

**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:** Luis Alejandro Cabanillas Prudencio

**DNI:** 04236930P

**ALUMNO 2:**

**Nombre y Apellidos:** Álvaro de las Heras Fernández

**DNI:** 03146833L

**Fecha:** 20-01-2019

**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**

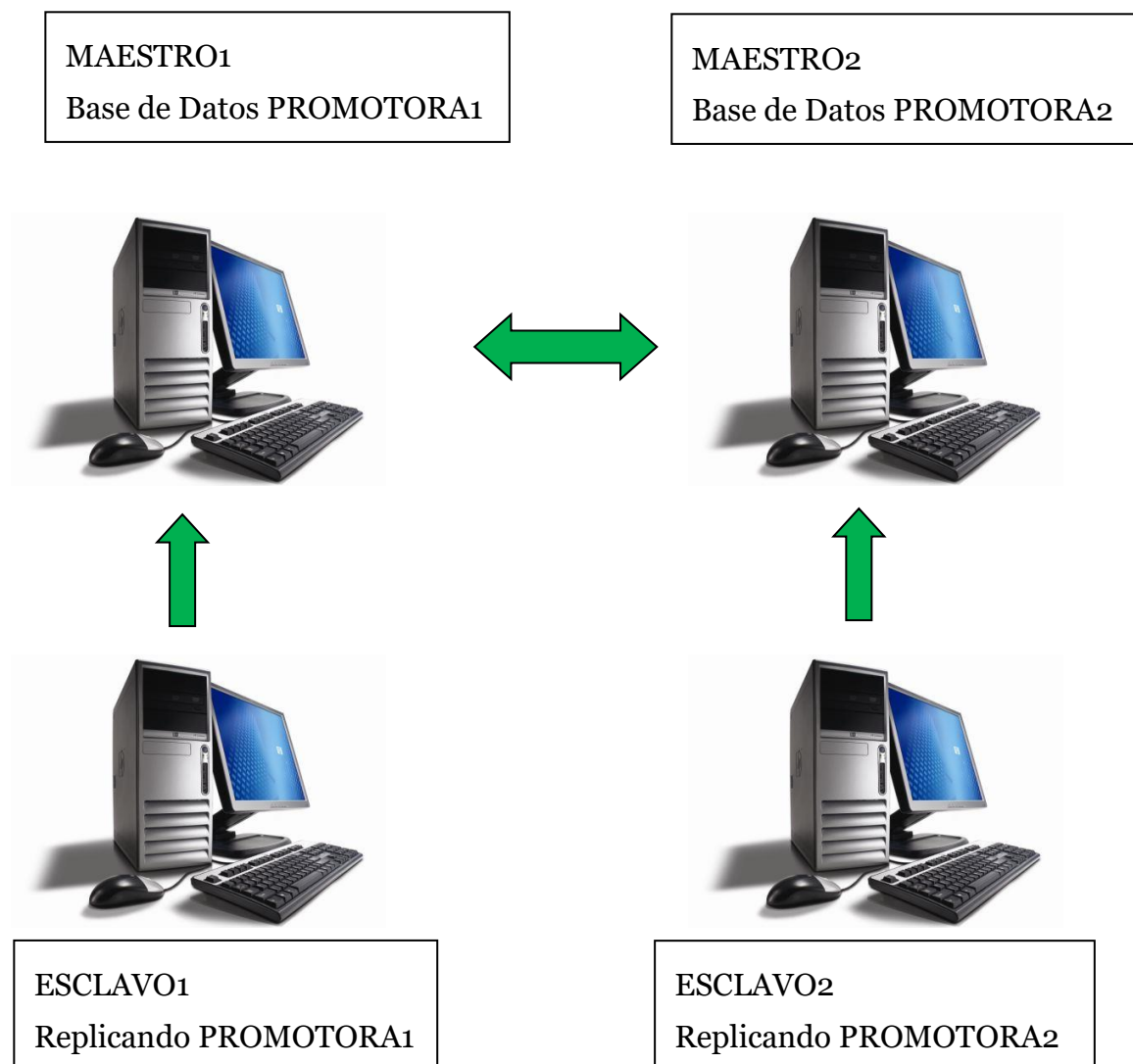
**AMBOS ALUMNOS DEBEN ENTREGAR EL FICHERO EN LA PLATAFORMA.**

## 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:

1. ¿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.

PostgreSQL pone a nuestra disposición una gran variedad de métodos para la replicación y aseguramiento de la disponibilidad de las bases de datos. Estos métodos son los siguientes, además de otros métodos que ofrecen otras empresas:

- **Shared Disk Failover**

Este método ofrece un sistema de recuperación rápido ante un fallo sin pérdida de datos al tener un servidor en espera. Sin embargo, solo permite un único array de discos, por lo que, si falla o se corrompe el sistema entero, lo hacen.

- **File System (Block Device) Replication**

Es una modificación del anterior que permite ir copiando los cambios que ocurren en los archivos en otro dispositivo, asegurando que sean correctos. Así se solucionan los problemas de corrupción y fallo.

- **Write-Ahead Log Shipping**

Este método es capaz de permitir operaciones de lectura mientras se está recuperando el servidor (hot standby). Además, permite elegir entre asíncrono o síncrono (este último no genera pérdida de datos). Si falla el esclavo rápidamente puede convertirse en maestro con casi toda la totalidad de los datos.

- **Logical Replication**

Este crea un canal de datos entre los servidores con los cambios que ocurren en las tablas a partir del WAL. Además, el canal es bidireccional por lo que no hace falta definir maestro ni esclavo.

