COMBINING HOMEGROWN AND STREAMING REPLICATION IN POSTGRESQL FOR CREATING DISTRIBUTED DATABASE NETWORK USING AWS AURORA

Peeyush Kumar
Boston University
peeyush@bu.edu

Abstract

In this term paper, an instance of distributed database network (via using Docker) has been engineered along with streaming PostgreSQL replication. In today's world, when a business grows rapidly over the globe, having one single master database will not be a very efficient and smart approach. To handle large businesses the concept of Distributed Database Management System (DDBMS) is utilized. In this paper, a small dvd Netflix business has been scaled to a country wide level having 6 physical server (with partitioned RDBMS) and one AWS Aurora Cloud PostgreSQL Server. The network has been designed in a manner that, along with each server having its own personal streaming replication, the data flows over the network in such a way that until all 7 servers fail, the business data will always offer Availability. Along with availability, consistency and data integrity has also been taken into account.

1. Introduction

Now our Netflix dvd rental business (Its a movie relational database that we have been working for this whole semester in Advance Database Management System course) has grown worldwide and have offices and wending machines in almost every country. Therefore, having one centralized database will not be very efficient in terms of performance and security. With one single database failure, the whole business of the company will shutdown and which will be conducive to decline in customer base. Now the question is what can we do to prevent this doomsday, the only solution is to implement the concept of *Distributed Database Management System (DDBMS)*. Since our business is in different countries we will split the database Horizontally according to the country wise attribute and ERD will also be split Vertically according to important attributes for Redbox machines (These are the Machines Which Dispense DVDs). For example, all the users of America will have their information stored in a database server that will be present in America and each Redbox machine will have its own fragmented Database. In this Project, a small instance of Distributed Database Network has been implemented. So the Problem statement for this project is :-

Problem Statement: *The Netflix Company has deployed various Redbox machines across America. So, a Distributed Database Approach is required to meet the company's needs for performance and security requirements. Also, an appropriate combination of Synchronous and Asynchronous Replication is required to ensure that Availability and Consistency is always present.*

In order to simulate the distributed network Docker and AWS Aurora PostgreSQL has been used along with PostgreSQL Original Server. Total 6 Docker Servers and 1 Cloud server has been used to implement the Distributed Database. The outline for the contents for the paper is as follows: First, we will go over the few important concepts in Distributed Database and PostgreSQL Replication. Secondly, some key technologies used and how they are used to simulate the Distributed DBMS will be explained along with the Proposed Distributed Network Design. Finally, an example for Distributed Transaction performed using Stored Functions will be explained.

2. Important Concepts

1. Replication In PostgreSQL

- The Replication is a process to make a copy of the existing database onto a different computer or server or database. The main reason we want replication is to ensure availability of database all the time and also to make sure that out company never goes offline in an event to server failure. There are various Replication types in PostgreSQL :

PRIMARY
REPLICA

Figure 1: The Generic Replication in PostgreSQL

a) Trigger Based Replication

- This Kind of replication is done using stored triggers. An event happens and the trigger replicates the data to the set location. The amount and type of data to replicate is subjective.

b) **Binary Replication**

- This kind of replication is done at binary level using WAL files. Here instead of replicating the data values we only replicate the operation performed on them.

c) **Asynchronous Replication**

- Asynchronous replication is a store and forward approach to data backup or data protection. Asynchronous replication writes data to the primary storage array first and then, depending on the implementation approach, commits data to be replicated to memory or a disk-based journal.

d) **Synchronous Replication (Streaming Replication)**

- It allows the updated information on the primary server to be transferred to the standby server in real time, so that the databases of the primary server and standby server can be kept in sync, hence Synchronous Replication

e) **Logical Replication**

- Logical replication is a method of replicating data objects and their changes, based upon their replication identity.

f) **Homegrown Replication**

- This Kind of replication is done using stored function which takes data to the target server or database. It's a custom replication process.

2. Write Ahead Logging (WAL) File

- WAL files are kind of a binary Change Logs. The changes in a database are first recorded in the log, which must be written to stable storage. When we reach the certain

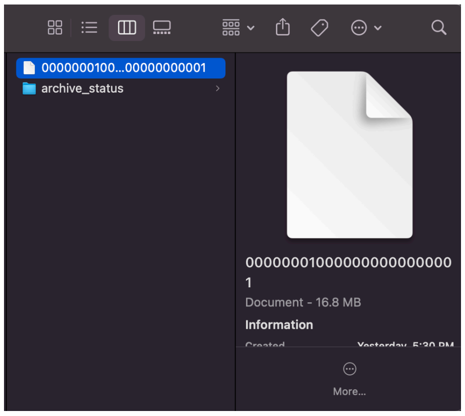
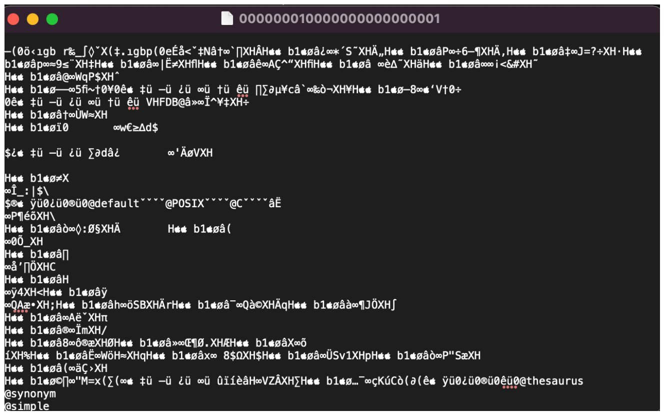


Figure 2: WAL FILE in MAC



```
–(0ö‹ıgb r‰_∫◊ˇX(‡.ıgbp(0eÉå<ˇ‡Nâ†∞`∏XHÂH b1øâ¿∞∗´S˜XHÄ„H b1øâP∞÷6–¶XHÄ,H b1øâ‡∞J=?÷XH·H
b1øâp∞≈9≤¨XH‡H b1øâ∞|Ë≠XHflH b1øâê∞AÇ^"XHfiH b1øâ ∞èΔ˜XHäH b1øâ∞∞i<&#XH˜
H b1øâ@∞WqP$XHˆ
H b1ø——∞5fi~†0¥0ê ‡ü –ü ¿ü ∞ü †ü êü ∏∑∂µ¥câ`∞‰ò¬XH¥H b1ø–8∞'V†0÷
0ê ‡ü –ü ¿ü ∞ü †ü êü VHFDB@â»∞Ï^¥‡XH÷
H b1øâ†∞ÙW≈XH
H b1øï0       ∞w€≥Δd$

$¿ ‡ü –ü ¿ü ∑∂dâ¿        ∞'ÄøVXH

H b1ø≠X
∞Î_:|$\
$® ÿü0¿ü0®ü0@defaultˇˇˇˇ@POSIXˇˇˇˇ@CˇˇˇˇâË
∞P¶éõXH\
H b1øâò∞◊:Ø§XHÄ        H b1øâ(
∞0Õ_XH
H b1øâ∏
∞å'∏ÖXHC
H b1øâH
∞ÿ4XH<H b1øâÿ
∞QAæ•XH;H b1øâh∞öSBXHÄrH b1øâ¯∞Qà©XHÄqH b1øââ∞¶JÖXH∫
H b1øâ∞Aëˇ XHπ
H b1øâ®∞ÏmXH/
H b1øâ8∞ô®æXHØH b1øâ»∞Œ¶Ø.XHÆH b1øâX∞õ
íXH%H b1øâË∞WöH≈XHqH b1øâx∞ 8$ΩXH$H b1øâ∞ÜSv1XHpH b1øâò∞P"SæXH
H b1øâ(∞äÇ›XH
H b1ø©∏∞"M=x(∑(∞ ‡ü –ü ¿ü ∞ü ûïíèâH∞VZÂXH∑H b1ø…¯∞çKúCò(∂(ê ÿü0¿ü0®ü0êü0@thesaurus
@synonym
@simple
```

