Proyecto II (Replicación y Modelo Multidimensional)

Profesor

Alberto Shum Chan

Bases de datos II

Integrantes

Taylor Hernandez (2020196104) y Pablo Muñoz (2020031899)

ITCR

OCT 26, 2021

Introducción

En este documento se describe el proceso necesario para lograr una replicación de bases de datos por medio de 'slony' y un modelo multidimensional a partir de la base de datos 'dvdrental' de 'postgresql', además se mostrará un *dashboard* desde la aplicación 'tableau' para un *Business Intelligence* más efectivo. Por último se crearán varios *stored procedures* para manejo de usuarios y datos.

Descripción del proyecto

Para efectos de longitud se asumirá que ya se tiene PostgreSQL y Slony, en caso contrario consultar:

- POSTGRESQL: https://www.datacamp.com/community/tutorials/installing-postgresql-windows-macosx?utm_source=adwords_ppc&utm_campaignid=1455363063&utm_adgroupid=65083631748&utm_device=c&utm_keyword=&utm_matchtype=b&utm_network=g&utm_adpostion=&utm_creative=332602034358&utm_targetid=aud-392016246653:dsa-429603003980&utm_loc_interest_ms=&utm_loc_physical_ms=9075466&gclid=CjwKCAjwzt6LBhBeEiwAbPGOgRSMcy7NR07Zt25MwIJA8IDTvr4kG2XdCXcBokxfKjvp9KEYhFo7vhoCUd4QAvD_BwE
- SLONY: https://www.enterprisedb.com/docs/slony/latest/01_installation/03_installing_slony_on_a_windows_host/

Después se procede a hacer el restore de la base de datos sample de Postgre, también se asume que este paso ya está hecho, en caso de que no consultar:

- https://www.postgresqltutorial.com/postgresql-sample-database/

Ahora sí comienza el proceso interesante, iniciamos con el procedure para insertar un cliente:

```
-- Procedure que permite insertar un cliente, recibiendo por entrada
toda la información correspondiente
-- Entrada: 3 INT, 3 String, 1 boolean y 2 dates.
CREATE OR REPLACE FUNCTION public."FC_INSERTAR_CLIENTE"(
    new_store_id smallint,
    new_first_name character varying,
    new_last_name character varying,
    new_email character varying,
    new_address_id smallint,
    new_activebool boolean,
    new_create_date date,
    new_last_update timestamp without time zone,
    new_active integer)
returns integer as $$
BEGIN
      insert into customer (store_id, first_name, last_name, email,
address_id, activebool, create_date, last_update, active)
      values     (new_store_id,     new_first_name,     new_last_name,
new_email,    new_address_id,    new_activebool,    new_create_date,
new_last_update, new_active);
      RETURN 1;
END;
$$ LANGUAGE plpgsql;
```

Después con el procedure insertar renta:

```
-- Procedure que permite insertar un cliente, recibiendo por entrada
toda la información correspondiente
-- Entrada: 3 INT, 3 String, 1 boolean y 2 dates.
CREATE OR REPLACE FUNCTION public."FC_INSERTAR_CLIENTE"(
    new_store_id smallint,
    new_first_name character varying,
    new_last_name character varying,
    new_email character varying,
```

```
    new_address_id smallint,
    new_activebool boolean,
    new_create_date date,
    new_last_update timestamp without time zone,
    new_active integer)
returns integer as $$
BEGIN
      insert into customer (store_id, first_name, last_name, email,
address_id, activebool, create_date, last_update, active)
      values    (new_store_id,    new_first_name,    new_last_name,
new_email,    new_address_id,    new_activebool,    new_create_date,
new_last_update, new_active);
      RETURN 1;
END;
$$ LANGUAGE plpgsql;
```

Ahora encontrar película:

```
-- Procedure que permite insertar un cliente, recibiendo por entrada
toda la información correspondiente
-- Entrada: 3 INT, 3 String, 1 boolean y 2 dates.
CREATE OR REPLACE FUNCTION public."FC_INSERTAR_CLIENTE"(
    new_store_id smallint,
    new_first_name character varying,
    new_last_name character varying,
    new_email character varying,
    new_address_id smallint,
    new_activebool boolean,
    new_create_date date,
    new_last_update timestamp without time zone,
    new_active integer)
returns integer as $$
BEGIN
      insert into customer (store_id, first_name, last_name, email,
address_id, activebool, create_date, last_update, active)
      values    (new_store_id,    new_first_name,    new_last_name,
new_email,    new_address_id,    new_activebool,    new_create_date,
new_last_update, new_active);
      RETURN 1;
END;
$$ LANGUAGE plpgsql;
```

Registrar retorno:

```
-- PROCEDIMIENTO QUE ACTUALIZA LA FECHA DE DEVOLUCIÓN DE UNA RENTA Y
ACTUALIZA EL INVENTARIO
-- ENTRADA: EL ID DE LA RENTA (SMALLINT)
CREATE OR REPLACE function public.FC_REGISTER_RETURN(
id_ren smallint)
returns integer as $$
declare
      -- Se declara la variable donde se almacenará el id
    id_inv integer;
begin

      -- Se registra la devolución
      UPDATE rental SET return_date = current_timestamp, last_update
= current_timestamp
      WHERE rental_id = id_ren;

      -- Se selecciona el id del inventario de esa renta
      SELECT inventory_id INTO id_inv FROM rental WHERE rental_id =
id_ren;

      -- Se hace un update de la fecha de la devolución
      UPDATE inventory SET last_update = current_timestamp
      WHERE inventory_id = id_inv;

      RETURN 1;
end;
$$ language plpgsql;
```

Por último se crean los roles respectivos:

```
create role EMP with login password 'contraseña123';
grant execute on function FC_FIND_MOVIE(movie Character Varying) to
EMP;
grant execute on function FC_INSERT_RENTAL(id_customer smallint,
id_staff smallint, id_inventory integer) to EMP;
grant execute on function FC_REGISTER_RETURN(id_rental smallint) to
EMP;
```

```
create role ADMIN with login password 'contraseña123' in role emp;
grant execute on function "FC_INSERTAR_CLIENTE"(
      new_store_id smallint,
      new_first_name character varying,
      new_last_name character varying,
      new_email character varying,
      new_address_id smallint,
      new_activebool boolean,
      new_create_date date,
      new_last_update timestamp without time zone,
      new_active integer) to ADMIN;

create user video with superuser;

create user empleado1 with role EMP;

create user administrador1 with role ADMIN;

GRANT EXECUTE ON ALL PROCEDURES IN SCHEMA public TO video;
```

Replicación

Para el proceso de replicación se seleccionó Slony, este se instala por medio del stack builder integrado en la instalación de PostgreSQL, se escogió este software ya que ofrece una sencilla replicación sin mucha complicación entre medias, además ya que lo ofrecen durante la instalación del PostgreSQL es mucho más sencillo y compatible. Para esta replicación se utilizó PostgreSQL 9.6. Además es importante mencionar que las dos bases de datos deben tener las mismas tablas que la otra para que la replicación funcione, de otra manera no se replicará en tablas no existentes.

