# PSAwise2324Team10Aufgabe05

## PostgreSQL

### **Passwords in /root/ on VM1**

### Installation

We installed our new database on VM1. After making sure that everything is up to date, we choose to install PostgreSQL due to personal preference and former experience via the usual command:

`sudo apt install postgresql`.

After making sure that PostgreSQL is active via `sudo systemctl status postgresql` and enabling it, we can access the database by switching to the postgres user that automatically comes with the PostgreSQL installation.
After switching to the right user, we can access PostgreSQL by running the following commands:

```
sudo -i -u postgres
psql
```

Simply run `\q` to go back to your original user and to exit the PostgreSQL terminal.

### Database Configuration

After switching to the postgres user and re-entering the PostgreSQL prompt terminal, we can create users with the following command structure:

```
CREATE USER new_username WITH PASSWORD 'user_password';
```

Our exercise sheet wants us to create two new users and two new databases. One of them should only be able to connect to one of the databases via localhost, while the other user should only be able to connect to the other database remotely from another VM in our subnet.

Set up the local user and its database:

```
# create new user
CREATE USER local with PASSWORD 'local';
# create new database
CREATE DATABASE local_db;
# grant full access
GRANT ALL PRIVILEGES ON DATABASE "local_db" to local;
```

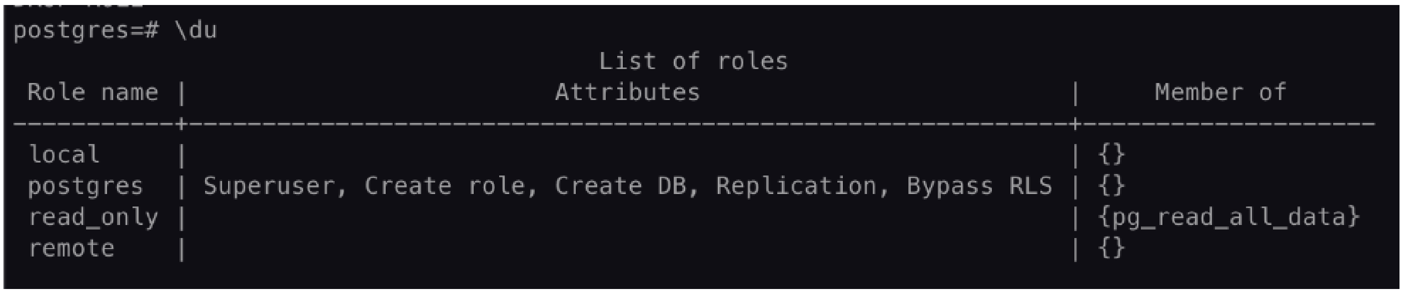
Set up the remote user and its database:

```
# create new user
CREATE USER remote with PASSWORD 'remote';
# create new database
CREATE DATABASE remote_db;
# grant full access
GRANT ALL PRIVILEGES ON DATABASE "remote_db" to remote;
```

Lastly, we need to create a third user that has read-only rights for all databases.