Figure 3: Encrypted (using scram-sha-256 or md5) Content of WAL FILE

checkpoint (usually is ~16MB), then the changes are written to the database. Figure 2 and 3, shows what a log file looks like in MAC OS.

3. Distributed Systems

- A distributed database is a database in which data is stored across different physical locations. It may be stored in multiple computers located in the same physical location; or maybe dispersed over a network of interconnected computers. The distributed systems ensure that we have availability most of the time and also benefits performance wise as the data is always near to the point where it is most needed.

4. Docker Virtual Ports and How to setup?

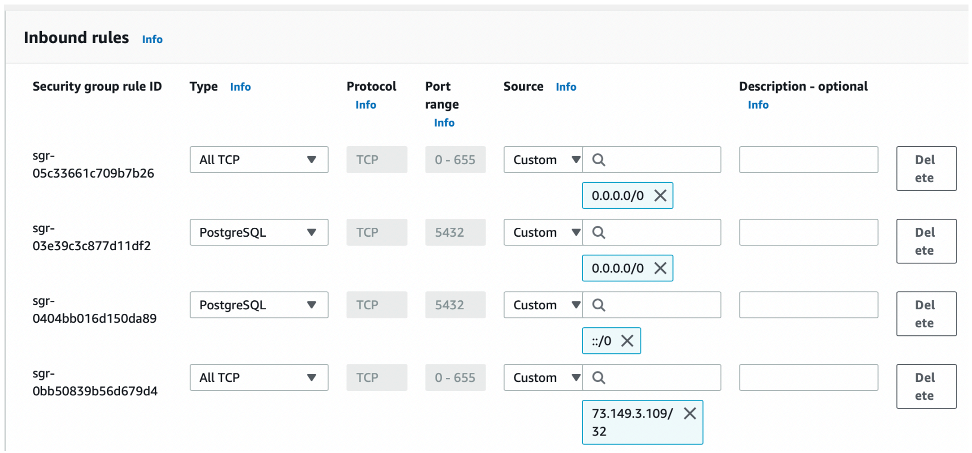
- Docker is a set of platform as a service products that use OS-level virtualization to deliver software in packages called containers. Docker lets use create virtual ports in the same computer. Let's say we wan to create a server "West Coast" with port number 5430. Inorder to do so, use this in Root Terminal:

```
docker run -p 5430:5432 --name West Coast -e POSTGRES_PASSWORD=mynewpassword postgres
```

- What this line does? : It's creates a virtual docker server (named "West Coast" and password "mynewpassword") with port number 5430 and maps it to the default Postgres port 5432. "postgres" is the Super User in this server.

5. AWS Aurora PostgreSQL and Specifications

- For cloud server, PostgreSQL instance of AWS Aurora is being used, with custom port number : 5440. The cloud server is being connect through docker using the security key and the important elements for ensuring this connectivity is as follows :



Inbound rules Info

Security group rule ID	Type Info	Protocol Info	Port range Info	Source Info	Description - optional Info	
sgr-05c33661c709b7b26	All TCP	TCP	0 - 655	Custom 0.0.0.0/0		Delete
sgr-03e39c3c877d11df2	PostgreSQL	TCP	5432	Custom 0.0.0.0/0		Delete
sgr-0404bb016d150da89	PostgreSQL	TCP	5432	Custom ::/0		Delete
sgr-0bb50839b56d679d4	All TCP	TCP	0 - 655	Custom 73.149.3.109/32		Delete

Figure 4 : Showing the custom Inbound rules created for VPC check in. This lets the servers to connect to AWS Cloud.

A. **IAM Authentication**: Stands for Identity and Access Management (IAM). It occurs whenever a user attempts to access your Database. It was enabled, so that two different PostgreSQL ports can interact smoothly.

B. **VPC Security**: VPC or Virtual Private Cloud in AWS has to be configured to accept connections from the outer sources. It's very important for security in Amazon Web Services. A custom VPC inbound rules were created for connecting with virtual docker ports. Figure Shows the custom inbound rules that were setup to make the connection with docker possible.

3. Performing Streaming Replication in PostgreSQL

Performing Streaming or Synchronous replication in PostgreSQL is a complex task in it self and requires a strong command line knowledge. The three important postgreSQL files to keep in mind are :-

A. **pg_hba.conf** : It is an access policy configuration file. It keeps track of incoming accesses and encryptions to apply.

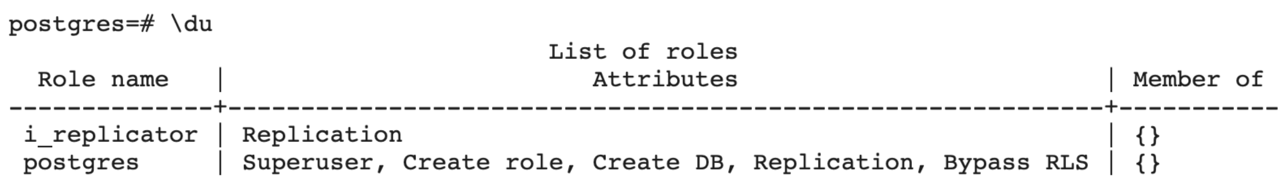
B. **postgresql.conf** : It is a main configuration file and the primary source of configuration parameter settings.

C. **recovery.conf** : This is the file we create. It contained the where about of MASTER node and when to fetch the data.