- **Trigger-Based Máster-Standby Replication**

Es un método maestro-esclavo que envía datos de forma asíncrona al esclavo. Este esclavo solo puede realizar consultas de lectura, el resto las hace el maestro.

- **Statement-Based Replication Middleware**

En este caso hay un programa que se encarga de recibir la consulta y enviársela a todos los servidores, que trabajan de forma independiente. También se puede configurar para hacer que redirijas las lecturas a un servidor, aunque todas las que modifiquen tienen que ser enviadas a todos los servidores

- **Asynchronous Multimaster Replication**

Se usa para servidores que no se conectan muy a menudo permitiendo un uso independiente con algunas comunicaciones para los conflictos. Estos se resolverán por los usuarios o con reglas.

- **Synchronous Multimaster Replication**

En este cualquier servidor puede aceptar la consulta, sin importar el tipo, para después enviar los datos modificados al resto antes de que se haga COMMIT. Como es posible deducir si hay muchas escrituras esto provocará un rendimiento muy bajo, aunque si es para lectura no hay estos problemas.

La opción que hemos elegido es Write-Ahead Log Shipping porque nos ofrece hot-standby, es decir, lectura mientras se replica, es rápido ante caídas y puede ser síncrono o asíncrono. Además, tenemos que descartar todas las opciones que no emplean maestro-esclavo como los multimaster, logical replication y Statement-Based Replication Middleware.

Trigger-Based Máster-Standby Replication este método le hemos descartado porque era asíncrono y el esclavo solo realiza lecturas.

Shared Disk Failover y File System Replication han sido descartados porque el primero no otorga seguridad y el segundo porque es estricto en el proceso de mirror.

Por tanto, nuestra opción elegida es Write-Ahead Log Shipping por las ventajas que ofrece.

2. ¿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.

PostgreSQL ofrece dos métodos para realizar consultas a servidores externos, estos son:

- **Dblink.** Este método es muy sencillo de implementar al requerir menos configuración para conectarse a otras bases de datos. A diferencia de `postgres_fdw`, no soporta el modo de solo lectura, tampoco sigue el standard SQL y es menos eficiente que éste último. Aunque es compatible con versiones antiguas y las conexiones solo son válidas para la sesión, que puede ser una ventaja.
- **postgres\_fdw.** Este método se implementó a partir de la versión 9.3 de PostgreSQL. Como hemos comentado, es una opción que mejora a `dblink` en el modo de lectura, siguiendo el estándar y siendo más eficiente. Las ventajas que tiene son la retrocompatibilidad y las conexiones persistentes. Por lo demás funciona igual que `dblink` permitiendo conectarse a realizar consultas a otras bases de datos.

En nuestro caso usaremos `postgres_fdw` al ser la opción que más ventajas aporta y ser compatible con PostgreSQL 10.

3. 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.

Para configurar los nodos maestros debemos permitir que reciban conexiones de otros equipos, para ello modificamos postgresql.conf y pg\_hba.conf como hicimos en la anterior práctica, configurando cada uno del siguiente modo:

Para configurar la conexión remota es necesario modificar la configuración de nuestro PostgreSQL para ello cambiaremos el fichero postgresql.conf. Tendremos que poner listen\_addresses = '\*' para que así se puedan atender todas las conexiones y poner el puerto que queremos, en nuestro caso el que usa PostgreSQL por defecto port = 5432. Otro añadido más es el de limitar el número de conexiones con max\_connections. Aquí la captura de nuestro fichero:

```
# - 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)
```

Una vez cambiado eso, modificaremos el archivo pg\_hba.conf para especificar qué IP's queremos que puedan acceder, cambiando la configuración para el host:


#	TYPE	DATABASE	USER	ADDRESS	METHOD
# IPv4 local connections:					
host	all	all	0.0.0.0/0	md5	

Con la IP 0.0.0.0/0 permitimos la conexión de cualquier IP, si quisiéramos restringirlo usaríamos la IP del equipo que queramos que acceda. Md5 es el tipo de seguridad que queremos que se emplee para la contraseña.

Para que los cambios se completen hace falta que reiniciemos el ordenador. Ya solo nos hace falta nuestra IP que, como estamos en una red DHCP, solo será válida para nuestra red, para fuera de ella no.


```
Dirección IPv4. . . . . : 192.168.1.37(Preferido)  
Máscara de subred . . . . . : 255.255.255.0  
Concesión obtenida. . . . . : miércoles, 9 de enero de 2019 13:47:18  
La concesión expira . . . . . : jueves, 10 de enero de 2019 7:47:19
```

Además, tuvimos otro problema relacionado con el firewall, por lo que cuando lo desactivamos pudimos realizarlo sin problema.

 **Red de dominio**

El firewall está desactivado.

Activar

 **Red privada (activa)**

El firewall está desactivado.