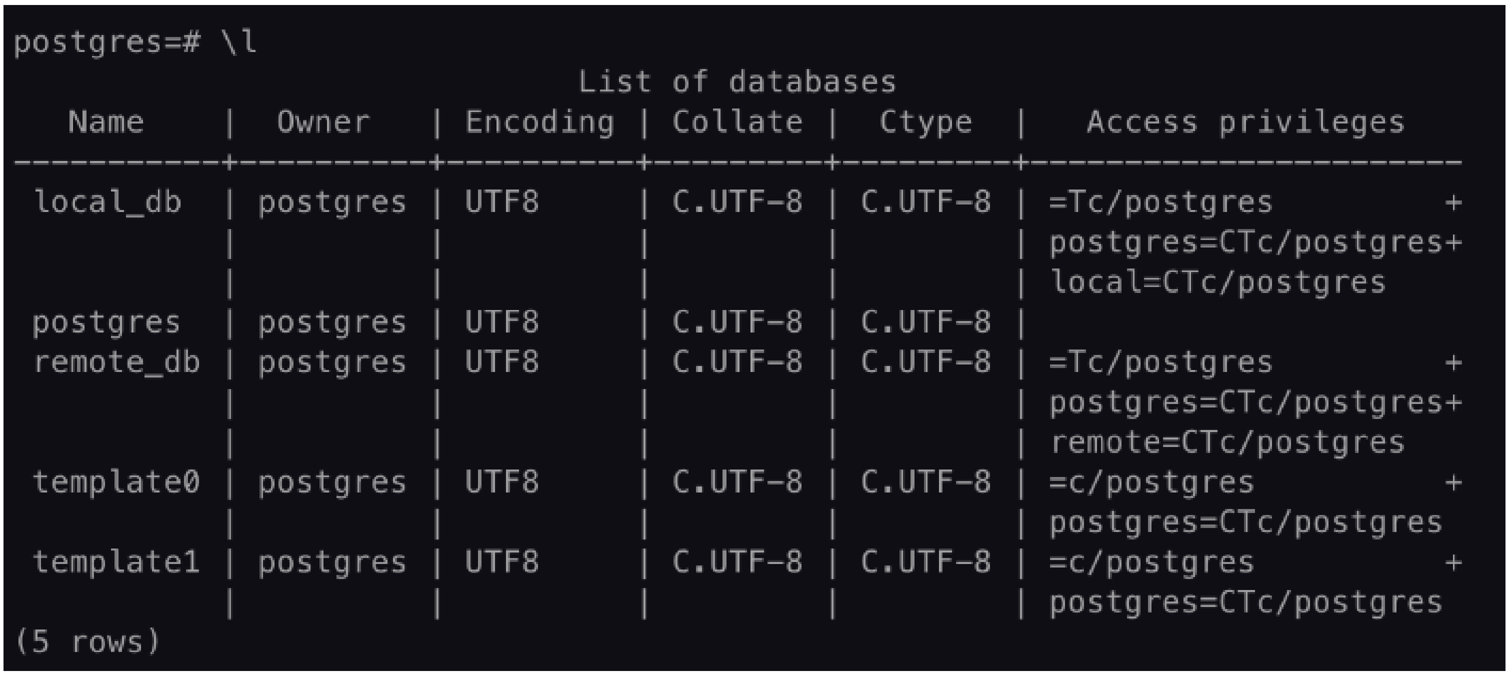
```
# create new user
CREATE USER read_only with PASSWORD 'read';
# grant read access
GRANT pg_read_all_data TO read_only;
```

We can verify the correctness of our configuration by running `\du` to list all users of our PostgreSQL server.



```
postgres=# \du
                                   List of roles
 Role name |                         Attributes                         |     Member of
-----------+------------------------------------------------------------+--------------------
 local     |                                                            | {}
 postgres  | Superuser, Create role, Create DB, Replication, Bypass RLS | {}
 read_only |                                                            | {pg_read_all_data}
 remote    |                                                            | {}
```

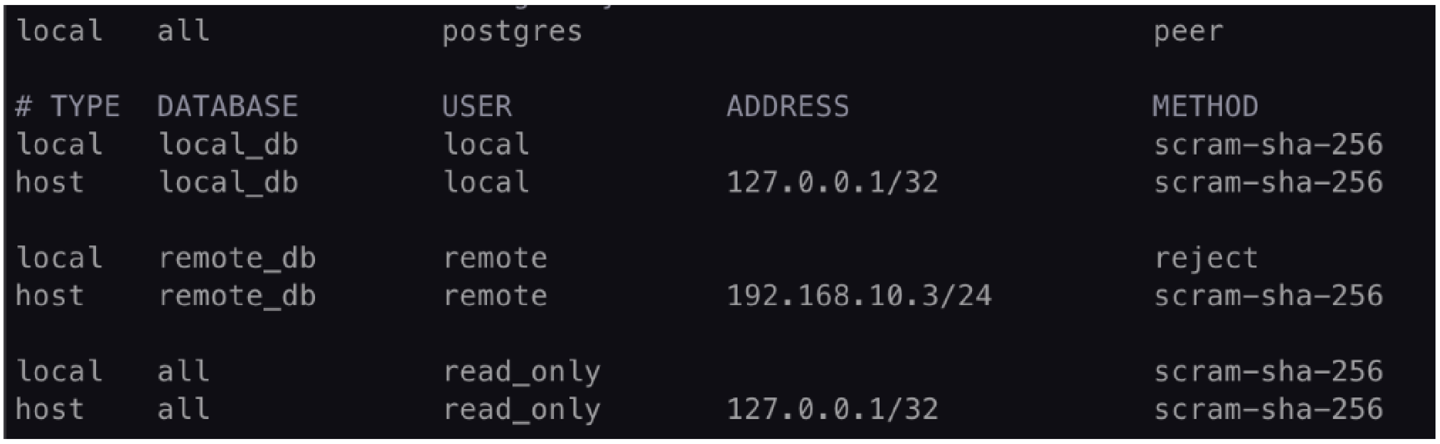
Additionally, both commands `\list` and `\l` have the same output, which is a list of all databases.



```
postgres=# \l
                              List of databases
   Name    |  Owner   | Encoding | Collate |  Ctype  |   Access privileges
-----------+----------+----------+---------+---------+-----------------------
 local_db  | postgres | UTF8     | C.UTF-8 | C.UTF-8 | =Tc/postgres         +
           |          |          |         |         | postgres=CTc/postgres+
           |          |          |         |         | local=CTc/postgres
 postgres  | postgres | UTF8     | C.UTF-8 | C.UTF-8 |
 remote_db | postgres | UTF8     | C.UTF-8 | C.UTF-8 | =Tc/postgres         +
           |          |          |         |         | postgres=CTc/postgres+
           |          |          |         |         | remote=CTc/postgres
 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
(5 rows)
```

