPv4 local connections:					
host	all	all	127.0.0.1/32	md5	
host	all	all	192.168.182.1/24	md5	
# IPv6 local connections:					
host	all	all	:::1/128	md5	
# Allow replication connections from localhost, by a user with the					
# replication privilege.					
host	replication	all	127.0.0.1/32	md5	
host	replication	all	:::1/128	md5	
<b>host</b>	<b>replication</b>	<b>all</b>	<b>192.168.182.132/32</b>	<b>md5</b>	

Realizamos un backup de la base de datos:





A continuación, copiamos el cluster (C:\Program Files\PostgreSQL\10\data), exceptuando los archivos de configuración pg\_hba.conf y postgresql.conf.

Tras todo esto, reiniciamos el servidor para que se apliquen los cambios.

Hacemos la misma operación para el MAESTRO2, añadiendo a ESCLAVO2.

### *Nodos esclavos*

Nodo	IP	Referencia a	Cuya IP es
ESCLAVO1	192.168.182.132	MAESTRO1	192.168.182.130
ESCLAVO2	192.168.182.133	MAESTRO2	192.168.182.131

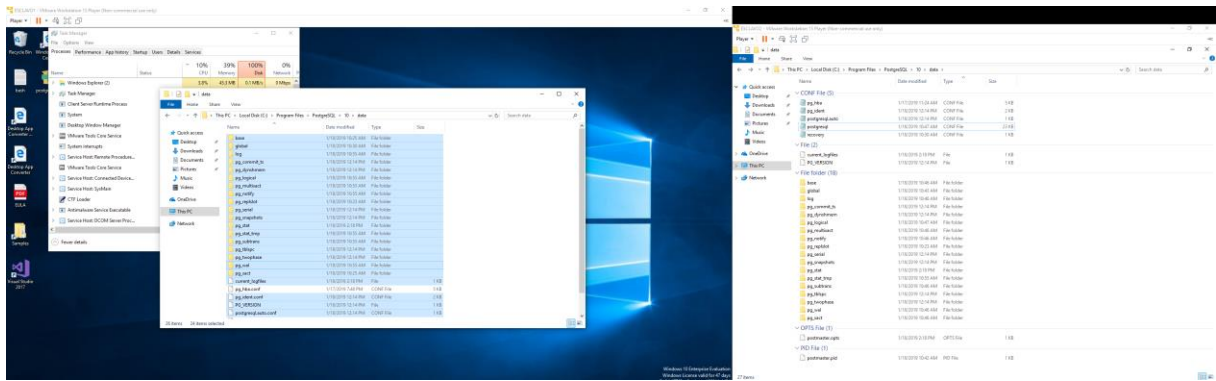
A el nodo ESCLAVO1, añadimos un archivo recovery.conf con el siguiente contenido:

```
standby_mode = 'on'
primary_conninfo = '192.168.182.130 port=5432 user=postgres
password=postgres'
restore_command = 'copy "%p" "C:\WALS\%f"'
recovery_target_timeline='latest'
```

Podemos reseñar que hace referencia al nodo MAESTRO1.

Y modificamos el archivo postgresql.conf poniendo a on el hot\_standby.

Finalmente, copiamos el cluster del maestro1 con el archivo recovery en el ESCLAVO1.



Se realizan las mismas operaciones para el ESCLAVO2.

**Operaciones que se pueden realizar en cada tipo de equipo de red. Provocar situaciones de caída de los nodos y observar mensajes, acciones correctoras a realizar para volver el sistema a un estado consistente.**

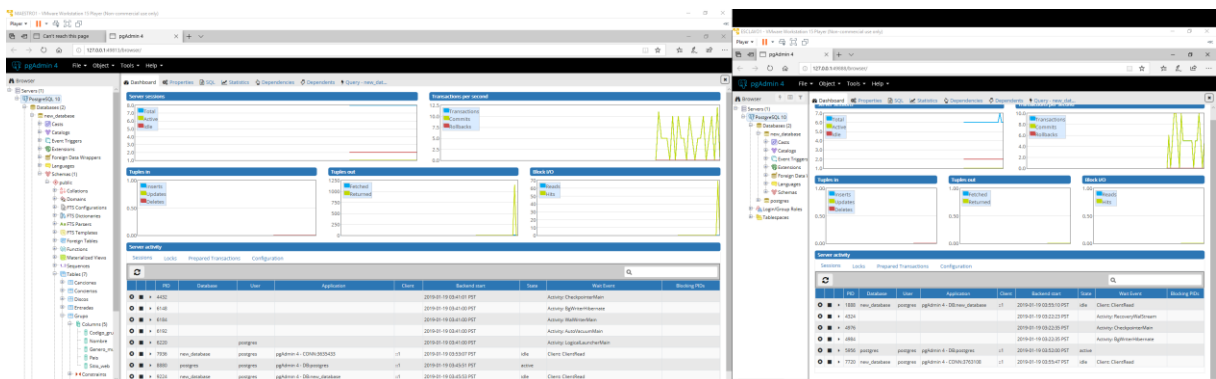
## *Nodo maestro*

Lectura, escritura y eliminación.