Primeramente se debe buscar la ubicación en la que se encuentra postgreSQL, luego buscar la carpeta '/bin', dentro de esta se deberán crear dos documentos de texto con los nombres: Maestro, Esclavo.

El primer archivo de nombre 'Maestro' debe verse así:

```
cluster name = Cluster_Replicacion;

node 1 admin conninfo='dbname=dvdrental host=localhost user=postgres
```

```
password=12345';

node 2 admin conninfo='dbname=dvdrental_slave host=localhost
user=postgres password=12345';

init cluster(id=1, comment='MAESTRO');

create set (id=1, origin=1, comment='All pgbench tables');
set add table (set id=1, origin=1, id=1, fully qualified name =
'public.actor', comment='actor');
set add table (set id=1, origin=1, id=2, fully qualified name =
'public.address', comment='address');
set add table (set id=1, origin=1, id=3, fully qualified name =
'public.category', comment='category');
set add table (set id=1, origin=1, id=4, fully qualified name =
'public.city', comment='city');
set add table (set id=1, origin=1, id=5, fully qualified name =
'public.country', comment='country');
set add table (set id=1, origin=1, id=6, fully qualified name =
'public.customer', comment='customer');
set add table (set id=1, origin=1, id=7, fully qualified name =
'public.film', comment='film');
set add table (set id=1, origin=1, id=8, fully qualified name =
'public.film_actor', comment='film_actor');
set add table (set id=1, origin=1, id=9, fully qualified name =
'public.film_category', comment='film_category');
set add table (set id=1, origin=1, id=10, fully qualified name =
'public.inventory', comment='inventory');
set add table (set id=1, origin=1, id=11, fully qualified name =
'public.language', comment='language');
set add table (set id=1, origin=1, id=12, fully qualified name =
'public.payment', comment='payment');
set add table (set id=1, origin=1, id=13, fully qualified name =
'public.rental', comment='rental');
set add table (set id=1, origin=1, id=14, fully qualified name =
'public.staff', comment='staff');
set add table (set id=1, origin=1, id=15, fully qualified name =
'public.store', comment='store');
set add table (set id=1, origin=1, id=16, fully qualified name =
'public.film_dim', comment='Film_dim');
set add table (set id=1, origin=1, id=17, fully qualified name =
```

```
'public.category_subdim', comment='Category_subdim');
set add table (set id=1, origin=1, id=18, fully qualified name =
'public.actor_subdim', comment='Actor_subdim');
set add table (set id=1, origin=1, id=19, fully qualified name =
'public.film_x_actor', comment='film_x_actor');
set add table (set id=1, origin=1, id=20, fully qualified name =
'public.film_x_category', comment='film_x_category');
set add table (set id=1, origin=1, id=21, fully qualified name =
'public.country_subdim', comment='Country_subdim');
set add table (set id=1, origin=1, id=22, fully qualified name =
'public.city_subdim', comment='City_subdim');
set add table (set id=1, origin=1, id=23, fully qualified name =
'public.address_dim', comment='Address_dim');
set add table (set id=1, origin=1, id=24, fully qualified name =
'public.date_dim', comment='Date_dim');
set add table (set id=1, origin=1, id=25, fully qualified name =
'public.sucursal_dim', comment='Sucursal_dim');
set add table (set id=1, origin=1, id=26, fully qualified name =
'public.hechos', comment='Hechos');
set add table (set id=1, origin=1, id=27, fully qualified name =
'public.payment_register', comment='payment_register');



store node (id=2, comment = 'Slave node', EVENT NODE=1);
store path (server = 1, client = 2, conninfo='dbname=dvdrental
host=localhost user=postgres password=12345');
store path (server = 2, client = 1, conninfo='dbname=dvdrental_slave
host=localhost user=postgres password=12345');

store listen(origin=1, provider=1, receiver=2);

store listen(origin=2, provider=2, receiver=1);

SUBSCRIBE SET (ID=1, PROVIDER = 1, RECEIVER = 2, FORWARD = YES);
WAIT FOR EVENT(ORIGIN=1, CONFIRMED=ALL, WAIT ON=1);
```

Es importante cambiar los parámetros con el nombre ‘dbname’, ‘host’, ‘user’, ‘password’.

Y agregar o quitar los nombres de las tablas a replicar, estas deben ingresarse manualmente con el formato:

```
set add table (set id=1, origin=1, id=1(Incrementa en +1 por cada
tabla agregada), fully qualified name = 'public.city'(Nombre de la
tabla), comment='city'(Comentario extra para ubicar las tablas en la
replicación));
```

El segundo archivo 'Esclavo' debe verse así:

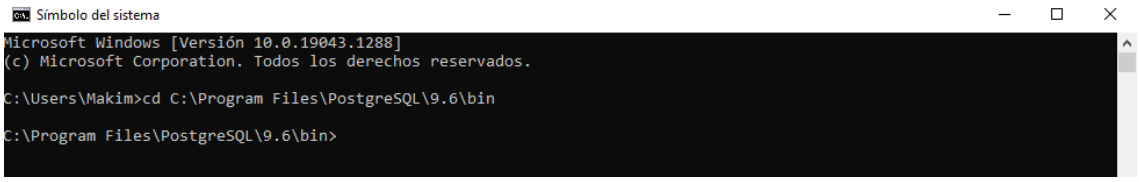
```
cluster name = Cluster_Replicacion;


node 1 admin conninfo = 'dbname=dvdrental host=localhost
user=postgres password=12345';
node 2 admin conninfo = 'dbname=dvdrental_slave host=localhost
user=postgres password=12345';



SUBSCRIBE SET(ID=1, PROVIDER = 1, RECEIVER = 2, FORWARD = YES);
```

Es importante cambiar los parámetros con el nombre 'dbname', 'host', 'user', 'password'.

Después de haber hecho esto debe dirigirse a la carpeta'/data' dentro del mismo directorio de postgreSQl y editar un archivo de nombre 'pg_hba.conf'. En la parte inferior debe agregar las dos bases de datos que quiere replicar, estas en su respectivo campo, como en este caso las dos bases se encuentran en la misma PC se ven igual.

```
# TYPE  DATABASE        USER            ADDRESS                 METHOD

# IPv4 local connections:
host    all             all             127.0.0.1/32            md5

# Maestro
#host   all             all             127.0.0.1/32            md5

# Esclavo
#host   all             all             127.0.0.1/32            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
```

Después se debe abrir una consola de windows (CMD) y por medio del uso del comando 'cd' debe dirigirse a la carpeta '/bin' de postgre, en este caso la dirección de la computadora utilizada es 'C:\Program Files\PostgreSQL\9.6\bin'. Este proceso debe repetirse en cuatro consolas diferentes, las cuatro tienen que estar abiertas al momento de la replicación.