## Database Access Configuration

To restrict the access of our new database users, we need to edit the config file `/etc/postgresql/14/main/pg_hba.conf` to look like pictured below. We have commented out the old configuration.



```
local   all             postgres                                peer

# TYPE  DATABASE        USER            ADDRESS                 METHOD
local   local_db        local                                   scram-sha-256
host    local_db        local           127.0.0.1/32            scram-sha-256

local   remote_db       remote                                  reject
host    remote_db       remote          192.168.10.3/24         scram-sha-256

local   all             read_only                               scram-sha-256
host    all             read_only       127.0.0.1/32            scram-sha-256
```

Finally, to enable connections from remote locations, we need to edit the file `/etc/postgresql/14/main/postgresql.conf` and navigate to the "Connections and Authentication".
There, add the IP address of VM1 to the listen address:

```
listen_addresses = 'localhost, 192.168.10.1'
```

Afterwards, restart the PostgreSQL service and adjust the firewall such that incoming connections to port 5432 are allowed by adding

```
# add to input chain
tcp dport 5432 accept

# add to forward chain
tcp sport 5432 accept

# add to output chain
tcp sport 5432 accept
```

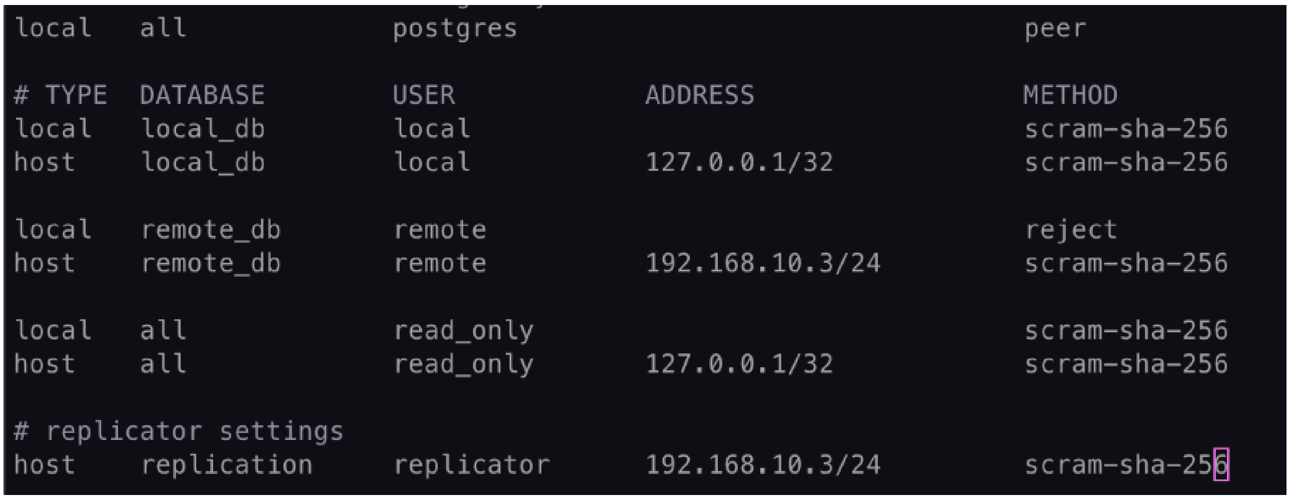
to the nftables.conf file.