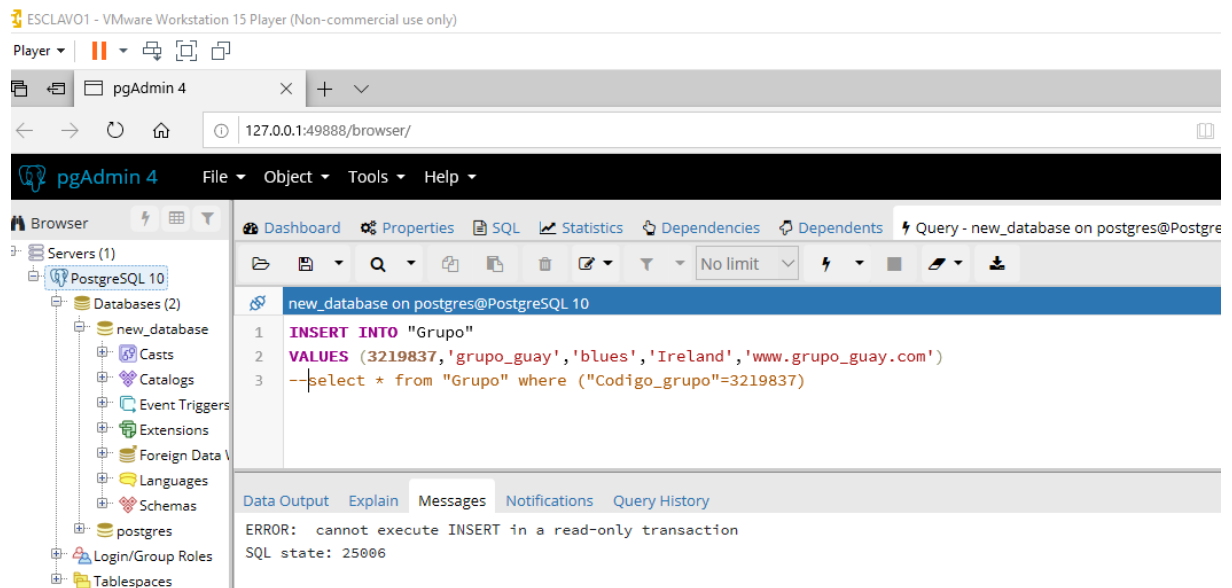
## *Nodo esclavo*

Sólo lectura (copiado). No deben escribir para no crear duplicidades ni errores tales como tener información incorrecta.

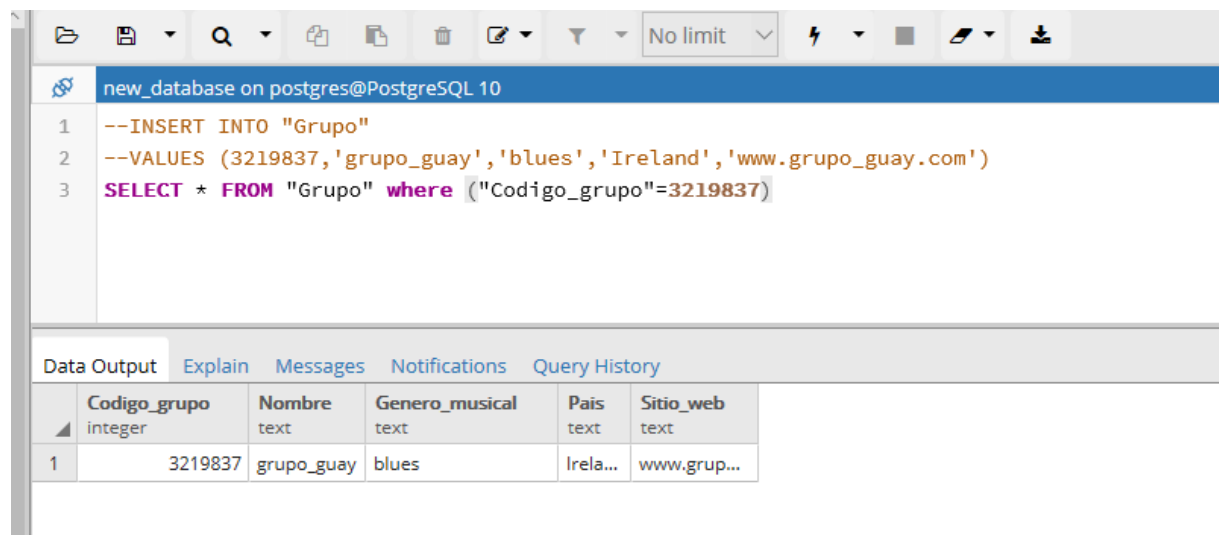
Podemos ver que los nodos están conectados (se envía adjunta esta imagen en el anexo, debido a que no se ve nada desde WORD):