It is important thing to remember is that we have to edit these files and their directories forcefully, specially in Mac Operating systems. Lets see how to set up the Streaming replication in postgreSQL. Lets assume that we want to set up replication for "West_Coast_Red_Box_2" with port 5426 :-

1. First, create a User that will replicate the data, let's say its name is "*i_replicator*", create this user using the following command (this user will have authority to replicate only, you could also use super user for this task but it is not advised) :

```
CREATE USER i_replicator REPLICATION LOGIN ENCRYPTED PASSWORD
'mynewpassword';
```

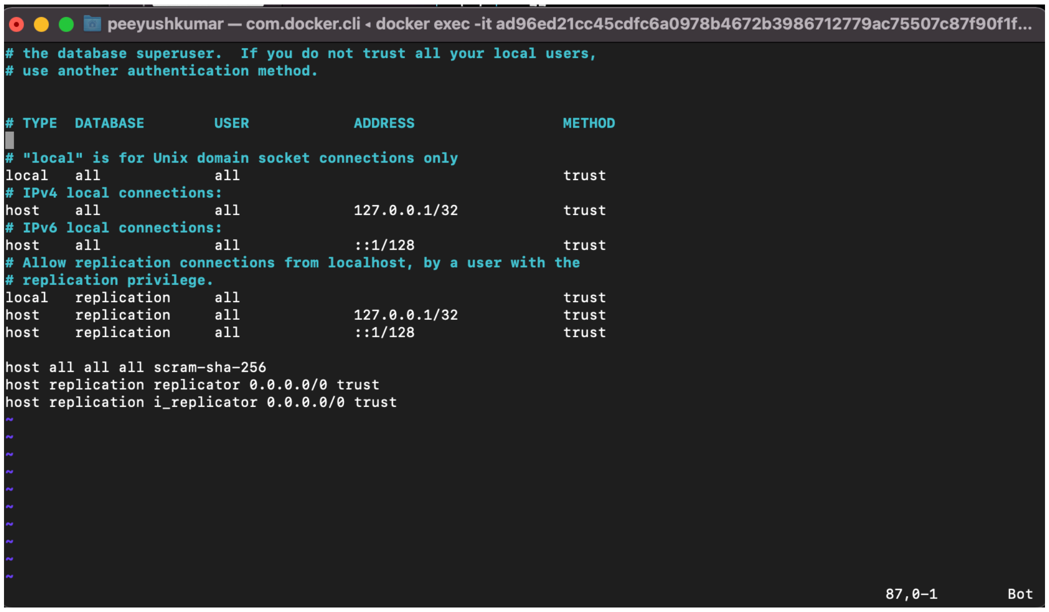


```
postgres=# \du
                                   List of roles
  Role name    |                         Attributes                         | Member of
---------------+------------------------------------------------------------+-----------
 i_replicator  | Replication                           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

Figure 5: Showing the 'i_replicator' user made in PostgreSQL, which have rights to replicate from primary node

2. Open the CLI for Port 5426 and runs this command (do all the steps in CLI of Port 5426). Adding this line to your ph_hba.conf will make sure that the postgreSQL server accepts connection from the user *i_replicator* :

echo 'host replication i_replicator 0.0.0.0/0 trust' >> /var/lib/postgresql/data/pg_hba.conf



```
peeyushkumar — com.docker.cli ‹ docker exec -it ad96ed21cc45cdfc6a0978b4672b3986712779ac75507c87f90f1f...
# the database superuser.  If you do not trust all your local users,
# use another authentication method.


# TYPE  DATABASE        USER            ADDRESS                 METHOD

# "local" is for Unix domain socket connections only
local   all             all                                     trust
# IPv4 local connections:
host    all             all             127.0.0.1/32            trust
# IPv6 local connections:
host    all             all             ::1/128                 trust
# Allow replication connections from localhost, by a user with the
# replication privilege.
local   replication     all                                     trust
host    replication     all             127.0.0.1/32            trust
host    replication     all             ::1/128                 trust

host all all all scram-sha-256
host replication replicator 0.0.0.0/0 trust
host replication i_replicator 0.0.0.0/0 trust
~
~
~
~
~
~
~
~
~
~
                                                    87,0-1        Bot
```

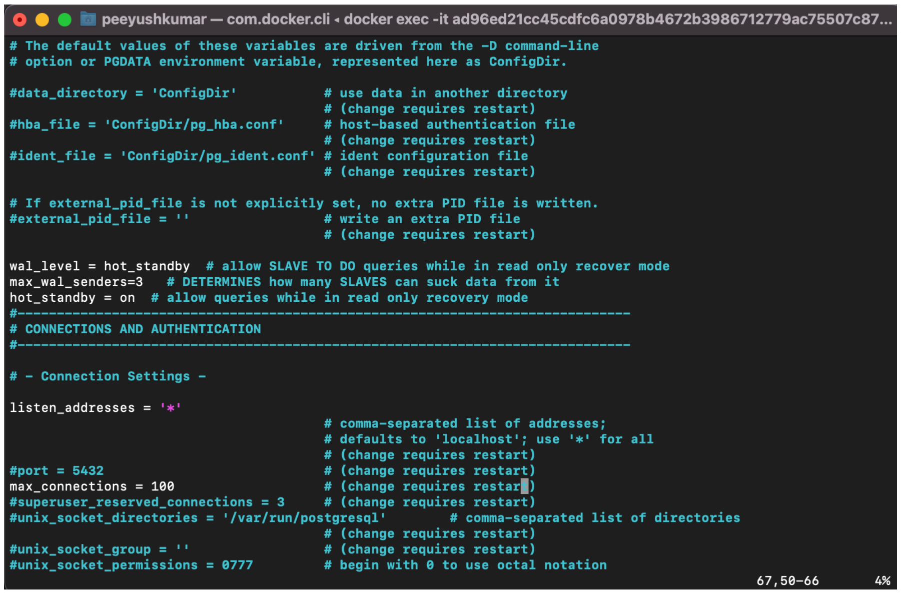
Figure 6 : In this it can be seen that the ip address of the servers has been added in the ph_hba.conf file of PostgreSQL so that the Replica node can connect to it for replication.

3. Insert the following line in your postgresql.conf file (add them above CONNECTIONS AND AUTHENTICATION) using **vim** command restart your sever or docker port (add the following line to the conf file using *vim* command and save the edit using :*wq!*):

wal_level = hot_standby # allow SLAVE TO DO queries while in read only recover mode

max_wal_senders=3 # DETERMINES how many SLAVES can suck data from it

hot_standby = on # allow queries while in read only recovery mode

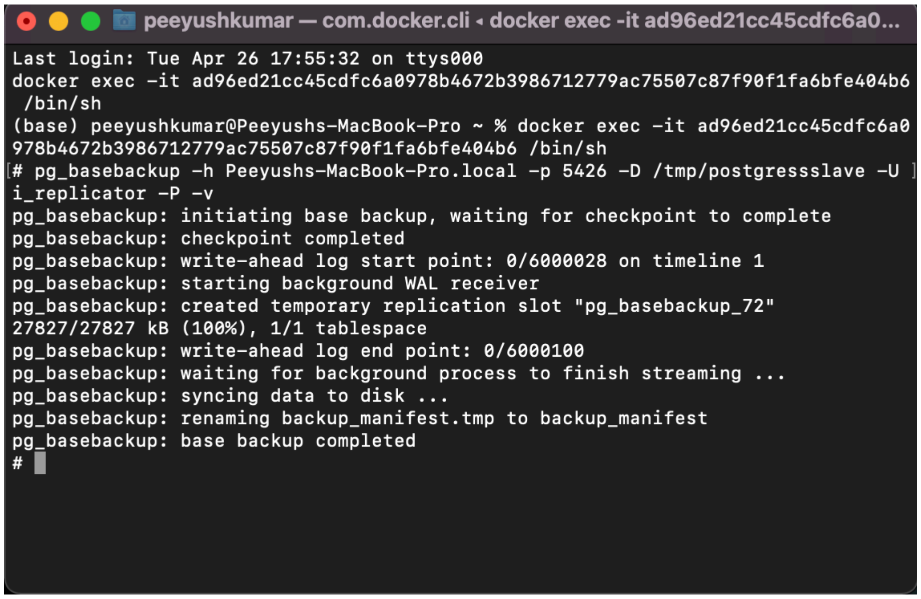