## Backup

A good backup is never stored in the same physical location as the original server since a fire could break out in the server room and then the physical storage would be in danger ... Since we don't use physical servers for our course, we can simulate this by replicating the data on a different VM, which is VM3 in our setup.

## Replicator Setup

On our master server VM1, we need to create a new user `CREATE USER replicator WITH REPLICATION PASSWORD 'replica';` and then also edit the file `/etc/postgresql/14/main/pg_hba.conf` again.



```
local   all             postgres                                peer

# TYPE  DATABASE        USER            ADDRESS                 METHOD
local   local_db        local                                   scram-sha-256
host    local_db        local           127.0.0.1/32            scram-sha-256

local   remote_db       remote                                  reject
host    remote_db       remote          192.168.10.3/24         scram-sha-256

local   all             read_only                               scram-sha-256
host    all             read_only       127.0.0.1/32            scram-sha-256

# replicator settings
host    replication     replicator      192.168.10.3/24         scram-sha-256
```

We also need to adjust the write ahead log (WAL) settings. Open the config file at `/etc/postgresql/14/main/postgresql.conf` and set the following:

```
wal_level = logical

# normally commented and set to off
wal_log_hints = on
```

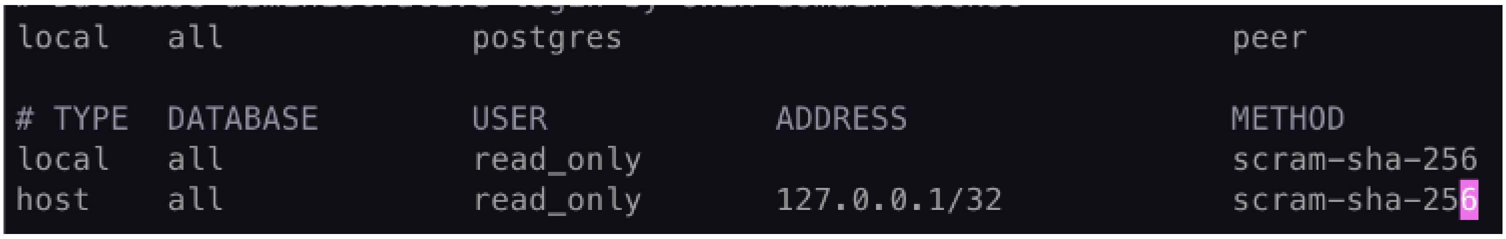
After saving the new edits, restart the postgres service to enable the new settings.

Next, switch to the slave server at VM3.

We need to repeat the first several steps of what we did for setting up PostgreSQL on VM1. Start first with installing PostgreSQL via apt.

Note, we don't need to add VM3's IP address to the listening addresses like we did for VM1, since we don't want users to actually connect to the database as it only serves as a backup of the actual databases at VM1.

This is what `/etc/postgresql/14/main/pg_hba.conf` looks like on the slave server.



```
local   all             postgres                                peer

# TYPE  DATABASE        USER            ADDRESS                 METHOD
local   all             read_only                               scram-sha-256
host    all             read_only       127.0.0.1/32            scram-sha-256
```