Activar

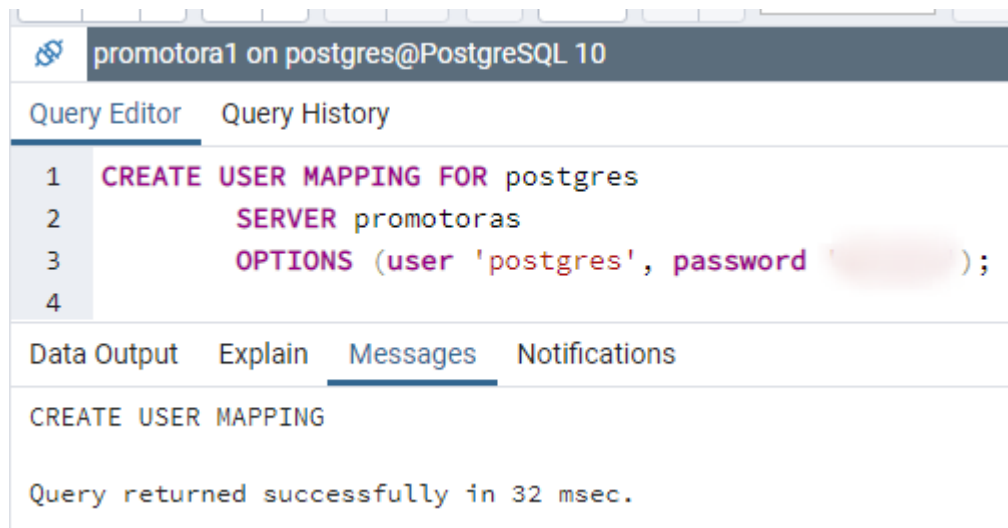
Ahora hay que configurar postgres\_fdw para poder realizar consultas remotas. Lo primero de todo es añadir la extensión:

```
promotora1 on postgres@PostgreSQL 10
Query Editor  Query History
1 CREATE EXTENSION postgres_fdw;
2
Data Output  Explain  Messages  Notifications
CREATE EXTENSION
Query returned successfully in 87 msec.
```

Después nos toca configurar el servidor que habrá entre los maestros poniendo los datos que queremos obtener:

```
promotora1 on postgres@PostgreSQL 10
Query Editor  Query History
1 CREATE SERVER promotoras
2     FOREIGN DATA WRAPPER postgres_fdw
3     OPTIONS (host '192.168.1.36', port '5432', dbname 'promotora2');
4
Data Output  Explain  Messages  Notifications
CREATE SERVER
Query returned successfully in 47 msec.
```

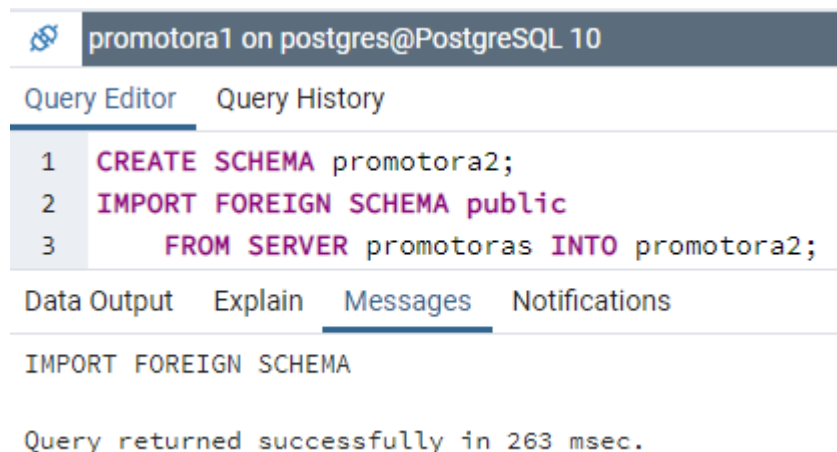
Finalmente, mapeamos el usuario que se registrará en la otra base de datos (hay que asegurarse en el otro ordenador que este dado de alta) y ya se podrían obtener los datos:



```
promotora1 on postgres@PostgreSQL 10
Query Editor  Query History
1  CREATE USER MAPPING FOR postgres
2      SERVER promotoras
3      OPTIONS (user 'postgres', password ' ');
4
Data Output  Explain  Messages  Notifications
CREATE USER MAPPING

Query returned successfully in 32 msec.
```

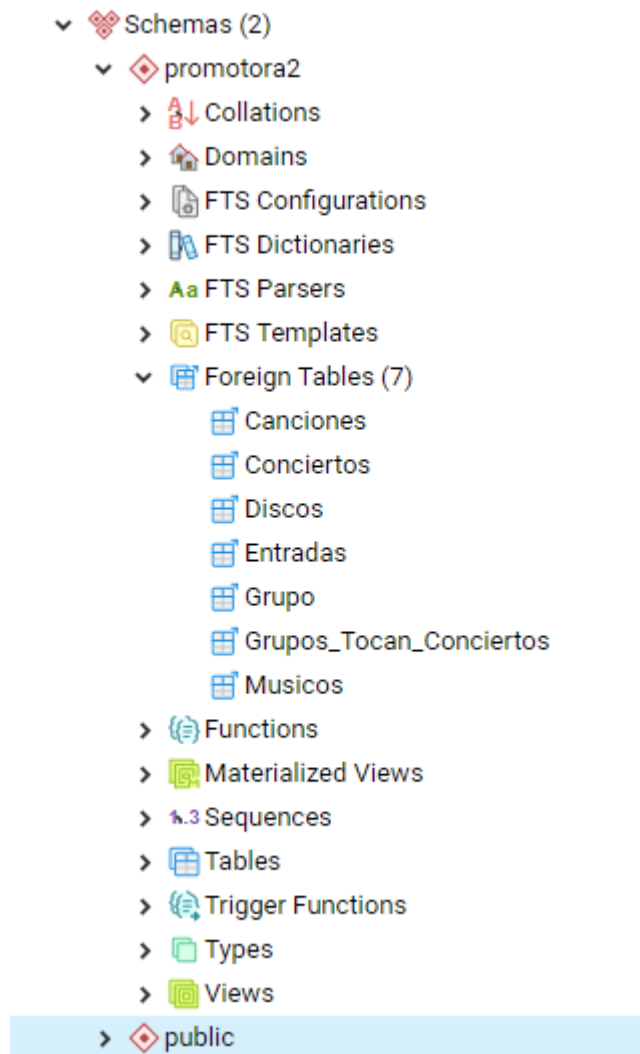
Probamos a ver los datos, para ello tenemos que importarlo en foreign tables que contienen foreign data que se obtiene a través de postgres\_fdw.



```
promotora1 on postgres@PostgreSQL 10
Query Editor  Query History
1  CREATE SCHEMA promotora2;
2  IMPORT FOREIGN SCHEMA public
3      FROM SERVER promotoras INTO promotora2;

Data Output  Explain  Messages  Notifications
IMPORT FOREIGN SCHEMA

Query returned successfully in 263 msec.
```



Ahora hay que repetir el proceso intercambiando los datos para el otro ordenador que representará a la otra promotora.