```
peeyushkumar — com.docker.cli ◂ docker exec -it ad96ed21cc45cdfc6a0978b4672b3986712779ac75507c87...
# The default values of these variables are driven from the -D command-line
# option or PGDATA environment variable, represented here as ConfigDir.

#data_directory = 'ConfigDir'           # use data in another directory
                                        # (change requires restart)
#hba_file = 'ConfigDir/pg_hba.conf'     # host-based authentication file
                                        # (change requires restart)
#ident_file = 'ConfigDir/pg_ident.conf' # ident configuration file
                                        # (change requires restart)

# If external_pid_file is not explicitly set, no extra PID file is written.
#external_pid_file = ''                 # write an extra PID file
                                        # (change requires restart)

wal_level = hot_standby  # allow SLAVE TO DO queries while in read only recover mode
max_wal_senders=3   # DETERMINES how many SLAVES can suck data from it
hot_standby = on  # allow queries while in read only recovery mode
#------------------------------------------------------------------------------
# CONNECTIONS AND AUTHENTICATION
#------------------------------------------------------------------------------

# - Connection Settings -

listen_addresses = '*'
                                        # 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)
#superuser_reserved_connections = 3     # (change requires restart)
#unix_socket_directories = '/var/run/postgresql'        # comma-separated list of directories
                                        # (change requires restart)
#unix_socket_group = ''                 # (change requires restart)
#unix_socket_permissions = 0777         # begin with 0 to use octal notation
                                                                    67,50-66      4%
```

Figure 7 : In this it can be seen that the mentioned lines have been added to the postgresql.conf file of PostgreSQL in order to enable replication and also so that the Replica node can connect to it.

4. Now setup up a backup file using a pg_basebackup (this makes the exact BINARY copy of all of the database cluster in that PORT). This command will make a dir name "postgressslave" (this is the directory where the replica server will reside) and will backup in this directory (*Peeyushs-MacBook-Pro.local is a host machine name*):

pg_basebackup -h Peeyushs-MacBook-Pro.local -p 5426 -D /tmp/postgressslave -U i_replicator -P -v



```
peeyushkumar — com.docker.cli ◂ docker exec -it ad96ed21cc45cdfc6a0...
Last login: Tue Apr 26 17:55:32 on ttys000
docker exec -it ad96ed21cc45cdfc6a0978b4672b3986712779ac75507c87f90f1fa6bfe404b6
 /bin/sh
(base) peeyushkumar@Peeyushs-MacBook-Pro ~ % docker exec -it ad96ed21cc45cdfc6a0
978b4672b3986712779ac75507c87f90f1fa6bfe404b6 /bin/sh
[# pg_basebackup -h Peeyushs-MacBook-Pro.local -p 5426 -D /tmp/postgressslave -U ]
i_replicator -P -v
pg_basebackup: initiating base backup, waiting for checkpoint to complete
pg_basebackup: checkpoint completed
pg_basebackup: write-ahead log start point: 0/6000028 on timeline 1
pg_basebackup: starting background WAL receiver
pg_basebackup: created temporary replication slot "pg_basebackup_72"
27827/27827 kB (100%), 1/1 tablespace
pg_basebackup: write-ahead log end point: 0/6000100
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
#
```

Figure 8 : Figure showing that the database has been backed-up in the Replica directory so that when we start the Replica server, both Primary and Replica are at same level or state.

5. Now create a recovery.conf file using :

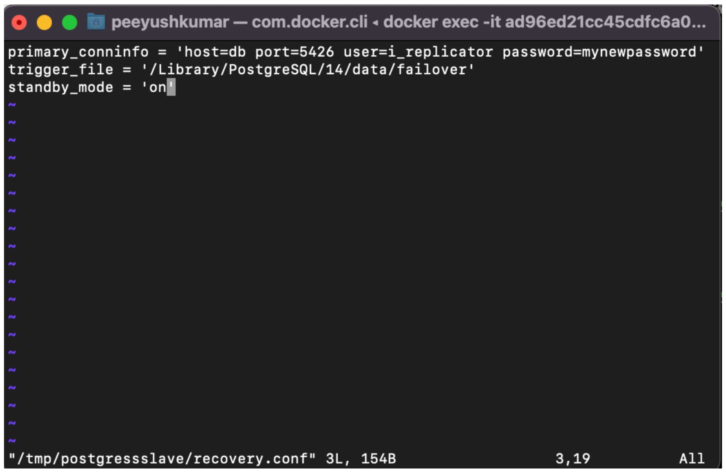
touch /tmp/postgressslave/recovery.conf

6. Add the following lines to the *recovery.conf* file :

primary_conninfo = 'host=db port=5426 user=i_replicator password=mynewpassword'

trigger_file = '/Library/PostgreSQL/14/data/failover'

standby_mode = 'on'



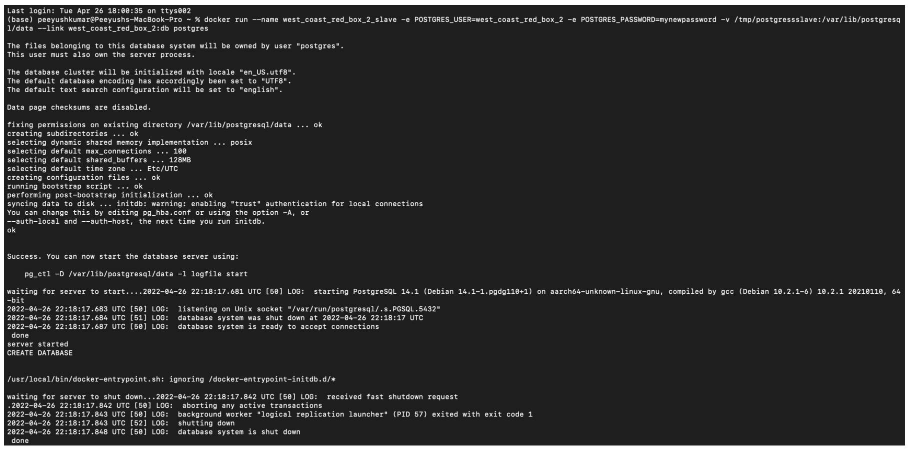
peeyushkumar — com.docker.cli ◂ docker exec -it ad96ed21cc45cdfc6a0...

