udp Escuela de Informática y Telecomunicaciones

Bases de Datos Avanzada

Actividad 7: Consistencia e integridad en Bases de Datos

Autores:
Nicolás Correa
Joaquín Fernández
David Pazán

Profesor:
Juan Ricardo Giadach

junio 13 del 2021

Índice

1. Introducción

Una base de datos relacional debe cumplir con las propiedades ACID (atomicidad, consistencia, aislamiento y durabilidad). En el presente informe se analizara la consistencia e integridad en un sistema de base de datos al ser sometida a distintas pruebas que necesitan algún tipo de proceso de seguridad para conservar estas propiedades.

2. Herramientas

El ordenador que se utiliza para esta actividad es el mismo que se utilizo para las anteriores, siendo poseedor de las siguientes especificaciones.

- Acer Aspire E-15-575G-76P4
- Sistema Operativo: Linux, con distribución Elementary Os versión Hera
- Intel core i7-7500U 4 cpus.
- SSD KINGSTON SA400 con capacidad 480gb
- Velocidad para lectura de 550 MB/s y escritura 450 MB/s.

Dicho hardware se mantiene en el desarrollo de los trabajos con la intención de obtener datos relacionados entre ellos.

3. Actividades

3.1. NOT NULL

Las bases de datos relacionales tienen la característica de soportar distintos tipos de variables, algunas de las cuales pueden tener valores de vació. En el siguiente apartado se estudiara lo opuesto, como evitar que una variable tome el valor de nulo y como responde la base de datos al ser sometida a valores de sentencias erróneas.

Primero que todo se crea el entorno para la aplicación de la sentencia $NotNull$; esta incluye la variable "nombre" con la limitación.

```
create table clientes(
    rut VARCHAR primary key,
    nombre VARCHAR not null,
    edad NUMERIC,
    sueldo NUMERIC
);
```

Tras la creación del entorno a trabajar, se generan dos querys las cuales han de generar un cambio en el dato "nombre VARCHAR not null", buscando reemplazarlo por un valor nulo. La base de datos responde no registrando el cambio para el caso 1 y mostrando una alerta de ERROR para un valor que viola la restricción "not null" para el caso 2, también evitando su registro.

```
-- CASO 1
update clientes set nombre = null where rut = '1';
-- CASO 2
insert into clientes(rut,nombre,edad,sueldo) values('1001',null,34,5000000);
```

```
tablas_aux=*# update clientes set nombre = null where rut = '1';
UPDATE 0
```

Figura 1: Comprobación de data no nula Caso 1

```
tablas_aux=# insert into clientes(rut,nombre,edad,sueldo) values('1001',null,34,500000);insert
dad,sueldo) values('1001',null,34,500000);
ERROR:  el valor null para la columna «nombre» viola la restricción not null
DETALLE:  La fila que falla contiene (1001, null, 34, 500000).
```

Figura 2: Comprobación de data no nula Caso 2

3.2. CHECK

Un CHECK es una restricción o limitación que deben cumplir los datos para que sean considerados válidos. En el siguiente apartado se estudia el comportamiento de la base de datos al ser sometida a sentencias erróneas al no cumplir con los parámetros proporcionados en el CHECK.

Para generar un entorno de trabajo es necesario crear una tabla que utilice algún CHECK en este caso limitara los valores del atributo sueldos, además es necesario alterar la tabla agregándole una restricción que restrinja la variable edad, como se puede apreciar continuación.

```
create table clientes(
    rut VARCHAR primary key not null,
    nombre VARCHAR not null,
    edad NUMERIC,
    sueldo NUMERIC check (sueldo>0), check(sueldo <= 1000000)
);
alter TABLE clientes add constraint enMisTiempos
CHECK (
    edad >= 18
);
```

Para poner a prueba los $CHECK$ creados, se ejecuta alguna consulta que someta a la base de datos a responder en caso de valores erróneos, esta se puede observar en el caso 1.

```
--CASO 1
copy clientes from '/home/joaquin/BBDD/BDDA7/Clientes.txt' demiliter ',';
--CASO 2
update clientes set edad = 14 where rut = '100';
```