4. 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.

### Configuración del maestro

En este caso tenemos que configurar primero el nodo maestro para permitir tener hot-standby. Lo primero es modificar los archivos de configuración `pg_hba.conf` y `postgresql.conf`.

Para ello añadimos un nuevo usuario que se le permita replicación con el usuario que lo hace como se ve a continuación, en `pg_hba`. Al ponerlo en trust no hace falta ningún tipo de verificación.



#	TYPE	DATABASE	USER	CIDR-ADDRESS	METHOD
# IPv4 local & remote connections:					
host	all		all	127.0.0.1/32	trust
host	all		all	0.0.0.0/0	md5
host	replication		postgres	192.168.1.39/32	trust
# IPv6 local connections:					
host	all		all	:::1/128	trust

Ahora modificaremos postgresql.conf para permitir obtener los archivos WAL para este sistema de replicación. Modificando archive\_mode para permitir la replicación, wal\_level en el modo replica o logical, archive\_command para indicar la ruta para almacenar los WAL y max\_wal\_senders para permitir enviar datos al esclavo.

```
#-----
# - Settings -
wal_level = replica # (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
# fdatsync (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 = 'true' # command to use to archive a logfile segment
# placeholders: %p = path of file to archive
# %f = file name only
# e.g. 'test ! -f /mnt/server/archivedir/%f && cp %p /mnt/server/archivedir/%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
max_wal_senders = 5
# (change requires restart)
wal_keep_segments = 32
#wal_sender_timeout = 60s # in milliseconds; 0 disables

max_replication_slots = 5
# (change requires restart)
#track_commit_timestamp = off # collect timestamp of transaction commi
# (change requires restart)

# - Master Server -

# These settings are ignored on a standby server.

#synchronous_standby_names = '' # standby servers that provide sync rep
# method to choose sync standbys, number of sync standby
# and comma-separated list of application_name
# from standby(s); '*' = all
#vacuum_defer_cleanup_age = 0 # number of xacts by which cleanup is de

```

## Configuración del esclavo o standby

Siempre hay que asegurarse de que tengan la misma arquitectura los sistemas operativos que van a mover el servidor, porque si no se producen errores por las diferencias entre estas.

Lo primero de todo será detener el servidor porque tendremos que borrar todo el contenido para reemplazarlo con la copia de seguridad.

```

C:\Windows\system32>pg_ctl stop -D "C:\Program Files\PostgreSQL\10\data"
esperando que el servidor se detenga.... listo
servidor detenido

```

Después tenemos que obtener el backup del maestro, esto lo haremos mediante el comando **pg\_basebackup**, que nos permite obtener una copia de la base de datos del host que le indiquemos. En este caso al tenerlo configurado como trust en el archivo de pg\_hba del maestro no hace falta introducir ninguna contraseña. Esto copiará todo el contenido de la carpeta data además al indicar como parámetros **-v** obtendremos más información acerca del backup en pantalla, al indicar **-P** nos mostrará el progreso y al pasar como parámetro **-wal-method=stream** que creará un canal extra para transmitir el WAL.

```
C:\Windows\system32>pg_basebackup -h 192.168.1.37 -p 5432 -U postgres -D "C:\Program Files\PostgreSQL\10\data" -v -P --wal-method=stream
pg_basebackup: iniciando el respaldo base, esperando que el checkpoint se complete
pg_basebackup: el checkpoint se ha completado
pg_basebackup: punto de inicio del WAL: 0/9000028 en el timeline 1
pg_basebackup: iniciando el receptor de WAL en segundo plano
30739/30739 kB (100%), 1/1 tablespace
pg_basebackup: posición final del WAL: 0/90000F8
pg_basebackup: esperando que el proceso en segundo plano complete el flujo...
pg_basebackup: el respaldo base se ha completado
```

Una vez hecho el backup lo sustituimos en la carpeta de data, en este caso al operar con distintas versiones es necesario realizar unos ajustes en los directorios para que sea capaz de encontrar los logs después.

Ahora es cuando en el archivo crearemos `recovery.conf`, para indicar al servidor standby que se encuentra en modo recuperación para ello modificamos el que trae de muestra PostgreSQL.