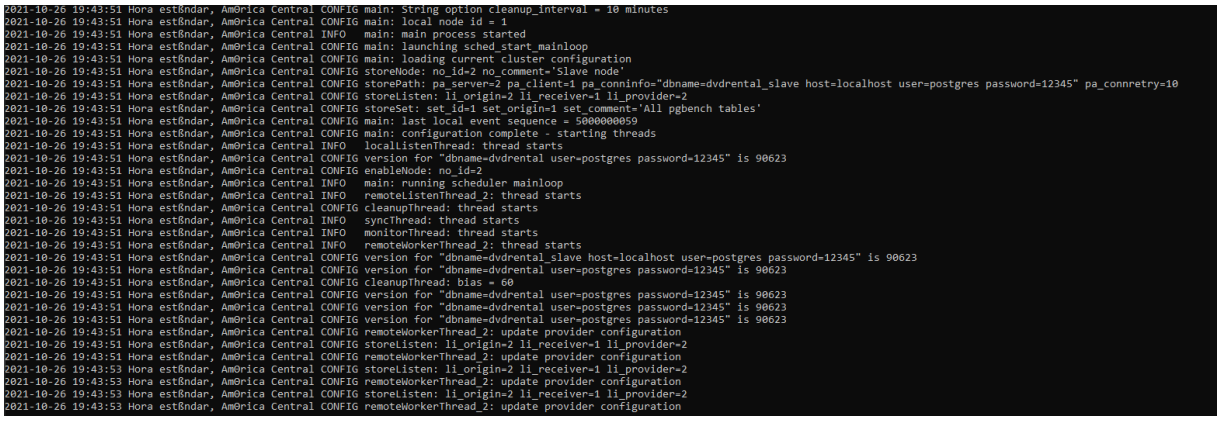
Ya estando en la carpeta en una de las consolas escribimos 'slonik maestro.txt' y presionamos enter.

```
C:\Program Files\PostgreSQL\9.6\bin>slonik maestro.txt
maestro.txt:44: waiting for event (1,5000000059) to be confirmed on node 2
```

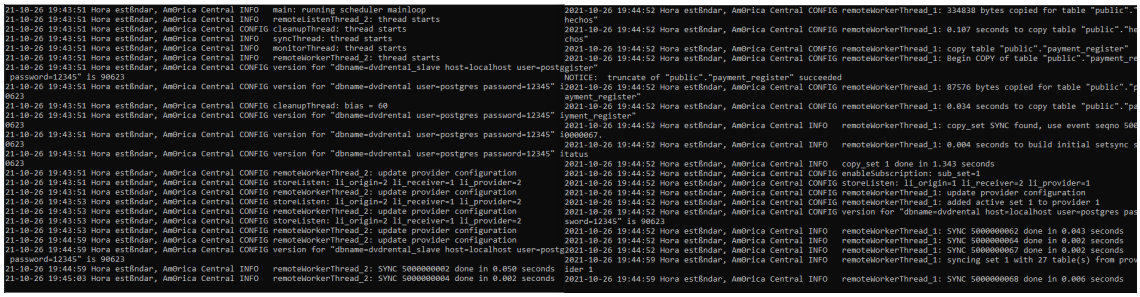
Debemos verificar que en la consola anterior salga un mensaje de espera. En otra de las terminales escribimos 'slonik esclavo.txt' y presionamos enter y esperamos otro mensaje de verificación.

```
C:\Program Files\PostgreSQL\9.6\bin>slonik esclavo.txt
waiting for events  (2,5000000001) only at (2,0) to be confirmed on node 1
```

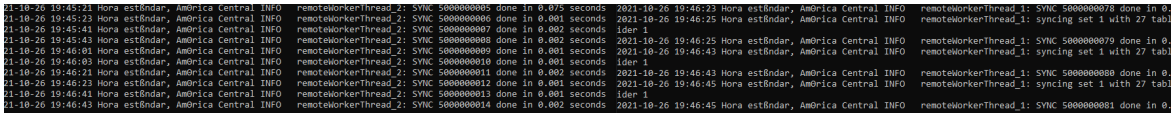
Después en una tercera terminal escribimos ‘slon Cluster_Replicacion "dbname=dvdrental user=postgres password=12345" ’ Debería verse así sin ningún warn o error.



En una cuarta ejecutamos el comando ‘slon Cluster_Replicacion "dbname=dvdrental_slave user=postgres password=12345"y también verificamos que no hayan warns o errores.



Para el momento en que en la parte inferior de las dos CMD’s salgan mensajes de sincronización la replicación está hecha, esta es una replicación sincrónica por lo que el usuario debe activarla cuando desee replicar las bases de datos.



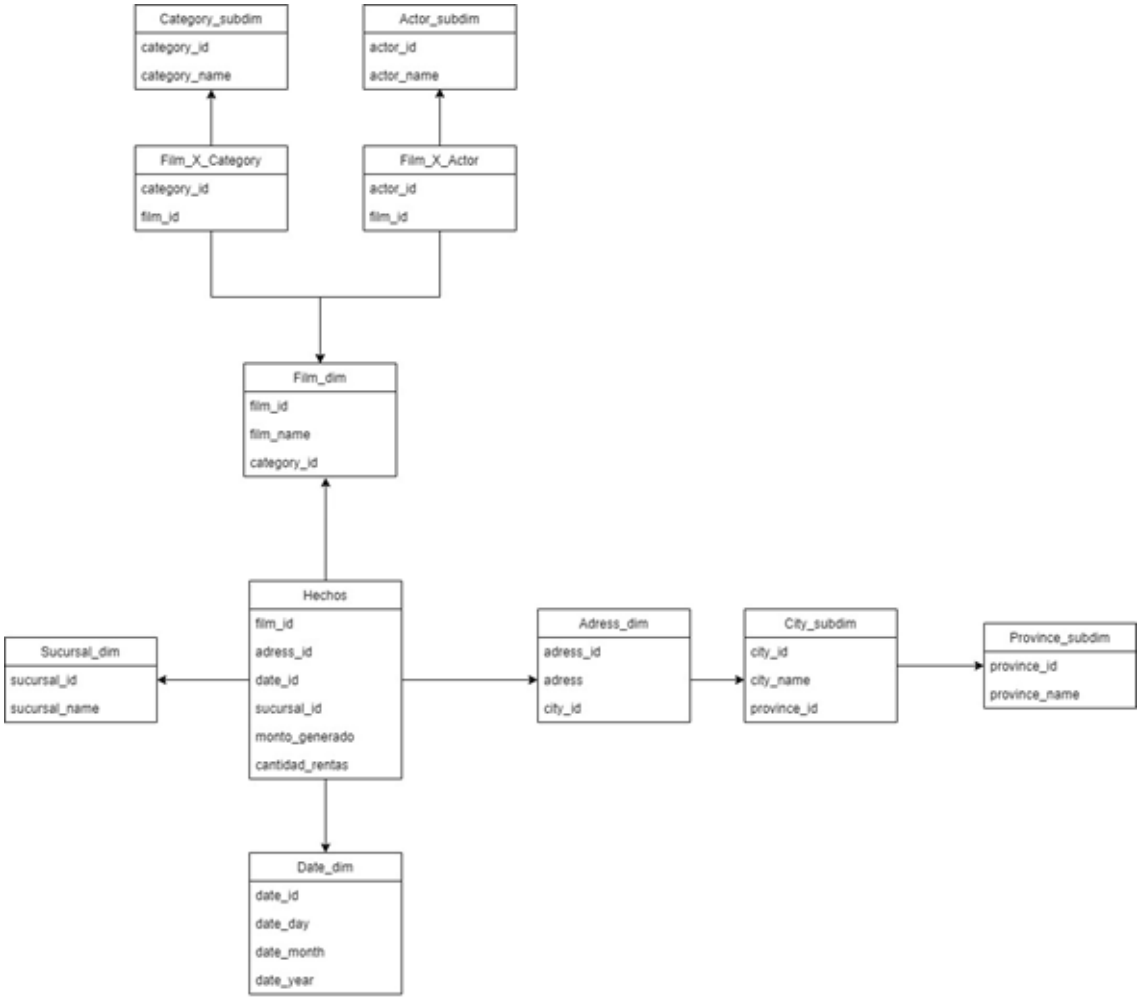
Verificamos en pgAdmin que se nos haya creado un nuevo schema con el nombre de '_Cluster_Replicacion'