Que no podemos insertar en los nodos esclavos:



Y que podemos leer en los nodos esclavos lo que insertemos en los maestros:



**Realizar una consulta sobre el MAESTRO1 que permita obtener el nombre de todos los grupos musicales junto con sus discos que hayan realizado por lo menos algún concierto en toda la base de datos distribuida. Explicar cómo se resuelve la consulta y su plan de ejecución.**

Entendemos que esta consulta debe de realizarse desde otro nodo. Por tanto, ejecutamos la siguiente consulta desde MAESTRO1.

La consulta es la siguiente (recordemos que la IP 192.168.182.130 referencia a MAESTRO2):

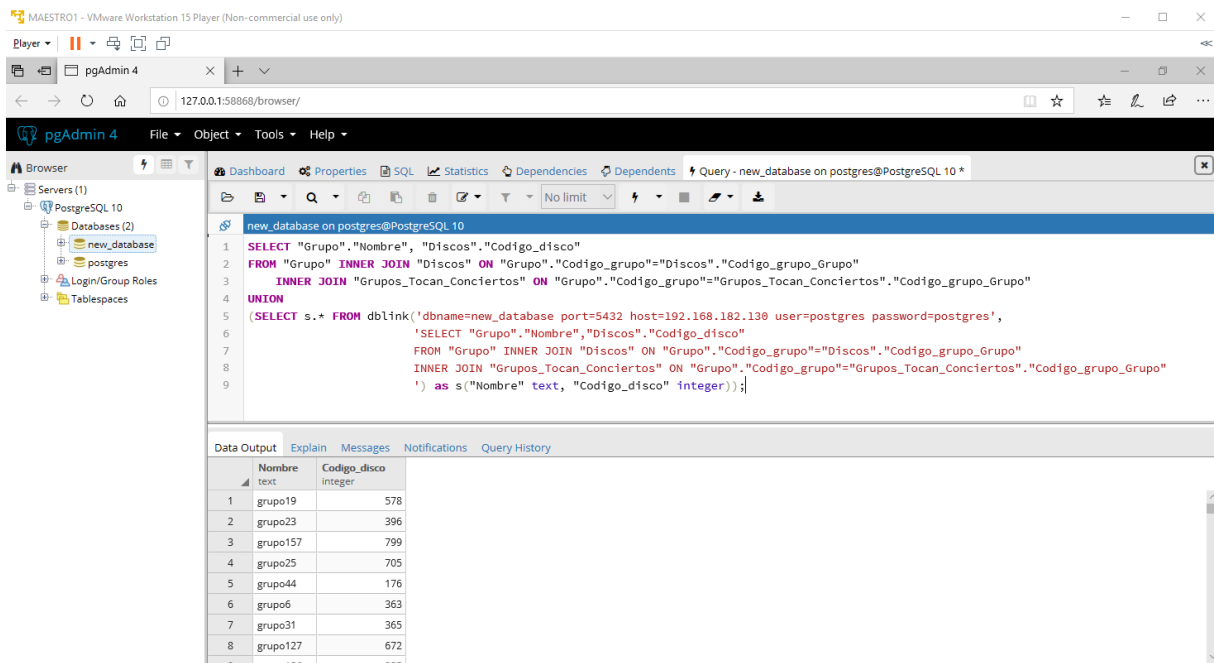
```
SELECT "Grupo"."Nombre", "Discos"."Codigo_disco"
FROM "Grupo" INNER JOIN "Discos" ON
"Grupo"."Codigo_grupo"="Discos"."Codigo_grupo_grupo"
```

```

INNER JOIN "Grupos_Tocan_Conciertos" ON
"Grupo"."Codigo_grupo"="Grupos_Tocan_Conciertos"."Codigo_grupo_Grupo"
UNION
(SELECT s.* FROM dblink('dbname=new_database port=5432
host=192.168.182.130 user=postgres password=postgres',
'SELECT
"Grupo"."Nombre","Discos"."Codigo_disco"
FROM "Grupo" INNER JOIN "Discos" ON
"Grupo"."Codigo_grupo"="Discos"."Codigo_grupo_Grupo"
INNER JOIN "Grupos_Tocan_Conciertos" ON
"Grupo"."Codigo_grupo"="Grupos_Tocan_Conciertos"."Codigo_grupo_Grupo"
') as s("Nombre" text, "Codigo_disco"
integer));