Los cambios que vamos a realizar son los siguientes en el recovery. Primero indicar que se encuentra en modo standby, con **standby\_mode**, y segundo que necesitará conectarse al maestro para obtener los archivos WAL, mediante **primary\_conninfo**, que rellenaremos con los datos del maestro.

```
# standby_mode = on
#
# primary_conninfo
#
# If set, the PostgreSQL server will try to connect to the primary using this
# connection string and receive XLOG records continuously.
#
primary_conninfo = 'host=192.168.1.37 port=5432 user=postgres password=philips'
#
```

También pondremos el modo hot-standby en postgresql.conf como se ve a continuación:

[illegible]

Ahora que ya está configurado nos queda poner en marcha el servidor para comprobar que se ha realizado adecuadamente.

```
A:\Windows\system32>pg_ctl start -D "C:\Program Files\PostgreSQL\10\data\pg10"
esperando que el servidor se inicie...2019-01-19 13:27:48 CET [4248]: [1-1] user=,db=,app=,client= LOG:  listening on IPv6 address ":::", port 5432
2019-01-19 13:27:48 CET [4248]: [2-1] user=,db=,app=,client= LOG:  listening on IPv4 address "0.0.0.0", port 5432
2019-01-19 13:27:48 CET [4248]: [3-1] user=,db=,app=,client= LOG:  redirigiendo la salida del registro al proceso recolector de registro
2019-01-19 13:27:48 CET [4248]: [4-1] user=,db=,app=,client= HINT:  La salida futura del registro aparecerá en el directorio «C:/POSTGR~1/data/logs/pg10».
    listo
servidor iniciado
C:\Windows\system32>
```

```
C:\Windows\system32>psql -h localhost -p 5432 -U postgres
psql (10.6)
ADVERTENCIA: El código de página de la consola (850) difiere del código de página de Windows (1252).
Los caracteres de 8 bits pueden funcionar incorrectamente.
Vea la página de referencia de psql «Notes for Windows users» para obtener más detalles.
Dígame «help» para obtener ayuda.

postgres=# \l

```

Nombre	Dueño	Codificación	Collate	Ctype	Privilegios
postgres	postgres	UTF8	Spanish_Spain.1252	Spanish_Spain.1252	
promotor1	postgres	UTF8	Spanish_Spain.1252	Spanish_Spain.1252	
template0	postgres	UTF8	Spanish_Spain.1252	Spanish_Spain.1252	=c/postgres + postgres=CTc/postgres
template1	postgres	UTF8	Spanish_Spain.1252	Spanish_Spain.1252	=c/postgres + postgres=CTc/postgres

```
(4 filas)
postgres=#
```

Además, si queremos comprobar que ha hecho el recovery correctamente podemos revisar el log del servidor para verlo como hemos hecho:

```
[1-1] user=,db=,app=,client= LOG:  el sistema de bases de datos fue interrumpido; última vez en funcionamiento en 2019-01-
[2-1] user=,db=,app=,client= LOG:  redo comienza en 0/9000028
[3-1] user=,db=,app=,client= LOG:  el estado de recuperación consistente fue alcanzado en 0/90000F8
[4-1] user=,db=,app=,client= LOG:  largo de registro no válido en 0/A000060: se esperaba 24, se obtuvo 0
[5-1] user=,db=,app=,client= LOG:  redo listo en 0/A000028
[6-1] user=,db=,app=,client= LOG:  checkpoint starting: end-of-recovery immediate
[7-1] user=,db=,app=,client= LOG:  checkpoint complete: wrote 0 buffers (0.0%); 0 WAL file(s) added, 0 removed, 0 recycled
[5-1] user=,db=,app=,client= LOG:  el sistema de bases de datos está listo para aceptar conexiones
```

Ya estaría configurado uno, ahora toca repetir el mismo proceso en el otro ordenador y máquina virtual.

5. 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.

Las operaciones que puedan realizar cada servidor dependerán del tipo de nodo que sean. Si se trata de un nodo maestro, este podrá realizar cualquier operación sobre el conjunto de datos, como INSERT, CREATE, UPDATE, DELETE entre otras, sin ninguna restricción. Mientras que los nodos esclavos sí que estarán limitados en operaciones porque solo son read-only, es decir, que lo único que pueden hacer es leer datos que les envíe el maestro, sin poder realizar modificación alguna.

Si nos centramos ahora en las operaciones que podrán realizarse con los datos del otro nodo maestro tendremos las siguientes INSERT, UPDATE, DELETE y SELECT, porque se restringe a los datos de las tablas, que en nuestro caso se hará empleando foreign tables, al usar el método postgres\_fdw. Esto lo hemos comprobado realizando

distintas operaciones como un INSERT que sí que ha sido válida y un ALTER TABLE que ha provocado un error.

promotora1 on postgres@PostgreSQL 10

Query Editor Query History

1

Data Output Explain Messages Notifications

ERROR: no existe la columna «data»  
CONTEXT: Remote SQL command: SELECT "Codigo\_cancion", "Nombre", "Compositor", "Fecha\_grabacion", "Duracion", "Codigo\_disco\_Discos", data FROM public."Canciones"  
SQL state: 42703

**Canciones**

General Definition Columns Constraints Options Security SQL

Columns

	Name	Data Type	Inherited From
<input checked="" type="checkbox"/>	Codigo_cancion	integer	
<input checked="" type="checkbox"/>	Nombre	text	
<input checked="" type="checkbox"/>	Compositor	text	
<input checked="" type="checkbox"/>	Fecha_grabacion	date	
<input checked="" type="checkbox"/>	Duracion	time without time z...	
<input checked="" type="checkbox"/>	Codigo_disco_Discos	integer	
<input checked="" type="checkbox"/>	data	text	

Cancel Reset Save

promotora1 on postgres@PostgreSQL 10

Query Editor Query History

```
1 INSERT INTO promotora2."Grupo"(  
2     "Codigo_grupo", "Nombre", "Genero_musical", "Pais", "Sitio_web")  
3     VALUES (10, 'Músicos', 'Rock', 'Qatar', 'mus.com');
```

Data Output Explain Messages Notifications

INSERT 0 1

Query returned successfully in 109 msec.