Como se puede apreciar en las siguientes imágenes, no se encuentra un caso que cumpla la condición (viola la regla del $CHECK$) por lo que la base de datos responde con mensajes de error.

```
tablas_aux=# create table clientes(
tablas_aux(#     rut VARCHAR primary key not null,
tablas_aux(#     nombre VARCHAR not null,
tablas_aux(#     edad NUMERIC,
tablas_aux(#     sueldo NUMERIC check (sueldo>1000000)
tablas_aux(# );
CREATE TABLE
tablas_aux=# copy clientes from '/home/joaquin/BDDA/BDDA7/Clientes.txt' delimiter ',';
ERROR:  el nuevo registro para la relación «clientes» viola la restricción «check» «clientes_sueldo_check»
DETALLE:  La fila que falla contiene (1, Andrée, 72, 829572).
CONTEXTO:  COPY clientes, linea 1: «1,Andrée,72,829572»
```

Figura 3: Comprobación del Check con insert

```
tablas_aux=*# alter TABLE clientes add constraint enMisTiempos
tablas_aux-*# CHECK (
tablas_aux(*#     edad >= 18
tablas_aux(*# );
ALTER TABLE
tablas_aux=*# update clientes set edad = 14 where rut = '100';
ERROR:  el nuevo registro para la relación «clientes» viola la restricción «check» «enmistiemp
os»
DETALLE:  La fila que falla contiene (100, Michèle, 14, 804511).
tablas_aux=!#
```

Figura 4: Comprobación del Check con update

3.3. FOREIGN KEY

Una FOREIGN KEY es una clave que sirve para relacionar dos tablas. El campo FOREIGN KEY se relaciona o vincula con la PRIMARY KEY de otra tabla de la base de datos. A continuación se pondrán a prueba 2 tablas relacionadas a través de una llave foránea.

Para comenzar es necesario crear el entorno de trabajo ejecutando los script correspondientes a la creación de las 2 tablas. El desarrollo se puede observar a continuación.

```
create table clientes(
    rut VARCHAR primary key,
    nombre VARCHAR not null,
    edad NUMERIC,
    sueldo NUMERIC
);
create table beneficio(
    id NUMERIC primary key,
    rut_cliente VARCHAR,
    constraint fk_cliente
    FOREIGN KEY (rut_cliente)
    REFERENCES clientes(rut)
);

--Aparecen las tablas respectivas, con la informaci n de las llaves for neas
\d beneficio
\d clientes

--Botar la tabla portadora de la FOREING KEY
--CASO 1
drop table clientes;
--CASO 2
update clientes set rut = '1001' where edad = 18;
```

El muestreo de las tablas creadas (clientes y beneficios) indican los índices que han de ser las respectivas $FOREINGKEY$ y las restricciones que posee, esto en la primera tabla mencionada.

Por otro lado se tiene la vista en el momento de intentar botar la tabla ha de entregar un error, el cual indica la dependencia que tiene la tabla beneficios con la que se busca borrar.

```
tablas_aux=# \d beneficio
                    Tabla «public.beneficio»
   Columna   |       Tipo        | Ordenamiento | Nulable  | Por omisión
-------------+-------------------+--------------+----------+-------------
 id          | numeric           |              | not null |
 rut_cliente | character varying |              |          |
Índices:
    "beneficio_pkey" PRIMARY KEY, btree (id)
Restricciones de llave foránea:
    "fk_cliente" FOREIGN KEY (rut_cliente) REFERENCES clientes(rut)

tablas_aux=# \d clientes
                     Tabla «public.clientes»
 Columna |       Tipo        | Ordenamiento | Nulable  | Por omisión
---------+-------------------+--------------+----------+-------------
 rut     | character varying |              | not null |
 nombre  | character varying |              | not null |
 edad    | numeric           |              |          |
 sueldo  | numeric           |              |          |
Índices:
    "clientes_pkey" PRIMARY KEY, btree (rut)
Referenciada por:
    TABLE "beneficio" CONSTRAINT "fk_cliente" FOREIGN KEY (rut_cliente) REFERENCES clientes(rut)
```