,

La replicación está lista para este momento y se puede verificar en el pgAdmin.

ETL y Modelo Multidimensional

Primeramente, se planifica y visualiza el modelo estrella, utilizando diagramas y planificando los datos que poseerá cada dimensión. Para esto se creó un diagrama UML en donde se puede ver el modelo estrella completo y sus respectivas dimensiones y subdimensiones.



SQL de creación del modelo:

```
DROP TABLE Hechos;
DROP TABLE Address_dim;
DROP TABLE City_subdim;
DROP TABLE Country_subdim;
DROP TABLE film_x_actor;
DROP TABLE film_x_category;
DROP TABLE Film_dim;
DROP TABLE Category_subdim;
```

```
DROP TABLE Actor_subdim;
DROP TABLE Date_dim;
DROP TABLE Sucursal_dim;
DROP TABLE payment_register;


CREATE TABLE Film_dim(
      film_id SERIAL NOT NULL,
      film_name character varying(255) NOT NULL,
      CONSTRAINT dimfilm_pkey PRIMARY KEY (film_id)
);

CREATE TABLE Category_subdim(
      category_id SERIAL NOT NULL,
      category_name character varying(255) NOT NULL,
      CONSTRAINT subcat_pkey PRIMARY KEY (category_id)
);

CREATE  TABLE Actor_subdim(
      actor_id SERIAL NOT NULL,
      actor_name character varying(255) NOT NULL,
      CONSTRAINT subact_pkey PRIMARY KEY (actor_id)
);

CREATE TABLE film_x_actor(
      actor_id integer NOT NULL,
      film_id integer NOT NULL,
      CONSTRAINT film_x_actor_pkey PRIMARY KEY (actor_id, film_id),
          CONSTRAINT   film_x_actor_pkey_actor_id_fkey   FOREIGN   KEY
(actor_id)
        REFERENCES public.Actor_subdim (actor_id) MATCH SIMPLE
        ON UPDATE CASCADE
        ON DELETE RESTRICT,
    CONSTRAINT film_x_actor_film_id_fkey FOREIGN KEY (film_id)
        REFERENCES public.Film_dim (film_id) MATCH SIMPLE
        ON UPDATE CASCADE
        ON DELETE RESTRICT
);

CREATE  TABLE film_x_category(
      category_id integer NOT NULL,
```

```
      film_id integer NOT NULL,
      CONSTRAINT    film_x_category_pkey    PRIMARY    KEY    (film_id,
category_id),
          CONSTRAINT   film_x_category_category_id_fkey   FOREIGN   KEY
(category_id)
        REFERENCES public.Category_subdim (category_id) MATCH SIMPLE
        ON UPDATE CASCADE
        ON DELETE RESTRICT,
    CONSTRAINT film_x_category_film_id_fkey FOREIGN KEY (film_id)
        REFERENCES public.Film_dim (film_id) MATCH SIMPLE
        ON UPDATE CASCADE
        ON DELETE RESTRICT
);

CREATE TABLE Country_subdim(
      country_id SERIAL NOT NULL,
      country_name character varying(255) NOT NULL,
      CONSTRAINT subcount_pkey PRIMARY KEY (country_id)
);

CREATE TABLE City_subdim(
      city_id SERIAL NOT NULL,
      city_name character varying(255) NOT NULL,
      country_id integer NOT NULL,
      CONSTRAINT subcit_pkey PRIMARY KEY (city_id),
      CONSTRAINT      City_subdim_country_id_fkey      FOREIGN      KEY
(country_id)
        REFERENCES public.Country_subdim (country_id) MATCH SIMPLE
        ON UPDATE CASCADE
        ON DELETE RESTRICT
);

CREATE TABLE Address_dim(
      address_id SERIAL NOT NULL,
      address_name character varying(255) NOT NULL,
      city_id integer NOT NULL,
      CONSTRAINT dimadr_pkey PRIMARY KEY (address_id),
      CONSTRAINT Address_dim_city_id_fkey FOREIGN KEY (city_id)
        REFERENCES public.City_subdim (city_id) MATCH SIMPLE
        ON UPDATE CASCADE
        ON DELETE RESTRICT
```

```
);

CREATE TABLE Date_dim(
      date_id SERIAL NOT NULL,
      date_day INTEGER NOT NULL,
      date_month INTEGER NOT NULL,
      date_year INTEGER NOT NULL,
      CONSTRAINT dimdate_pkey PRIMARY KEY (date_id)
);

CREATE TABLE Sucursal_dim(
      sucursal_id SERIAL NOT NULL,
      CONSTRAINT dimsuc_pkey PRIMARY KEY (sucursal_id)
);

CREATE TABLE Hechos(
      date_id integer NOT NULL,
      sucursal_id integer NOT NULL,
      address_id integer NOT NULL,
      film_id integer NOT NULL,
      monto numeric(8,2) NOT NULL,
      cantidad integer NOT NULL,
    CONSTRAINT Hechos_date_id_fkey FOREIGN KEY (date_id)
        REFERENCES public.Date_dim (date_id) MATCH SIMPLE
        ON UPDATE CASCADE
        ON DELETE RESTRICT,
    CONSTRAINT Hechos_sucursal_id_fkey FOREIGN KEY (sucursal_id)
        REFERENCES public.Sucursal_dim (sucursal_id) MATCH SIMPLE
        ON UPDATE CASCADE
        ON DELETE RESTRICT,
      CONSTRAINT Hechos_adress_id_fkey FOREIGN KEY (address_id)
        REFERENCES public.Address_dim (address_id) MATCH SIMPLE
        ON UPDATE CASCADE
        ON DELETE RESTRICT,
      CONSTRAINT Hechos_film_id_fkey FOREIGN KEY (film_id)
        REFERENCES public.Film_dim (film_id) MATCH SIMPLE
        ON UPDATE CASCADE
        ON DELETE RESTRICT
);

CREATE TABLE payment_register(
```

```
    payment_id integer NOT NULL,
    CONSTRAINT register_pkey PRIMARY KEY (payment_id)
);
```

Luego, ya con el modelo en mente, se procede a realizar la extracción de datos. Como esta es una simulación de un ETL, no se utilizarán integration services, pero si se simulará la extracción utilizando tablas de la misma base de datos.