Las distintas situaciones de caída que se pueden producir son las siguientes:

### Caída de nodo esclavo

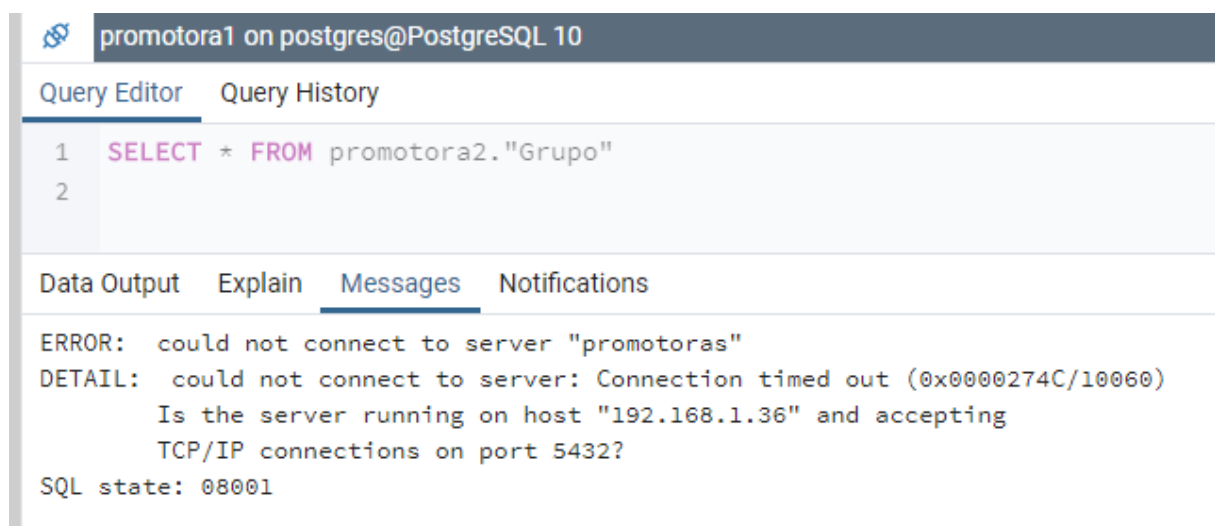
Para simular la caída de un nodo esclavo es tan simple como cerrar la máquina virtual que lo contenga. En este caso su caída no tendrá ningún impacto sobre el funcionamiento del sistema porque sólo se dedica a leer y almacenar registros WAL. Sin embargo, sí que podríamos tener un problema si su caída se combinara con la caída del nodo maestro, porque se podrían perder datos llegando a ser irrecuperables. Si se recuperase después se tendrían que actualizar los datos mediante recovery.

### Caída de nodo maestro

La caída de un nodo maestro nos conlleva ya más problemas porque al ser distribuida está conectada a otro maestro. Para simularla detendremos el servidor de PostgreSQL con el comando `pg_ctl stop`, deteniendo el servidor para así mantener el esclavo. En este caso el esclavo seguirá funcionando correctamente porque es independiente (en nuestro caso no lo es al ser una máquina virtual, pero hacemos que actúe como tal) e incluso podría comenzar con los procesos de recuperación ante caídas, al detectar los fallos en las conexiones.

```
2019-01-20 01:17:03 CET [2740]: [1-1] user=,db=,app=,client= FATAL: no se pudo hacer la conexión al servidor primario: no s
¿Está el servidor en ejecución en el servidor «192.168.1.37» y aceptando
conexiones TCP/IP en el puerto 5432?
```

Pero el otro nodo maestro ya no podrá acceder a los datos que teníamos en este maestro ni tampoco a los de su esclavo, por lo que tenemos que aplicar estos procesos de recuperación. Sino no será posible establecer la conexión, como se puede ver en nuestro caso al intentar acceder al foreign data wrapper que tenemos:



The screenshot shows a PostgreSQL Query Editor interface. At the top, it says 'promotora1 on postgres@PostgreSQL 10'. Below that, there are tabs for 'Query Editor' and 'Query History'. The 'Query Editor' tab is active, showing a SQL query: `1 SELECT * FROM promotora2."Grupo"` and `2`. Below the query editor, there are tabs for 'Data Output', 'Explain', 'Messages', and 'Notifications'. The 'Messages' tab is active, showing an error message: `ERROR: could not connect to server "promotoras"`, `DETAIL: could not connect to server: Connection timed out (0x0000274C/10060)`, `Is the server running on host "192.168.1.36" and accepting`, `TCP/IP connections on port 5432?`, and `SQL state: 08001`.

Lo que tendríamos que hacer para corregir esto es convertir al nodo standby en el nodo maestro, para ello tenemos que usar el comando `pg_ctl promote`, que podemos activar manualmente o con un trigger, esto hará que acabe el modo standby y se le permitan más operaciones. Aunque todo esto conlleva cierto tiempo y tener que comunicar al antiguo maestro si se recuperase que ahora ya no lo es. Además de volver a configurar las IP de los `postgres_fdw` y si fuera necesario los `pg_hba.conf`.

Estas podrán ocurrir también combinadas entre ellas, pero los resultados serán similares.

6. 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.