```
primary_conninfo = 'host=db port=5426 user=i_replicator password=mynewpassword'
trigger_file = '/Library/PostgreSQL/14/data/failover'
standby_mode = 'on'
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
"/tmp/postgressslave/recovery.conf" 3L, 154B                    3,19          All
```

Figure 9 : Adding the lines in newly created recovery configuration file for Replica server.

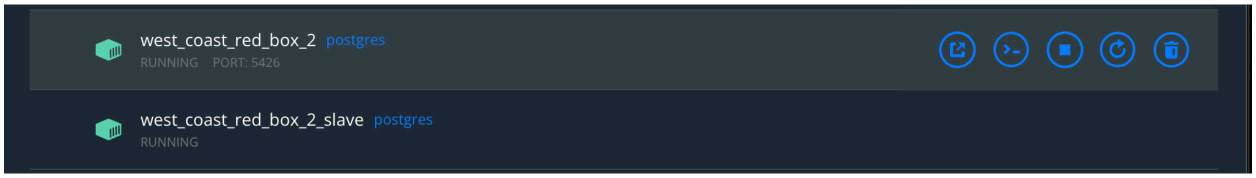
7. Now run this command on root terminal to start replica server:

docker run --name west_coast_red_box_2_slave -e POSTGRES_USER=west_coast_red_box_2 -e POSTGRES_PASSWORD=mynewpassword -v /tmp/postgressslave:/var/lib/postgresql/data --link west_coast_red_box_2:db postgres



```
Last login: Tue Apr 26 18:00:35 on ttys002
(base) peeyushkumar@Peeyushs-MacBook-Pro ~ % docker run --name west_coast_red_box_2_slave -e POSTGRES_USER=west_coast_red_box_2 -e POSTGRES_PASSWORD=mynewpassword -v /tmp/postgressslave:/var/lib/postgresq
l/data --link west_coast_red_box_2:db postgres

The files belonging to this database system will be owned by user "postgres".
This user must also own the server process.

The database cluster will be initialized with locale "en_US.utf8".
The default database encoding has accordingly been set to "UTF8".
The default text search configuration will be set to "english".

Data page checksums are disabled.

fixing permissions on existing directory /var/lib/postgresql/data ... ok
creating subdirectories ... ok
selecting dynamic shared memory implementation ... posix
selecting default max_connections ... 100
selecting default shared_buffers ... 128MB
selecting default time zone ... Etc/UTC
creating configuration files ... ok
running bootstrap script ... ok
performing post-bootstrap initialization ... ok
syncing data to disk ... initdb: warning: enabling "trust" authentication for local connections
You can change this by editing pg_hba.conf or using the option -A, or
--auth-local and --auth-host, the next time you run initdb.
ok


Success. You can now start the database server using:

    pg_ctl -D /var/lib/postgresql/data -l logfile start

waiting for server to start....2022-04-26 22:18:17.681 UTC [50] LOG:  starting PostgreSQL 14.1 (Debian 14.1-1.pgdg110+1) on aarch64-unknown-linux-gnu, compiled by gcc (Debian 10.2.1-6) 10.2.1 20210110, 64
-bit
2022-04-26 22:18:17.683 UTC [50] LOG:  listening on Unix socket "/var/run/postgresql/.s.PGSQL.5432"
2022-04-26 22:18:17.684 UTC [51] LOG:  database system was shut down at 2022-04-26 22:18:17 UTC
2022-04-26 22:18:17.687 UTC [50] LOG:  database system is ready to accept connections
 done
server started
CREATE DATABASE