```

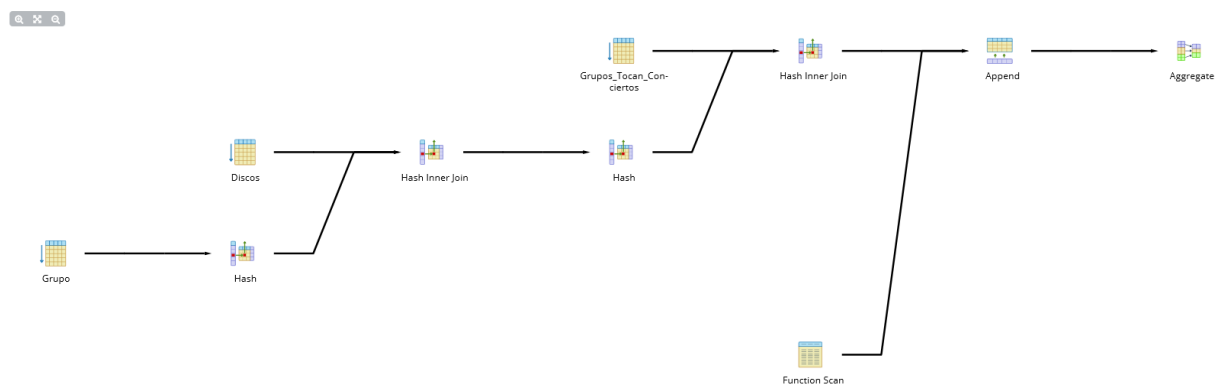
Vemos que la consulta funciona correctamente:



The screenshot shows the pgAdmin 4 interface. The query editor displays the SQL query from the previous block. The 'Data Output' tab shows the results of the query, which are as follows:

	Nombre text	Codigo_disco integer
1	grupo19	578
2	grupo23	396
3	grupo157	799
4	grupo25	705
5	grupo44	176
6	grupo6	363
7	grupo31	365
8	grupo127	672

Utilizando Explain podemos ver el plan de ejecución de la consulta:



## **Si el nodo MAESTRO1 se quedase inservible, ¿Qué acciones habría que realizar para poder usar completamente la base de datos en su modo de funcionamiento normal? ¿Cuál sería la nueva configuración de los nodos que quedan?**

Primero de nada, hemos realizado este apartado de la práctica con MAESTRO2 en lugar de MAESTRO1 debido a que la máquina virtual de este último se ha dañado.

En primer lugar, hemos añadido un nuevo registro en MAESTRO2 para comprobar que se replicaba en ESCLAVO2 debidamente.

Una vez verificado, hemos tirado MAESTRO2.

A partir de aquí, se pueden hacer distintas cosas. En primer lugar, se podría generar un backup desde ESCLAVO2 y meterlo en MAESTRO2.

En segundo lugar, se podría utilizar ESCLAVO2 como MAESTRO2. Para ello habría que reconfigurar los archivos postgresql.conf y pg\_hba.conf para que tuviesen todos los permisos de escritura y generasen los WALs. Además, se tendría que modificar la IP para que las consultas de presuntas aplicaciones fuesen a parar al antiguo ESCLAVO2 (ahora MAESTRO2).

A su vez, se podría tirar el cluster de MAESTRO2 y copiopegar el cluster de ESCLAVO2, modificándolo para estar en modo de replicación, de forma que lo que acaba resultando es un cambio de roles: ESCLAVO2 pasa a ser MAESTRO2 y MAESTRO2 pasa a ser ESCLAVO2.