Para empezar, se tomará como el eje principal la tabla de payments, la cual es la que almacena aquellos datos contables que se utilizarán en la tabla de hechos. A partir de cada pago realizado, se logrará una conexión con el resto de las tablas, para así obtener los datos requeridos de las mismas.

```
select py.payment_id, py.amount,
st.store_id,
ad.address, ct.city, cot.country,
rt.rental_date,
fm.film_id, fm.title, fm.release_year
from payment py
inner join staff sf on sf.staff_id=py.staff_id
inner join store st on st.store_id=sf.store_id
inner join address ad on ad.address_id=st.address_id
inner join city ct on ct.city_id=ad.city_id
inner join country cot on cot.country_id=ct.country_id
inner join rental rt on rt.rental_id=py.rental_id
inner join inventory inv on inv.inventory_id=rt.inventory_id
inner join film fm on fm.film_id=inv.film_id;
```

Una vez realizada la conexión, se procederá a extraer los datos e insertarlos en nuestra simulación de ETL. Para esto se utilizará un cursor para ir línea por línea, dimensión por dimensión, verificando dato por dato, asegurándonos de que esté en la dimensión correspondiente, y si no está, que lo inserte y nos devuelva el id para así poder llenar la tabla de hechos.

```
-- Verificar e insertar store
select Verifycate_store(reg.store_id) into idstore;

-- verificar e insertar address
select Verifycate_address(reg.address, reg.city, reg.country) into idaddress;

-- verificar e insertar date
select Verifycate_date(reg.rental_date) into iddate;

-- verificar e insertar film
select Verifycate_film(reg.film_id, CONCAT(reg.title, to_char(reg.release_year, '9999')))
into idfilm;

INSERT INTO Hechos VALUES (iddate, idstore, idaddress, idfilm, reg.amount, 1);
```

Una vez procesada la línea de payments, se registrará el id de esta en una tabla de payments procesados, los cuales nos ayudaran para no repetir la inserción la próxima vez que se realice el proceso de “sincronización” del ETL.

```
IF Verifycate_payment(reg.payment_id)=0 then

    -- Verificar e insertar store
    select Verifycate_store(reg.store_id) into idstore;

    -- verificar e insertar address
    select Verifycate_address(reg.address, reg.city, reg.country) into idaddress;

    -- verificar e insertar date
    select Verifycate_date(reg.rental_date) into iddate;

    -- verificar e insertar film
    select Verifycate_film(reg.film_id, CONCAT(reg.title, to_char(reg.release_year, '9999')))
    into idfilm;

    INSERT INTO Hechos VALUES (iddate, idstore, idaddress, idfilm, reg.amount, 1);

    INSERT INTO payment_register VALUES(reg.payment_id);

END IF;
```

SQL completo de llenado:

```
-- confirm payment
CREATE OR REPLACE FUNCTION Verifycate_payment(id_pay integer)
RETURNS integer AS
$BODY$
```

```
DECLARE
	resp integer;
BEGIN
	SELECT count(*) into resp from payment_register pr where
pr.payment_id=id_pay;
	return resp;
END
$BODY$
LANGUAGE 'plpgsql';

-- confrim Sucursal dimension
CREATE OR REPLACE FUNCTION Verifycate_store(id_store integer)
RETURNS integer AS
$BODY$
DECLARE
	resp integer;
BEGIN
	SELECT count(*) into resp from Sucursal_dim sd where
sd.sucursal_id=id_store;
	IF resp=0 THEN
		INSERT INTO Sucursal_dim VALUES(id_store);
	END IF;
	return id_store;
END
$BODY$
LANGUAGE 'plpgsql';


-- confirm Address dimension
CREATE OR REPLACE FUNCTION Verifycate_address
(add_n character varying(255), city_n character varying(255),
count_n character varying(255))
RETURNS integer AS
$BODY$
DECLARE
	add_id integer;
	resp1 integer;
	cit_id integer;
	resp2 integer;
	count_id integer;
	resp3 integer;
```

```
BEGIN

      -- Country
      SELECT  count(*)  into  resp3  from  Country_subdim  where
country_name=count_n;
      IF resp3=0 then
            insert  into  Country_subdim  (country_name)  VALUES
(count_n);
      END IF;
      SELECT  ct.country_id  into  count_id  from  Country_subdim  ct
where country_name=count_n;

      -- City
      SELECT  count(*)  into  resp2  from  City_subdim  cs  where
cs.city_name=city_n and cs.country_id=count_id;
      IF resp2=0 then
            insert into City_subdim (city_name, country_id) VALUES
(city_n, count_id);
      END IF;
      SELECT  cs.city_id  into  cit_id  from  City_subdim  cs  where
cs.city_name=city_n and cs.country_id=count_id;

      -- Address
      SELECT  count(*)  into  resp1  from  Address_dim  ad  where
ad.address_name=add_n and ad.city_id=cit_id;
      IF resp1=0 then
            insert into Address_dim (address_name, city_id) VALUES
(add_n, cit_id);
      END IF;
      SELECT ad.address_id into add_id from Address_dim ad where
ad.address_name=add_n and ad.city_id=cit_id;

      return add_id;
END
$BODY$
LANGUAGE 'plpgsql';

-- confirm Date dimension
CREATE OR REPLACE FUNCTION Verifycate_date(date_in timestamp without
time zone) RETURNS integer AS
$BODY$
```

```
DECLARE
      date_id integer;
      resp integer;
BEGIN
      SELECT count(*) into resp from Date_dim
      where date_day=EXTRACT(DAY FROM date_in)
      and date_year=EXTRACT(YEAR FROM date_in)
      and date_month=EXTRACT(MONTH FROM date_in);
      IF resp=0 then
            insert into Date_dim (date_day, date_year, date_month)
            VALUES  (EXTRACT(DAY  FROM  date_in),  EXTRACT(YEAR  FROM
date_in), EXTRACT(MONTH FROM date_in));
      END IF;
      SELECT dd.date_id into date_id from Date_dim dd
      where date_day=EXTRACT(DAY FROM date_in)
      and date_year=EXTRACT(YEAR FROM date_in)
      and date_month=EXTRACT(MONTH FROM date_in);
      return date_id;
END
$BODY$
LANGUAGE 'plpgsql';


-- confirm Film dimension
CREATE  OR  REPLACE  FUNCTION  Verifycate_film(f_id  integer,  f_name
character varying(255)) RETURNS integer AS
$BODY$
DECLARE
      movie_id integer;
      cat_id integer;
      act_id integer;
      resp integer;
      reg1          RECORD;
      reg2          RECORD;
      cur_act  cursor (film_i integer) for
            select  CONCAT(ac.first_name,  ac.last_name)  actor_name
from film_actor fa
            inner join actor ac on ac.actor_id=fa.actor_id
            where fa.film_id=film_i;
      cur_cat  cursor (film_i integer) for
            select ca.name category_name from film_category fc
```