/usr/local/bin/docker-entrypoint.sh: ignoring /docker-entrypoint-initdb.d/*

waiting for server to shut down...2022-04-26 22:18:17.842 UTC [50] LOG:  received fast shutdown request
.2022-04-26 22:18:17.842 UTC [50] LOG:  aborting any active transactions
2022-04-26 22:18:17.843 UTC [50] LOG:  background worker "logical replication launcher" (PID 57) exited with exit code 1
2022-04-26 22:18:17.843 UTC [52] LOG:  shutting down
2022-04-26 22:18:17.848 UTC [50] LOG:  database system is shut down
 done
```

Figure 10 : Show that the Replica serve has been created and is now ready to replicate and take over the Primary in time of failure


west_coast_red_box_2 postgres
RUNNING PORT: 5426

west_coast_red_box_2_slave postgres
RUNNING

Figure 11 : Showing the Replica Server which was created for the Primary server "west_coast_red_box_2"

4. Proposed Distributed Network Design

In the Proposed Distributed Design, I scaled the Netflix DVD Rental Business across the country. While scaling I have assumed that the business also have some DVD dispensing Vending Machines. A small instance of the distributed network has been successfully coded. In the mentioned instance, we have 6 Physical servers and 1 Cloud server. In order to create

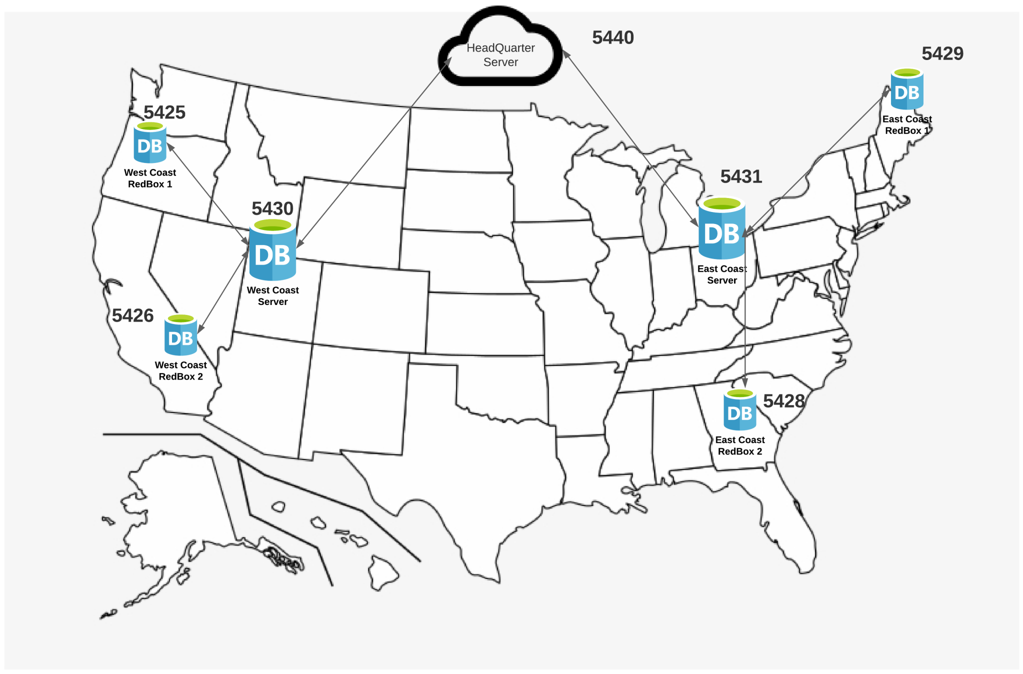


Figure 12 : Showing the the proposed Distributed Network Design for United States Business. It also shows the different port numbers for different servers.

different servers DOCKER is used!! For the Cloud Server, AWS Aurora PostgreSQL has been used. Figure 12 , show the high level design for the distributed network.

5. Setting Up the Proposed Distributed DBMS

In order to link Different Servers in PostgreSQL, DBLINK extension is used. In DBLINK we create a foreign server and its mappings. Figure 14 shows the network links for the whole distributed network. Let's say we have two postgresql servers "West_Coast" with port 5430 and "West_Coast_Red_Box_1" with port 5425, and we are to connect "West_Coast_Red_Box_1" to "West_Coast" so that It can fetch data from it. n order, to achieve this task we do the following in the psql CLI of "West_Coast_Red_Box_1" :

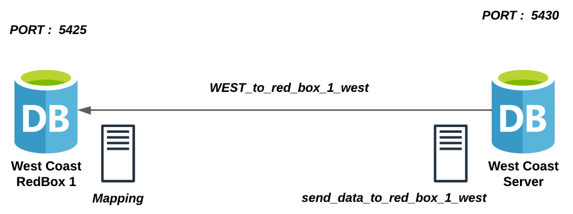


Figure 13 : Showing the end result for the mentioned steps for linking two servers in PostgreSQL.

1. We install DBLINK extension in it

CREATE EXTENSION dblink;

2. We create a foreign server for the WEST COAST SERVER named 'send_data_to_red_box_1_west' and this server will fetch data to itself from the WEST COAST SERVER in case of requirement.

CREATE CREATE SERVER send_data_to_red_box_1_west FOREIGN DATA WRAPPER dblink_fdw OPTIONS (host 'Peeyushs-MacBook-Pro.local' dbname 'postgres' port '5430');

3. We create a mapping for the send_data_to_red_box_1_west SERVER in the west_coast_red_box_1 so that they can share data sucessfullly

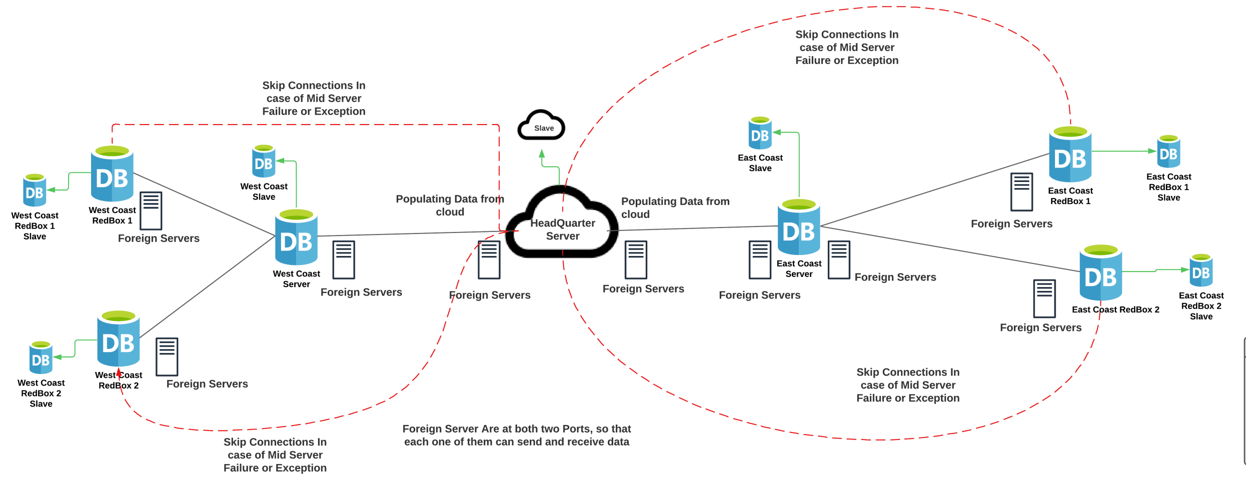
CREATE USER MAPPING FOR postgres SERVER send_data_to_red_box_1_west OPTIONS (user 'postgres', password 'mynewpassword');

4. Then grant access of FOREIGN SERVER send_data_to_red_box_1_west TO the west_coast_red_box_1

GRANT USAGE ON FOREIGN SERVER send_data_to_red_box_1_west to postgres;

5. FINALLY connect the two servers. Whenever we need to activate the connect we call this line of code.