La consulta que realizaremos será un INNER JOIN sobre tres tablas por el campo de codigo\_grupo.

promotora1 on postgres@PostgreSQL 10

Query Editor Query History

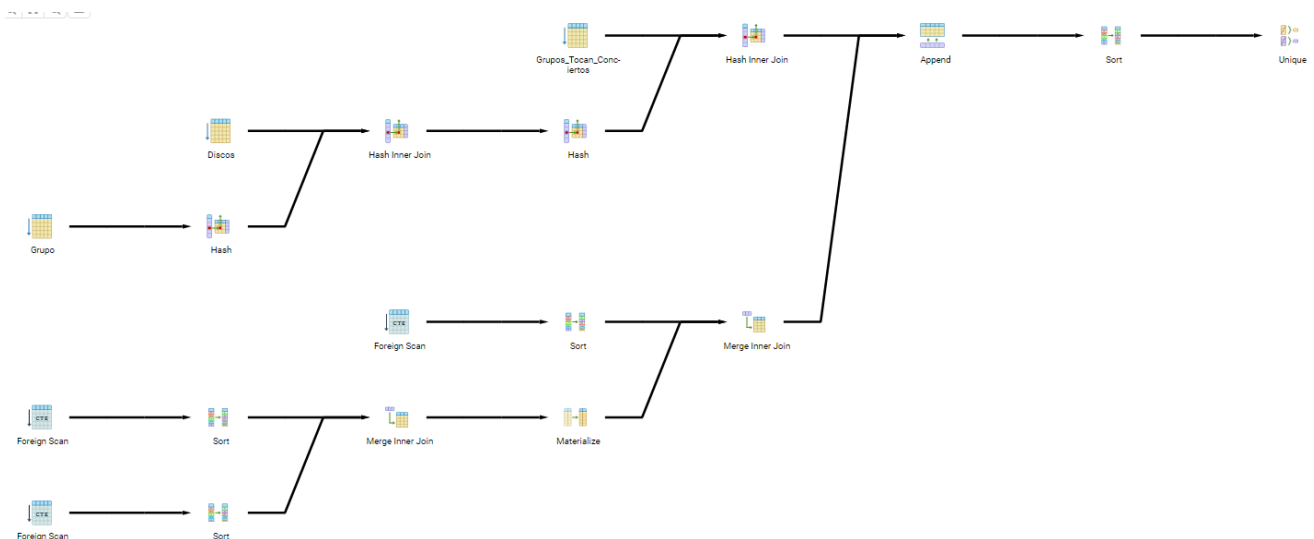
```
1 SELECT "Grupo"."Nombre", "Discos"."Titulo"
2 FROM public."Grupo"
3     INNER JOIN public."Discos" ON public."Grupo"."Codigo_grupo"="Discos"."Codigo_grupo_Grupo"
4     INNER JOIN "Grupos_Tocan_Conciertos" ON "Grupo"."Codigo_grupo"="Grupos_Tocan_Conciertos"."Codigo_grupo_Grupo"
5 UNION
6 SELECT promotora2."Grupo"."Nombre", promotora2."Discos"."Titulo"
7 FROM promotora2."Grupo"
8     INNER JOIN promotora2."Discos" ON promotora2."Grupo"."Codigo_grupo"=promotora2."Discos"."Codigo_grupo_Grupo"
9     INNER JOIN promotora2."Grupos_Tocan_Conciertos" ON promotora2."Grupo"."Codigo_grupo"=promotora2."Grupos_Tocan_Conciertos"."Codigo_grupo_Grupo"
```

Data Output Explain Messages Notifications

	Nombre text	Titulo text
1	Pedro	MixMusical
2	Trompeteros	Ay ay ay mi amor

Al emplear postgres\_fdw, las tablas del otro maestro vienen como FOREIGN TABLES, que contienen datos que no se encuentran en este servidor. Estas tablas no pueden tener el mismo nombre que las del esquema, pero en nuestro caso coincidía por lo que hemos tenido que crear otro esquema para tenerlas. Este es promotora2, su funcionamiento es el mismo que las tablas normales, a diferencia de que solo puedes trabajar con los datos, nada de modificar tablas o crearlas etc.

Si analizamos la query nos muestra esto:





Como se puede ver, la gran diferencia que hay es el foreign scan, esta operación consultará los datos de la otra base de datos. Al igual que con las tablas locales, estas tendrán sus estadísticas y costes estimados.

7. 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?

Este supuesto ya le hemos tratado antes, pero ahora vamos a desarrollarlo.

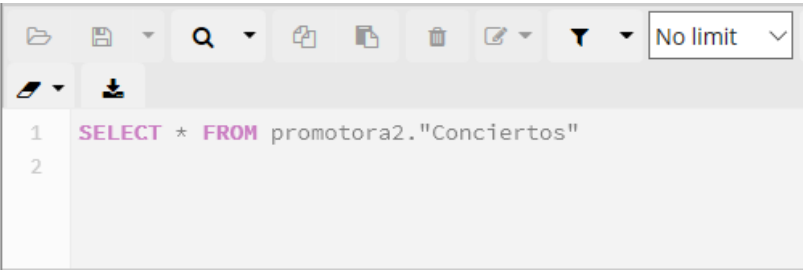
En el caso de que fallase nuestro nodo MAESTRO1, por ejemplo, el esclavo empezará con los procesos de fail over para así poder acabar el modo standby y poder atender más consultas, porque antes solo era read-only.

Sin embargo, el cómo se activa depende de la configuración. Podemos hacerlo automático con un trigger\_file en el recovery o manualmente. En nuestro caso lo haremos manualmente usando pg\_ctl promote "base".

```
C:\Windows\system32>pg_ctl promote -D "C:\Program Files\PostgreSQL\10\data\pg10"
esperando que el servidor se promueva.... listo
servidor promovido

C:\Windows\system32>
```

Una vez hecho esto, ya podría actuar como maestro, además de estar conectado con el otro nodo para mantener los datos de la base de datos distribuida. Vamos a ver si podemos conectarnos al MAESTRO2.