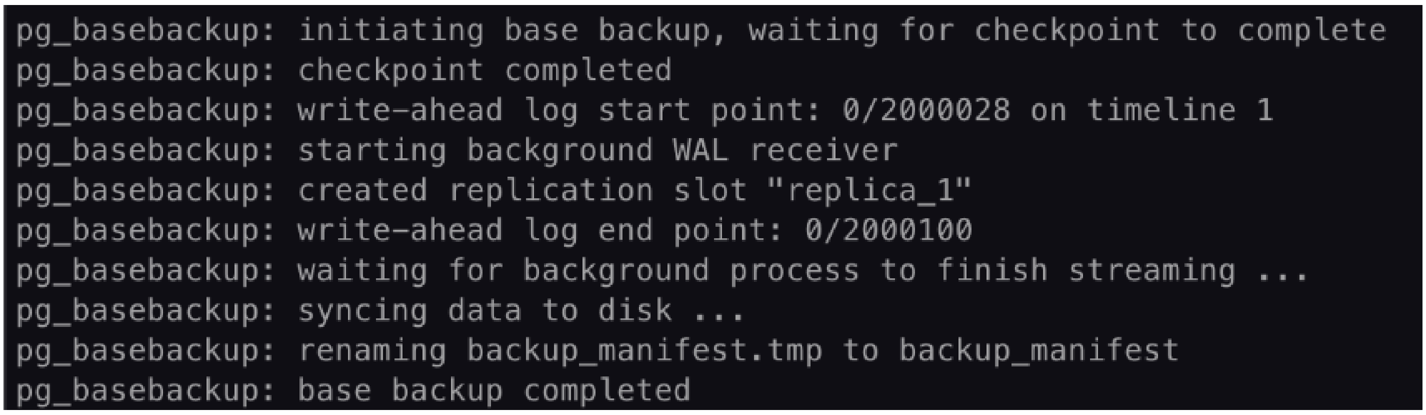
After making sure it works as intended and is enabled, turn the postgres service off again. Since we need to make sure we have a clean state on our slave server, run `sudo rm -rv /var/lib/postgresql/14/main/`.
Then, we can use `pg_basebackup`, which is a utility tool used to take a base backup of a running PostgreSQL database cluster. [Here]() is a link to its documentation for more information.

```
sudo pg_basebackup -h 192.168.10.1 -U replicator -X stream -C -S replica_1 -v
-R -W -D /var/lib/postgresql/14/main/
```

Explanation of the options we used above:

- -h denotes the IP address of the server we want to replicate
- -U denotes the user that connects to the server
- -X clarifies that the WAL data is streamed while the backup is being taken
- -C specifies the that the replication slot named by the -S option should be created before starting the backup
- -S names the replication slot
- -v for verbose output
- -R creates a standby.signal file and appends connection settings to the postgresql.auto.conf file in the target directory, which eases setting up a standby server using the results of the backup.
- -W prompts password input
- -D specifies where the data is being saved

This is what the output should look like:

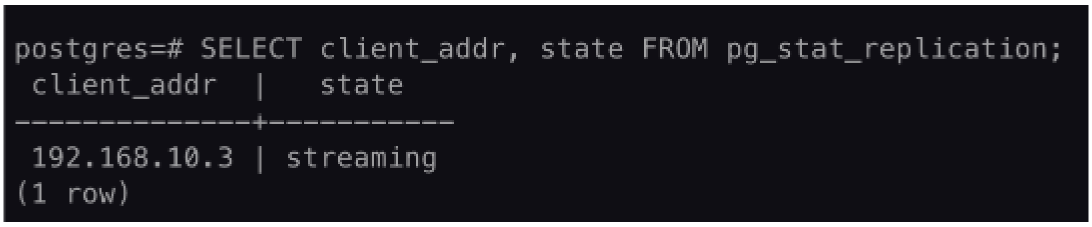


```
pg_basebackup: initiating base backup, waiting for checkpoint to complete
pg_basebackup: checkpoint completed
pg_basebackup: write-ahead log start point: 0/2000028 on timeline 1
pg_basebackup: starting background WAL receiver
pg_basebackup: created replication slot "replica_1"
pg_basebackup: write-ahead log end point: 0/2000100
pg_basebackup: waiting for background process to finish streaming ...
pg_basebackup: syncing data to disk ...
pg_basebackup: renaming backup_manifest.tmp to backup_manifest
pg_basebackup: base backup completed
```

Finally, change ownership of the backup location to the postgres user and restart the PostgreSQL service.

```
sudo chown postgres -R /var/lib/postgresql/14/main/
sudo systemctl restart postgresql
```

You can test if this setup was successful by logging into the master server and querying the pg_stat_replication table with `SELECT client_addr, state FROM pg_stat_replication;`.



```
postgres=# SELECT client_addr, state FROM pg_stat_replication;
 client_addr  |   state
--------------+-----------
 192.168.10.3 | streaming
(1 row)
```

## Backup Script

We want to create a script that makes a backup on the master server.