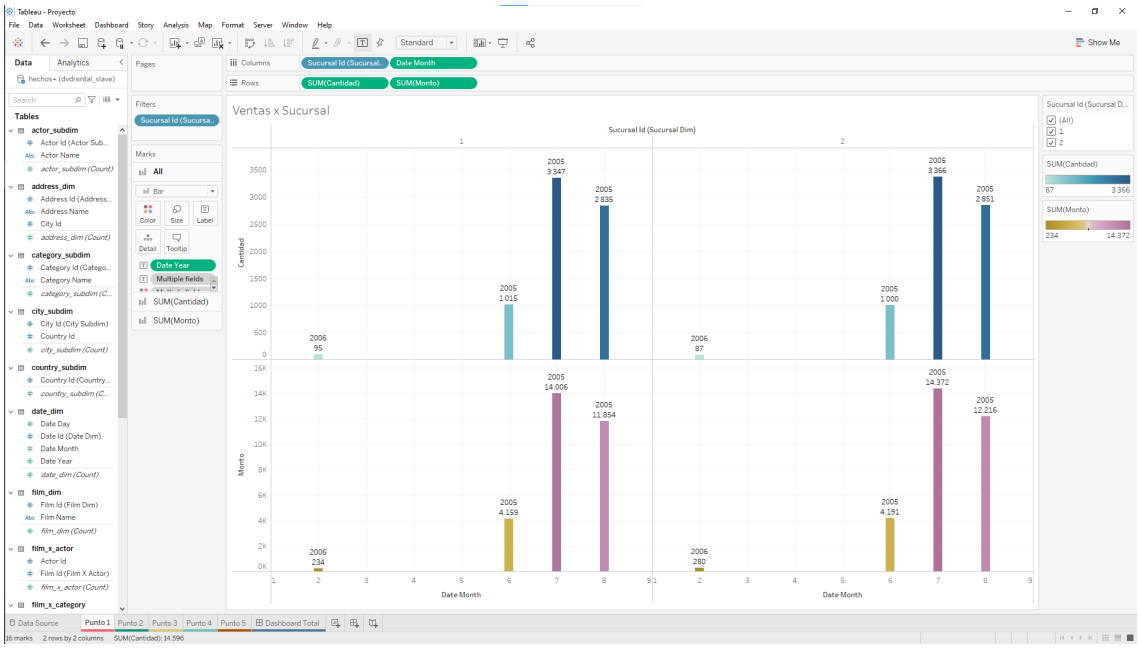
```
            inner join category ca on ca.category_id=fc.category_id
            where fc.film_id=film_i;

BEGIN

      -- Se encuentra y verifica la pelicula
      SELECT   count(*)   into   resp   from   Film_dim   fl   where
fl.film_name=f_name;
      IF resp=0 then
            insert into Film_dim (film_name)
            VALUES (f_name);
      END IF;
      SELECT  fl.film_id  into  movie_id  from  Film_dim  fl  where
fl.film_name=f_name;

      -- Se verifican los actores
      open cur_act(f_id);
      FETCH cur_act INTO reg1;
      WHILE( FOUND ) LOOP

            SELECT  count(*)  into  resp  from  Actor_subdim  ac  where
ac.actor_name=reg1.actor_name;
            IF resp=0 then
                  insert into Actor_subdim (actor_name)
                  VALUES (reg1.actor_name);
            END IF;
            SELECT  ac.actor_id  into  act_id  from  Actor_subdim  ac
where ac.actor_name=reg1.actor_name;

            SELECT  count(*)  into  resp  from  film_x_actor  fa  where
fa.film_id=movie_id and fa.actor_id=act_id;
            IF resp=0 then
                  insert into film_x_actor (film_id, actor_id)
                  VALUES (movie_id, act_id);
            END IF;
      FETCH cur_act INTO reg1;

      END LOOP ;
      close cur_act;

      -- Se verifican las categorias
```

```
      open cur_cat(f_id);
      FETCH cur_cat INTO reg2;
      WHILE( FOUND ) LOOP

            SELECT count(*) into resp from Category_subdim ca where
ca.category_name=reg2.category_name;
            IF resp=0 then
                  insert into Category_subdim (category_name)
                  VALUES (reg2.category_name);
            END IF;
            SELECT ca.category_id into cat_id from Category_subdim
ca where ca.category_name=reg2.category_name;

            SELECT count(*) into resp from film_x_category fc where
fc.film_id=movie_id and fc.category_id=cat_id;
            IF resp=0 then
                  insert into film_x_category (film_id, category_id)
                  VALUES (movie_id, cat_id);
            END IF;
      FETCH cur_cat INTO reg2;

      END LOOP ;
      close cur_cat;

      return movie_id;
END
$BODY$
LANGUAGE 'plpgsql';

CREATE OR REPLACE PROCEDURE SP_FILL_ETL()
LANGUAGE 'plpgsql'
AS $$
declare
      reg          RECORD;
      cur_films  cursor for
            select py.payment_id, py.amount,
            st.store_id,
            ad.address, ct.city, cot.country,
            rt.rental_date,
            fm.film_id, fm.title, fm.release_year
            from payment py
```

```
            inner join staff sf on sf.staff_id=py.staff_id
            inner join store st on st.store_id=sf.store_id
            inner join address ad on ad.address_id=st.address_id
            inner join city ct on ct.city_id=ad.city_id
            inner join country cot on cot.country_id=ct.country_id
            inner join rental rt on rt.rental_id=py.rental_id
            inner        join        inventory        inv        on
inv.inventory_id=rt.inventory_id
            inner join film fm on fm.film_id=inv.film_id;
      idstore integer;
      idfilm integer;
      idaddress integer;
      iddate integer;
BEGIN
      open cur_films;
      FETCH cur_films INTO reg;

      WHILE( FOUND ) LOOP
            IF Verifycate_payment(reg.payment_id)=0 then

                  -- Verificar e insertar store
                  select     Verifycate_store(reg.store_id)     into
idstore;

                  -- verificar e insertar address
                  select  Verifycate_address(reg.address,  reg.city,
reg.country) into idaddress;

                  -- verificar e insertar date
                  select    Verifycate_date(reg.rental_date)    into
iddate;

                  -- verificar e insertar film
                  select                Verifycate_film(reg.film_id,
CONCAT(reg.title, to_char(reg.release_year, '9999')))
                  into idfilm;

                  INSERT   INTO   Hechos   VALUES   (iddate,   idstore,
idaddress, idfilm, reg.amount, 1);

                  INSERT            INTO            payment_register
```

```
VALUES(reg.payment_id);

            END IF;
      FETCH cur_films INTO reg;
      END LOOP ;
      close cur_films;
END; $$;

CALL SP_FILL_ETL();
```