Figura 5: Atributos tablas con FK

```
tablas_aux=# drop table clientes;
ERROR:  no se puede eliminar tabla clientes porque otros objetos dependen de él
DETALLE:  restricción «fk_cliente» en tabla beneficio depende de tabla clientes
SUGERENCIA:  Use DROP ... CASCADE para eliminar además los objetos dependientes.
tablas_aux=# drop table beneficio;
DROP TABLE
tablas_aux=# drop table clientes;
DROP TABLE
```

Figura 6: Comprobación FK Caso 1

```
tablas_aux=*# update clientes set rut = '1001' where edad = 18;
ERROR:  llave duplicada viola restricción de unicidad «clientes_pkey»
DETALLE:  Ya existe la llave (rut)=(1001).
```

Figura 7: Comprobación FK Caso 2

3.4. ASSERTION

ASSERTION se utiliza para especificar tipos adicionales de restricciones que están fuera del alcance de las restricciones del modelo relacional incorporado (claves primarias y únicas, integridad de la entidad e integridad referencial). La técnica básica para escribir tales afirmaciones es especificar una consulta que seleccione cualquier tupla que viole la condición deseada. Utilizando el código en PL-pgSQL puesto a continuación es posible generar un un código que elimine la tabla beneficio en el caso de no existir personas con beneficio dentro de ella.

```
1 --Caso 1
2 do $$
3 declare
4   nPersonas integer;
5 begin
6   select count(*)
```

```
  into nPersonas
  from clientes, beneficio
  where clientes.rut = beneficio.rut_cliente and clientes.edad>=65;
  drop table beneficio;
  assert nPersonas > 0, 'Si hay personas con beneficio que sean mayores de edad
    por lo tanto no se elimina';
end$$;

--Caso 2
do $$
declare
  nPersonas integer;
begin
  select count(*)
  into nPersonas
  from clientes, beneficio
  where clientes.rut = beneficio.rut_cliente and clientes.edad>=65;

  assert nPersonas > 0, 'Hay personas con beneficio';
end$$;
```

La diferencia entre caso 1 y caso 2 es que en el primero existen casos en el que el valor de "nPersonas" es menor o igual a cero provocando que la tabla sea eliminada, la base de datos responde con un mensaje de "DO", mientras que en el segundo este valor es mayor a cero por lo que no se elimina la tabla, mostrando un mensaje de ERROR.

```
tablas_aux=*# do $$
tablas_aux$*# declare
tablas_aux$*#    nPersonas integer;
tablas_aux$*# begin
tablas_aux$*#    select count(*)
tablas_aux$*#    into nPersonas
tablas_aux$*#    from clientes, beneficio
tablas_aux$*#    where clientes.rut = beneficio.rut_cliente and clientes.edad>=65;
tablas_aux$*#    drop table beneficio;
tablas_aux$*#    assert nPersonas = 0, 'Si hay personas con beneficio que sean mayores de edad por lo tanto no se elimina';
tablas_aux$*# end$$;
DO
```

Figura 8: Comprobación del ASSERTION deleted

```
tablas_aux=*# do $$
tablas_aux$*# declare
tablas_aux$*#    nPersonas integer;
tablas_aux$*# begin
tablas_aux$*#    select count(*)
tablas_aux$*#    into nPersonas
tablas_aux$*#    from clientes, beneficio
tablas_aux$*#    where clientes.rut = beneficio.rut_cliente and clientes.edad>=65;
tablas_aux$*#
tablas_aux$*#    assert nPersonas > 0, 'Hay personas con beneficio';
tablas_aux$*# end$$;
ERROR:  Hay personas con beneficio
CONTEXTO:  función PL/pgSQL inline_code_block en la línea 10 en ASSERT
```

Figura 9: Comprobación del ASSERTION select

3.5. TRIGGERS

Un Trigger o disparador es una acción definida en una tabla de la base de datos y ejecutada automáticamente por una función programada por el desarrollador. Esta acción se activará, según la sentencia $INSERT$, $UPDATE$ o $DELETE$ de la tabla afiliada. De esta manera tiene distintas formas de aplicación las cuales pueden ser:

- Para que ocurra antes de cualquier $INSERT$, $UPDATE$ o $DELETE$
- Para que ocurra después de cualquier $INSERT$, $UPDATE$ o $DELETE$
- Para que se ejecute una sola vez por comando SQL
- Para que se ejecute por cada linea afectada por un comando SQL