```
SELECT dblink_connect(‘WEST_to_red_box_1_west' 'send_data_to_red_box_1_west');
```



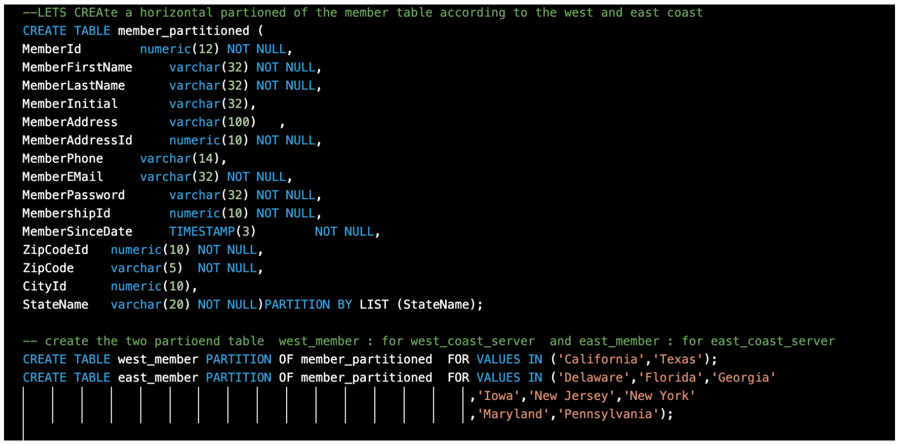
Skip Connections In case of Mid Server Failure or Exception
Skip Connections In case of Mid Server Failure or Exception
Slave
West Coast RedBox 1 Slave
West Coast RedBox 1
Foreign Servers
West Coast Slave
West Coast Server
Foreign Servers
Populating Data from cloud
HeadQuarter Server
Foreign Servers
Populating Data from cloud
Foreign Servers
East Coast Slave
East Coast Server
Foreign Servers
Foreign Servers
East Coast RedBox 1
East Coast RedBox 1 Slave
Foreign Servers
East Coast RedBox 2
East Coast RedBox 2 Slave
Foreign Servers
West Coast RedBox 2 Slave
West Coast RedBox 2
Foreign Servers
Skip Connections In case of Mid Server Failure or Exception
Skip Connections In case of Mid Server Failure or Exception
Foreign Server Are at both two Ports, so that each one of them can send and receive data

Figure 14 : Showing the network diagram for the whole distributed network linked through DBLINK extension. The red connections are activated when in time of failure or some exception. The green links are for data replication for Primary and Replica Node. The black links are always activated for hierarchical replication and also for distributed transactions.

6. Populating Data on Distributed Network using Data Partition

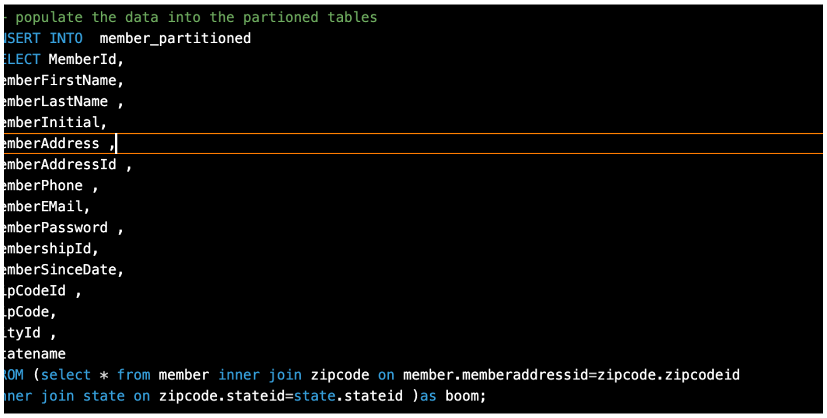
The original Netflix schema was distributed over the distributed network via using Table Partitioning. The schema for a Red_Box, Coast Wise Server and the Headquarter were different. Note: the Stored function were used to Populate the database (Code attached with the paper).

- **For Red Box** : The Vertical Partitioning was done.
- **For Coast Wise Server** : The Horizontal Partitioning was on the basis of ‘*State_name*’ attribute value.



```
--LETS CREAte a horizontal partioned of the member table according to the west and east coast
CREATE TABLE member_partitioned (
MemberId        numeric(12) NOT NULL,
MemberFirstName     varchar(32) NOT NULL,
MemberLastName      varchar(32) NOT NULL,
MemberInitial       varchar(32),
MemberAddress       varchar(100)    ,
MemberAddressId     numeric(10) NOT NULL,
MemberPhone     varchar(14),
MemberEMail     varchar(32) NOT NULL,
MemberPassword      varchar(32) NOT NULL,
MembershipId        numeric(10) NOT NULL,
MemberSinceDate     TIMESTAMP(3)        NOT NULL,
ZipCodeId   numeric(10) NOT NULL,
ZipCode     varchar(5)  NOT NULL,
CityId      numeric(10),
StateName   varchar(20) NOT NULL)PARTITION BY LIST (StateName);

-- create the two partioend table  west_member : for west_coast_server  and east_member : for east_coast_server
CREATE TABLE west_member PARTITION OF member_partitioned  FOR VALUES IN ('California','Texas');
CREATE TABLE east_member PARTITION OF member_partitioned  FOR VALUES IN ('Delaware','Florida','Georgia'
                                                            ,'Iowa','New Jersey','New York'
                                                            ,'Maryland','Pennsylvania');
```

Figure 14 : Showing the horizontal partition query for the coast servers