Tableau y *Dashboard*



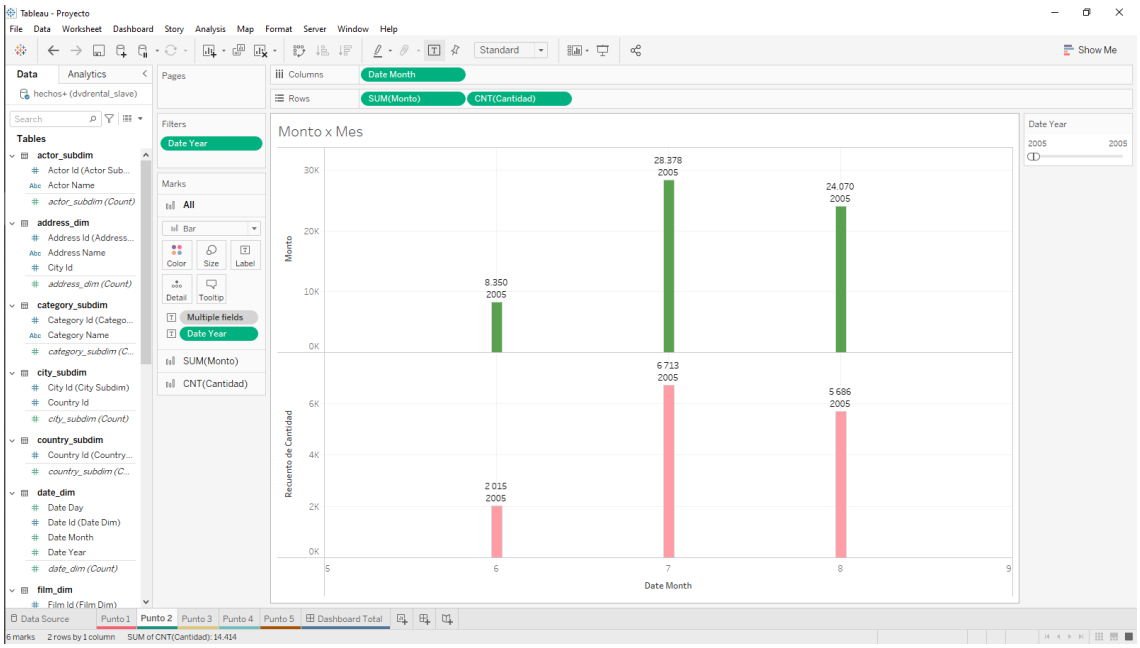


Tableau - Proyecto
File Data Worksheet Dashboard Story Analysis Map Format Server Window Help
Standard
Show Me
Data
Analytics
hechos+ (dvdrental_slave)
Search
Tables
actor_subdim
Actor Id (Actor Sub...
Actor Name
actor_subdim (Count)
address_dim
Address Id (Address...
Address Name
City Id
address_dim (Count)
category_subdim
Category Id (Catego...
Category Name
category_subdim (C...
city_subdim
City Id (City Subdim)
Country Id
city_subdim (Count)
country_subdim
Country Id (Country...
country_subdim (C...
date_dim
Date Day
Date Id (Date Dim)
Date Month
Date Year
date_dim (Count)
film_dim
Film Id (Film Dim)
Pages
Filters
Date Year
Marks
All
Bar
Color
Size
Label
Detail
Tooltip
Multiple fields
Date Year
SUM(Monto)
CNT(Cantidad)
Columns
Date Month
Rows
SUM(Monto)
CNT(Cantidad)
Monto x Mes
Monto
30K
20K
10K
0K
8.350
2005
28.378
2005
24.070
2005
Recuento de Cantidad
6K
4K
2K
0K
2 015
2005
6 713
2005
5 686
2005
5
6
7
8
9
Date Month
Date Year
2005
2005
Data Source
Punto 1
Punto 2
Punto 3
Punto 4
Punto 5
Dashboard Total
6 marks
2 rows by 1 column
SUM of CNT(Cantidad): 14.414

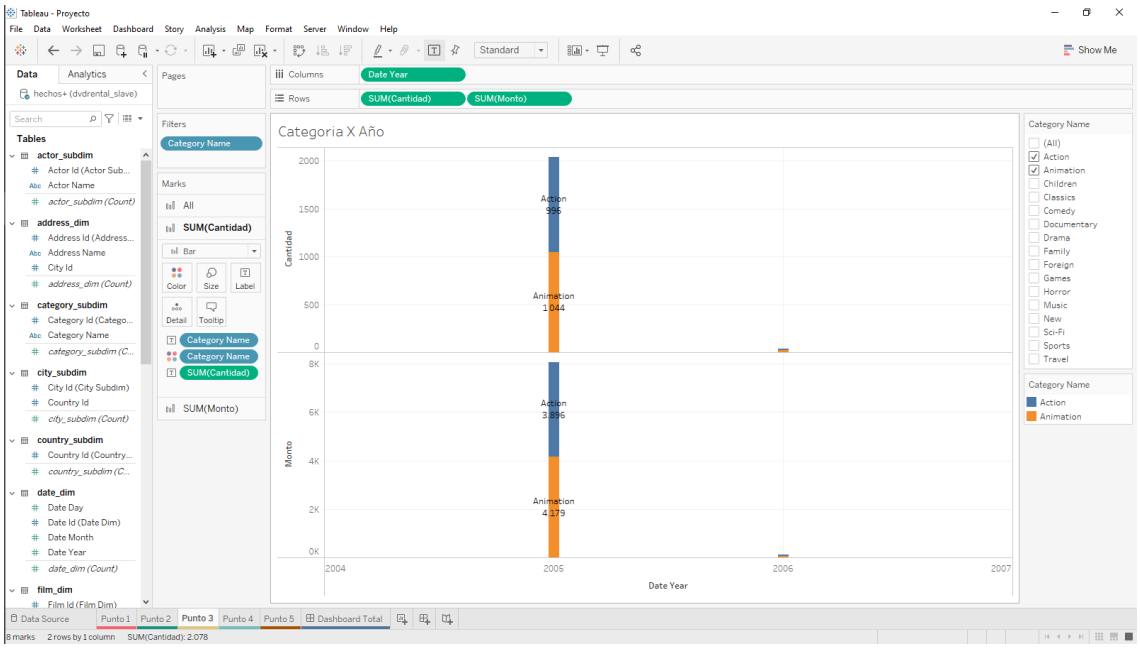


Tableau - Proyecto
Categoria X Año
Cantidad
Monto
Date Year
Action 986
Animation 1,044
Action 3,896
Animation 4,179
Punto 3