## **Según el método propuesto por PostgreSQL, ¿podría haber inconsistencias en los datos entre la base de datos del nodo maestro y la base de datos del nodo esclavo? ¿Por qué?**

Tal y como se ha configurado, podría haber una inconsistencia de datos entre MAESTRO1 y MAESTRO2. Es decir, si yo inserto una tupla con una PK n en MAESTRO1 y unos datos, y otra tupla en MAESTRO2 con una PK n pero distintos datos, se estarían creando datos distintos con misma PK, por lo que se induciría a errores a la hora de hacer consultas.

Esto se podría solucionar con algún script intermedio que filtre consultas y cree prohibiciones para que no se den este tipo de situaciones.

Por otra parte, al estar haciendo WALs a lo largo del tiempo, se debe estudiar el uso que recibe la base de datos para llegar a un compromiso entre número de replications y número de consultas. Si la base de datos recibe un número enormemente superior de consultas respecto a la cantidad de replications que se hacen, si el nodo maestro cayese y hubiese que recuperar los datos del nodo esclavo, se producirían inconsistencias y pérdidas de datos.

Claro que, si a su vez generas demasiada replicación, estarás sobrecargando el sistema.

## Conclusiones.

Las primeras anotaciones van referidas a la virtualización.

Hemos tenido problemas al realizar la práctica en relación con las máquinas virtuales. Sugeriría que se aconsejase a usar VMWare Workstation **PRO**, ya que soluciona bastantes problemas de interconexión entre las máquinas virtuales al tener un editor de redes integrado.

El segundo problema se halla en virtualizar 4 máquinas y configurar postgres en cada una de ellas. Sería más práctico tener ya una máquina virtual configurada y copiopegar el cluster en las otras, en lugar de hacer todos los pasos en cada una de ellas.

Respecto a la esencia de la práctica en sí:

Encontramos que existen numerosas soluciones para conseguir una seguridad y replicación de los datos, y entendemos que solo hemos usado una de estas soluciones. Seguramente cada una de ellas sea efectiva para distintos tipos de situaciones.

Dentro de lo que hemos usado, encontramos útil haber estudiado los distintos tipos de backups que se pueden hacer de una base de datos. Entender estos distintos sistemas es importante si el día de mañana debemos trabajar con ello.

De estos backups, destacamos 2:

- Realizar un `pg_dump` especialmente útil para migrar bases de datos pequeñas. Puede ser útil para migrar, más que datos en sí, configuraciones para una base de datos distribuida.
- Realizar un backup con la herramienta `pg_basebackup` es útil para salvar la configuración y estado de un clúster en un momento determinado.

Por la parte de conexión entre bases de datos, vemos que conectar BBDD por parte de la configuración de postgres es relativamente sencilla y es muy práctico a la hora de tener bases de datos distribuidas. Lo cual a la vez es más seguro y contiene una mayor tolerancia a fallos que tener todo en un único servidor.

Por poner un ejemplo práctico, si tuviésemos distintas promotoras y una cayese, no dejamos al resto sin poder trabajar por no tener los datos necesarios.

Respecto a tema de replicación de datos, es útil tener una arquitectura cuya infraestructura salve los datos continuamente. Esto permite, de nuevo, una gran tolerancia frente a fallos.

En conclusión final, estas herramientas son útiles de cara a desarrollar bases de datos con una gran tolerancia a fallos y disponibilidad.

## Bibliografía encontrada para realizar la práctica

- Dblink: <https://stackoverflow.com/questions/46324/possible-to-perform-cross-database-queries-with-postgres>

- FDW: <http://www.craigkerstiens.com/2013/08/05/a-look-at-FDWs/>
- VMWARE:
  - Conexión entre máquinas: <https://youtu.be/neZbPuHbuYY>
  - Evitar actualizaciones de Windows: <https://www.tenforums.com/tutorials/8013-enable-disable-windows-update-automatic-updates-windows-10-a.html>
- Otros:
- <https://www.linode.com/docs/databases/postgresql/how-to-back-up-your-postgresql-database/>
- <https://medium.com/leboncoin-engineering-blog/managing-postgresql-backup-and-replication-for-very-large-databases-61fb36e815a0>

La memoria debe ser especialmente detallada y exhaustiva sobre los pasos que el alumno ha realizado y mostrar evidencias de que ha funcionado el sistema.

## **Bibliografía**

- Capítulo 25: Backup and Restore.
- Capítulo 26: High Availability, Load Balancing, and Replication.
- Appendix F: Additional Supplied Modules. F.11. dblink
- Appendix F: Additional Supplied Modules. F.34. Postgres\_fdw