```
 populate the data into the partioned tables
ISERT INTO  member_partitioned
ELECT MemberId,
emberFirstName,
emberLastName ,
emberInitial,
emberAddress ,
emberAddressId ,
emberPhone ,
emberEMail,
emberPassword ,
embershipId,
emberSinceDate,
ipCodeId ,
ipCode,
ityId ,
tatename
ROM (select * from member inner join zipcode on member.memberaddressid=zipcode.zipcodeid
nner join state on zipcode.stateid=state.stateid )as boom;
```

Figure 15 : Showing the horizontal partition query for the coast servers

7. Distributed Transactions

- Let’s say we want to rent a dvd from “west_coast_red_box_2”, then the following steps will take place:

1. *First the membership status table will be checked to see if memberid is valid and is allowed to rent. If successfull it will follow ahead or else give error.*
2. *The rental table in the Red box 2 server will be updated.*
3. *The rental table , dvd and payment table in the West coast server will be updated.*

4. *The membershipstatus table in the Red box 2 server will be updated.*
5. *The membershipstatus table in the Red box 1 server will be updated.*

- Let's say we want to rent a dvd from "west_coast_red_box_2", then the following steps will take place But this time a member form east coast came to west coast for rental and rent from redbox 2:

1. *First the membership status table will be checked to see if memberid is valid and is allowed to rent.*
2. *Since, the memberid will not be available in WEST COAST SERVER, the west_coast_red_box_2 will directly contact HeadQuarters Cloud and verify from EAST COAST RECORDS. If success full it will follow ahead or else give error.*
3. *The rental table in the Red box 2 server will be updated.*
4. *The rental table , dvd and payment table in the East coast server will be updated via Head Quarters.*
5. *The membershipstatus table in the Red box 2 of East Coast server will be updated.*
6. *The membershipstatus table in the Red box 1 of East Coast server will be updated.*

Supplementary Materials

In this section, some additional information about the project is given. The supplementary informations includes the original small scale ERD diagram, the distributed network ERD diagrams for different servers and the Official block diagram of replication in PostgreSQL.

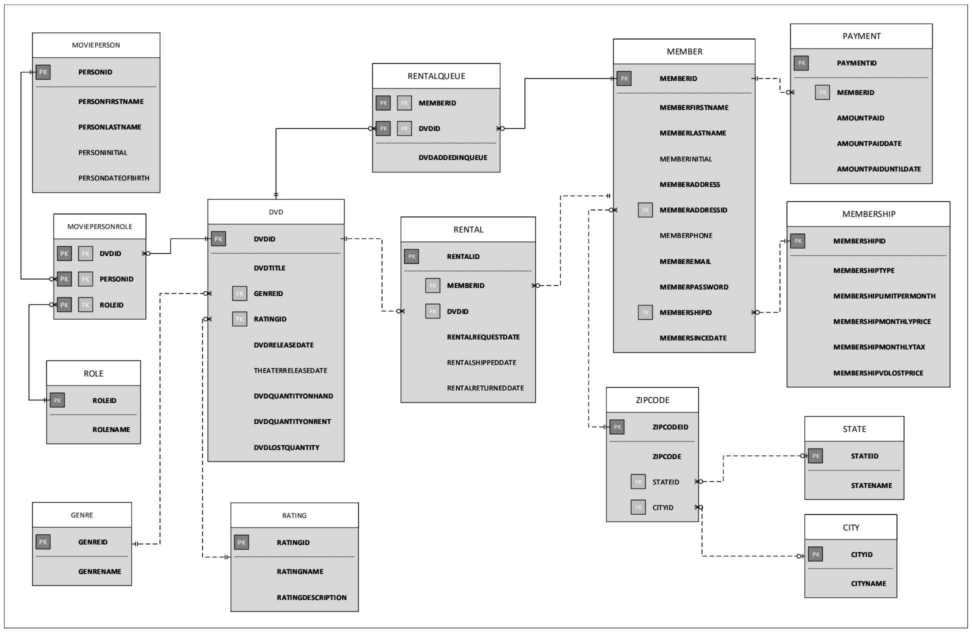


Figure 16 : Showing the original ERD for the Netflix Database.

1. The Official ERD

In Figure 16, the original ERD for Netflix Business is given. This ERD was made for the small business and cannot handle a distributed database.

2. The Partitioned ERD Diagram for the Coast Servers and Headquarter Cloud

In Figure 17, the Partitioned ERD Diagram for the database present in Coast Servers and Headquarter Cloud for Netflix Business is given. This ERD is made ignorer to scale a small business to a distributed database.

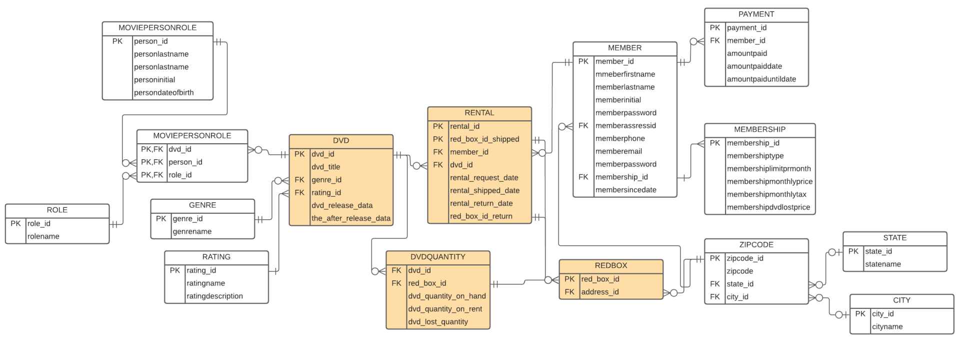


Figure 17 : Showing the Partitioned ERD for the Netflix Distributed Database. The Yellow color shows the changes made to the original ERD.

3. The Block Diagram for Streaming Replication Process in PostgreSQL

In Figure 18, the process of how WAL files transfer between Replica and Primary node is shown.

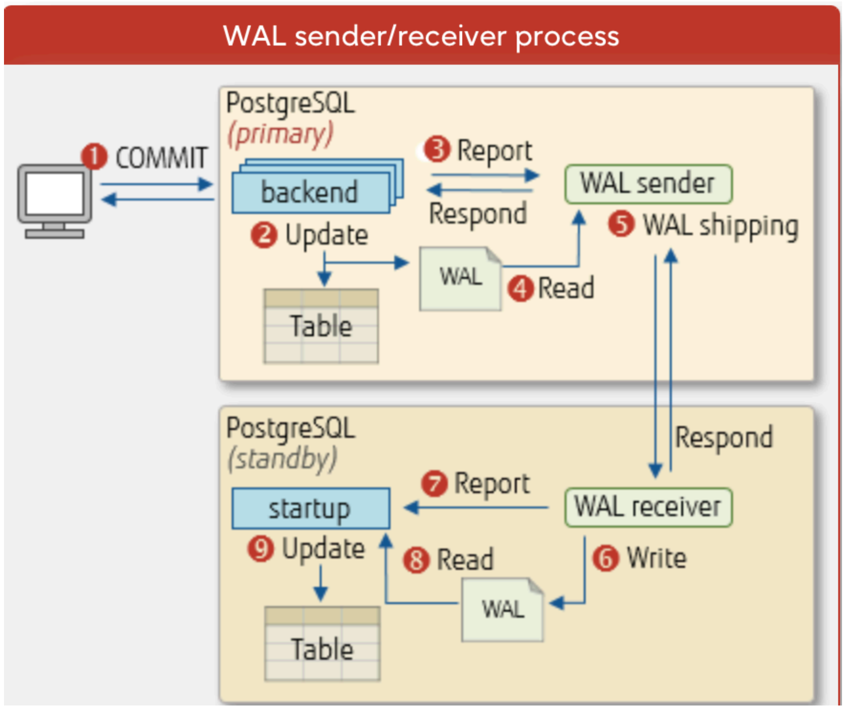


Figure 18 : Showing the block diagram for Streaming Replication in PostgreSQL

4. The Partitioned ERD Diagram for the Redbox Machines

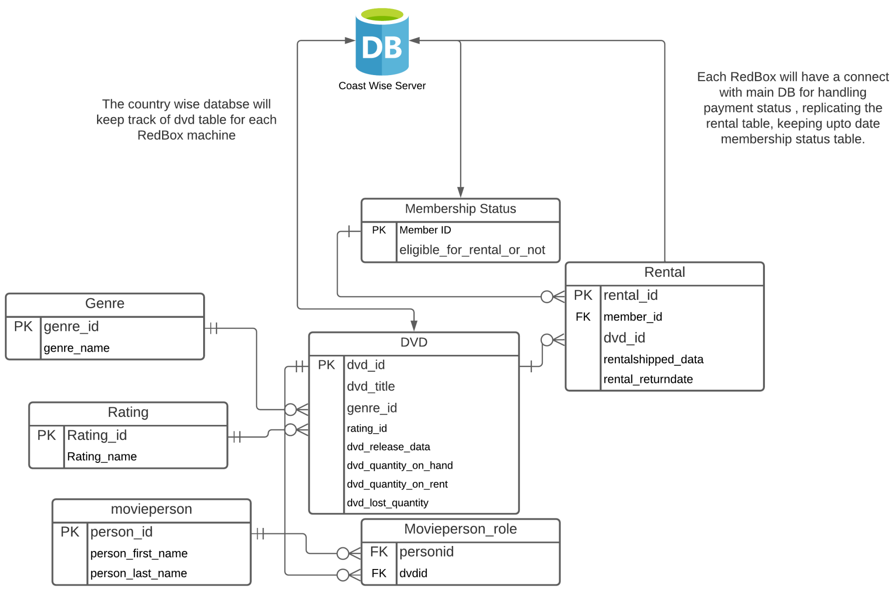


Figure 19: Showing the Partitioned ERD for the Netflix Distributed Database present in Redboxes. The tables here are Vertically Partitioned.