The screenshot shows a PostgreSQL query editor interface. At the top, there is a toolbar with icons for file operations and a search bar. Below the toolbar, a SQL query is entered: `SELECT * FROM promotora2."Conciertos"`. The query is numbered 1 and 2. Below the query editor, there are tabs for "Data Output", "Explain", "Messages", "Notifications", and "Query History". The "Data Output" tab is selected, showing a table with 6 columns: "Codigo\_concierto" (integer), "Fecha\_realizacion" (date), "Pais" (text), "Ciudad" (text), and "Recinto" (text). The table contains two rows of data.

	Codigo_concierto integer	Fecha_realizacion date	Pais text	Ciudad text	Recinto text
1	1	1939-11-06	Core...	Fiji	Recinto f...
2	2	1969-12-06	Core...	Seul	Plaza

Aunque, como hemos dicho, tenemos que tener cuidado en caso de que se vuelva a poner en marcha el MAESTRO caído: hemos de comunicarle que ahora ya no es el maestro.

Finalmente, tenemos que modificar en el otro servidor maestro la IP que le asignamos al servidor de postgres\_fdw, para que sea ahora la de éste nuevo maestro. También si no tuvieramos puesto en pg\_hba.conf que permitiera conectarse a todas las IP tendremos que añadir esta nueva.



Por tanto, la configuración resultante sería de 1 maestro, sin esclavo conectado a otro maestro con un esclavo.

8. 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?

Sí que es posible que haya inconsistencias entre ambos.

Estas inconsistencias pueden ser de dos tipos: de lectura o escritura. Además, podrán ser mucho peores si el sistema de replicación es asíncrono, aunque eso sí será más rápido. Esto se debe a que el maestro enviará de forma asíncrona los datos del WAL una vez se escriban sin esperar a que el esclavo los realice. Además, podría darse el caso de que el esclavo llegue a estar a la par del maestro, si dispone del tiempo suficiente. En este caso las inconsistencias que se pueden producir son de lectura, porque puede que lea datos del esclavo que son distintos del maestro, porque no hayan sido actualizados; y de escritura porque no hay garantías de que el WAL se haya escrito en el esclavo, pudiendo estar desincronizados por este retraso. Además, si el maestro y escribe datos, pero no envía el WAL antes de caerse provocará una pérdida de datos.

Si fuera síncrono aún puede haber inconsistencias de datos, pero concretamente en la lectura, porque espera a que se escriban los datos en el esclavo antes de enviar los datos. Esta inconsistencia se debe a que no se asegura que el cambio se haya aplicado a toda la base de datos del esclavo por lo que un cliente podría leer antes de aplicar la escritura en el esclavo.

Si quisiéramos asegurar la consistencia de datos tendríamos que aplicar el síncrono activando el modo **synchronous\_commit** en **remote\_apply**, de esta forma no solo esperaría a la escritura de los cambios, sino que también lo haría a la aplicación de estos antes de enviar los datos al cliente. Siendo la única forma posible de pérdida de datos la caída del maestro y esclavo.

En definitiva, estas son todas las posibles inconsistencias que se pueden producir y cómo evitarlas, al entender porque surgen.

## 9. Conclusiones.

En esta práctica hemos creado una base de datos distribuida con nodos hot-standby, lo que nos permite tener un mejor manejo de todo lo que puede ofrecer PostgreSQL.

Si nos centramos en la parte de la base de datos distribuida, constituida por varias remotas, es especialmente útil en varios aspectos; el primero es que nos ofrece una mayor seguridad ante caídas, porque la caída de un nodo no implica que deje de funcionar en su totalidad, sólo menos datos disponibles. El segundo que nos ofrece es una mayor facilidad para atender a muchos clientes, al no centrar todas las peticiones a un solo nodo, sino que podemos distribuir esta carga aumentando la velocidad y evitando saturación entre nodos con más peticiones y otros con menos. El último beneficio es la seguridad ante ataques, porque no está toda la información junta sino separada, haciendo más difícil su obtención.

Para ello, hemos visto las dos herramientas que PostgreSQL nos ofrece que son `dblink` y `postgres_fdw`, siendo el último el recomendado y más completo. Aunque siendo `dblink` útil para conexiones temporales y retrocompatibilidad con versiones más antiguas que no soportan `postgres_fdw`.

También hemos podido ver todos los métodos que nos ofrece PostgreSQL para la replicación de datos y su recuperación. Nosotros nos hemos centrado en `hot-standby` de tal forma que modificando los archivos de configuración hemos sido capaces de crear un servidor standby, que podemos poner en modo activo fácilmente en caso de que el maestro se caiga. Esto es muy importante para las bases de datos, para poder dar así soporte en caso de caídas y también usarlo en caso de poner el maestro en mantenimiento sin dejar de dar servicio, aunque se requiera de más servidores y espacio.

Al ver todos estos métodos también hemos aprendido qué métodos nos interesan según qué casos, como puede ser asíncrono si una pequeña inconsistencia de datos nos da igual a cambio de ser rápidos, o síncrono en caso de que queramos consistencia.

Sin embargo, también hemos tenido problemas con PostgreSQL, principalmente se han debido a que no soporta nodos standby en distintas arquitecturas, por lo que no es posible tener un nodo maestro en Windows y un esclavo en Ubuntu, por ejemplo. Al igual que hay que modificar directorios, nombres y parámetros para adaptarlo a las distintas versiones, como puede ser `pg_xlog` y `pg_wal`, entre otros.

Por tanto, podemos concluir que con esta práctica hemos aprendido como crear bases distribuidas, permitiendo la ejecución remota de consultas con `postgres_fdw`, y como hacer nodos `hot-standby`; viendo su funcionamiento y las ventajas que aportan. Además de superar los problemas que surgen con las arquitecturas y versiones.

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`