Since there is no default directory for backups, create a new one:

```
sudo mkdir -p /var/log/postgres/psql_backups/
sudo chown -R root:root /var/log/postgres/psql_backups/
sudo chmod 0770 /var/log/postgres/psql_backups/
```

The following script is saved as `/usr/local/sbin/psql_backup.sh`.

```
#!/bin/bash


# Backup destination
backup_dir="/var/log/postgresql/psql_backups"
log_dir="/var/log/postgresql/psql_logs"


# Check if directories exists
if [ ! -d "$backup_dir" ]; then
    mkdir -p "$backup_dir"
else
    echo "Backup directory exists."
fi

# Check if directory exists
if [ ! -d "$log_dir" ]; then
    mkdir -p "$log_dir"
else
    echo "Log directory exists."
fi

# Get today's date
date=$(date +%Y%m%d_%H%M%S)

# Create std and err log files
```

```
log_std="${log_dir}/backup_${date}_std.log"
log_err="${log_dir}/backup_${date}_err.log"

# Redirect to log files
exec 1>> ${log_std}
exec 2>> ${log_err}

# Create backup file name with the date
backup_file="${backup_dir}/backup_${date}.sql"

# Create backup on master server
PGPASSWORD="read" pg_dumpall -p 5432 -U read_only -f ${backup_file}

if [ $? -eq 0 ]; then
    echo "Backup completed successfully."
else
    echo "Backup failed!"
fi
```

According to documentation, the preferred way of passing a password is creating a .pgpass file in the home directory of the user that runs the script. Since the user running the script is not the same user as "read_only", we will just set the environment variable PGPASSWORD to the correct password.

## Crontab Setup

Install cron first.

```
sudo apt-get install cron
```

Afterwards, enter the crontab editor:

```
crontab -e
```

And enter the following into the editor to do a daily backup at 5AM: `00 05 * * * /usr/local/sbin/psql_backup.sh`

## Testing

The testing script is located at `/home/patricia/test_PSA_05.sh` on VM1.

```
#!/bin/bash

# PostgreSQL database connection details
DB_HOST="127.0.0.1"
DB_PORT="5432"
DB_USER="read_only"
DB_PW="read"


pg_isready -h "$DB_HOST" -p "$DB_PORT"

# Check the exit status of pg_isready
if [ $? -eq 0 ]; then
    echo "PostgreSQL is active and accepting connections."
else
    echo "PostgreSQL is not active or not accepting connections."
fi


# create .gpass file for script automation to skip user input required for the
password
PGPASS_FILE=~/.pgpass
CHECK_USER="local"
DB_NAME="local_db"
echo "$DB_HOST:$DB_PORT:$DB_NAME:$DB_USER:$DB_PW" > "$PGPASS_FILE"
chmod 600 "$PGPASS_FILE"

SQL_CHECK_USER="SELECT 1 FROM pg_user WHERE usename = '$CHECK_USER';"

echo "Check if local user and local database exist."
result=$(psql -h "$DB_HOST" -p "$DB_PORT" -d "$DB_NAME" -U "$DB_USER" -tAc
"select 1 from pg_user where usename='$CHECK_USER';" 2>&1)

# Check the result of the query
if [ "$result" = "1" ]; then
    echo "User $CHECK_USER and database $DB_NAME exist in PostgreSQL."
else
    echo "User $CHECK_USER does not exist in PostgreSQL or an error
occurred: $result"
fi
rm "$PGPASS_FILE"

echo "Check if remote user and remote database exist by connecting to it and
checking if the database contains entries."
DB_NAME="remote_db"

# create .gpass file for script automation to skip user input required for the
password
echo "$DB_HOST:$DB_PORT:$DB_NAME:$DB_USER:$DB_PW" > "$PGPASS_FILE"
chmod 600 "$PGPASS_FILE"

psql -h "$DB_HOST" -p "$DB_PORT" -U "$DB_USER" -d "$DB_NAME" -c "\dt" >
/dev/null 2>&1
```

```
# Check the exit status of the psql command
if [ $? -eq 0 ]; then
    echo "Connection successful. The database contains tables, thus the remote
user and remote database fulfill the requirements for correct functionality."
else
    echo "Connection failed or the database does not contain any tables."
fi

rm "$PGPASS_FILE"
```