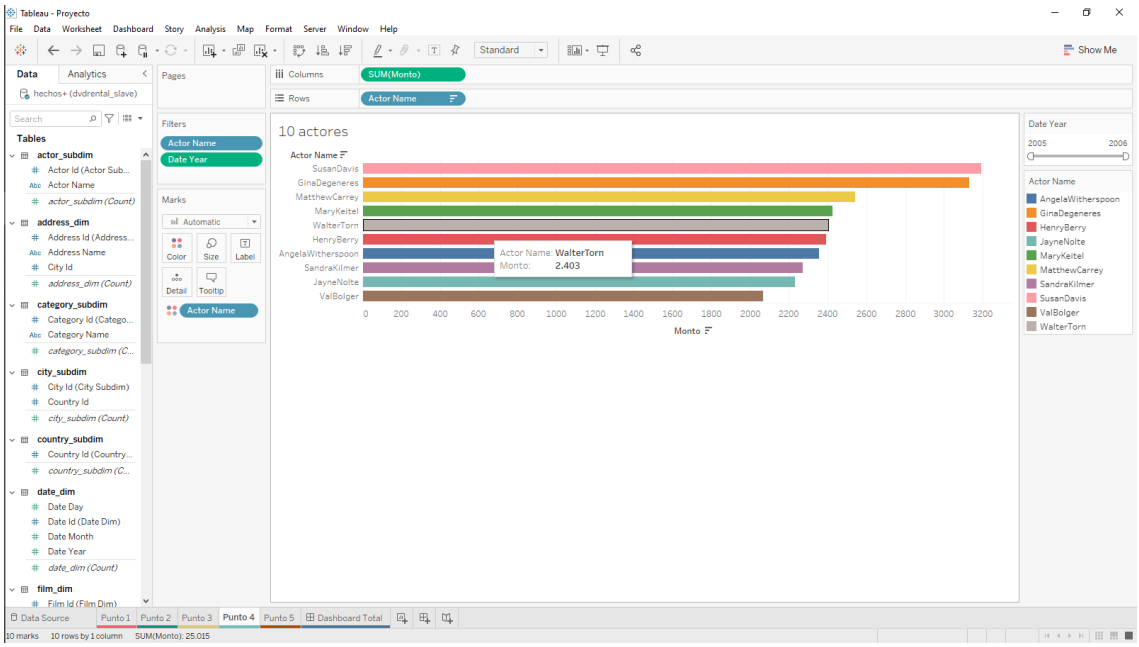
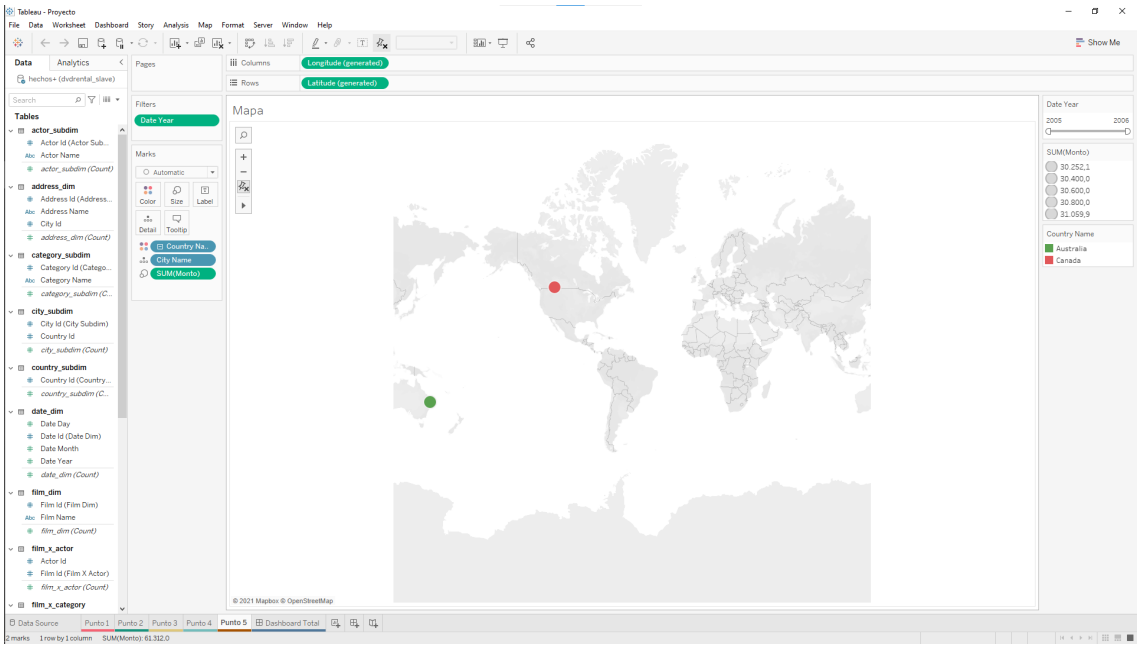


Tableau - Proyecto
10 actores
Actor Name: WalterTorn
Monto: 2,403
Monto
Punto 4



Conclusión

Este proyecto nos ha mostrado una cara poco conocida de los sistemas de bases de datos, lleva a otro nivel los sistemas mismos, ya que ya no es solo realizar procedimientos o funciones, ahora tenemos que analizar el manejo de roles, tener usuarios con sus permisos propios, asegurarnos de que la información esté correcta en la base slave, que todo se replique correctamente, tenemos que analizar los datos que se extraen al realizar un ETL, fijarnos que los valores contables no se repitan. Ya todo es un proceso más allá de solo utilizar la herramienta individualmente, ahora es un proceso más complejo, donde las cosas tienen un por que y un como.

El proyecto nos ha abierto la mente a una nueva forma de ver los sistemas, y ha sido una gran manera de acercarnos poco a poco a un manejo más real de bases de datos en lo que puede ser el mundo laboral actual. Tanto el manejo de seguridad, como la replicación y la realización de un ETL, nos hacen ver las bases como una herramienta conjunta, donde cada quien tiene sus responsabilidades, cada persona sus permisos, y cada dato su procedencia. Todo se encuentra enlazado de una u otra manera, creando así un gran ecosistema de trabajo.

Bibliografía:

https://www.postgresqltutorial.com/postgresql-sample-database/

https://www.tableau.com

https://wiki.postgresql.org/wiki/Slony

https://www.howtoforge.com/configuring-slony-i-cascading-replication-on-postgresql-8.3

https://www.youtube.com/watch?v=Eh6nGFaq4AU