A continuación se presenta el código ejecutado para generar el entorno en el que se trabajaran los TRIGGERS.

```
create table clientes(
    rut VARCHAR,
    nombre VARCHAR
    edad NUMERIC
    sueldo NUMERIC
    constraint clientes_pkey PRIMARY KEY (rut)
);
create table beneficio(
    id int primary key
    rut_clientes VARCHAR
);

```

Con la intención de poner a prueba esta herramienta, se creo un "Trigger" que simplifica el traspaso de la tabla personas a la de beneficiarios. La condición para esto ocurra es cumplir una edad mayor a 65 años y un valor de sueldo menor o igual a 500000, de esta forma al ingresar un nuevo cliente se hace una revisión de estos valores, si cumple, automáticamente sus datos pasan a la tabla beneficio.

El Trigger fue elaborado en torno a una función o procedimiento, donde se almacenan los datos en una estructura de tipo cursor la cual servirá para guardar todos los resultados obtenidos tras realizar la consulta. De esta forma se agrega el rut de la persona y se le da un id en la tabla beneficio, el id solamente sería la cantidad de filas más 1 y así sucesivamente en caso de que se agreguen más clientes a la tabla que cumplan con los requerimientos del "Trigger".

A continuación se puede observar el script donde se efectúa esta operación.

```
tablas_aux=# create or replace function Verify() returns Trigger
tablas_aux-# as
tablas_aux-# $BODY$
tablas_aux$# declare
tablas_aux$#     pana clientes%rowtype;
tablas_aux$# n1 int;
tablas_aux$# registro Record;
tablas_aux$# Cur_gente Cursor for select * from beneficio;
tablas_aux$# begin
tablas_aux$# n1 := 0;
tablas_aux$# for registro in Cur_gente loop
tablas_aux$#

tablas_aux$# n1 = n1 + 1;
tablas_aux$# end loop;
tablas_aux$#
tablas_aux$#     select rut from clientes into pana
tablas_aux$#     where clientes.edad>65 and clientes.sueldo<=500000;
tablas_aux$# --new.rut_cliente = pana;
tablas_aux$# --new.id = n1+1;
tablas_aux$#     insert into beneficio(id,rut_cliente) values(n1+1,pana);
tablas_aux$# return new;
tablas_aux$# end;
tablas_aux$# $BODY$
tablas_aux-# language plpgsql;
CREATE FUNCTION
tablas_aux=# CREATE TRIGGER Asigned
tablas_aux-#      AFTER INSERT ON clientes
tablas_aux-#      FOR EACH ROW
tablas_aux-#      EXECUTE PROCEDURE Verify();
CREATE TRIGGER
tablas_aux=# insert into clientes(rut,nombre,edad,sueldo) values('8','ElPepe',72,250000);
INSERT 0 1
tablas_aux=# select * from beneficio;
 id | rut_cliente
----+-------------
  1 | (8,,,)
```

Figura 10: Comprobación del TRIGGER

Como se puede observar, al realizar una inserción en la tabla beneficio también se registraron valores en la tabla beneficio, por lo que la función de TRIGGER fue ejecutada.

4. Análisis de resultados

Al utilizar las herramientas de postgres para el ingreso o actualización de nulos en las variables del apartado "NOT NULL" del presente informe (3.1), es posible observar el comportamiento de la base de datos al ser sometida a consultas erróneas, como puede ser una inserción o actualización datos con el parámetro "Null". La base de datos responde ante esto con un mensaje de error, puesto que se intentó violentar la restricción del campo asociado a la columna. Si la base de datos aceptara estos registros podría violentarse la consistencia de los datos debido a que para el usuario los valores dentro de la tabla son indudablemente distintos de nulos.

El apartado "B)", se basa en el uso de la restricción $CHECK$ la cual utiliza una expresión del tipo booleana para evaluar los valores antes de realizar una operación de inserción, esto se puede apreciar al momento de crear la tabla clientes e ingresar datos en esta. Es posible observar que surgieron errores al intentar violentar el campo asociado a la "constraint" $CHECK$ con valores erróneos a la condición procesada.

Por otra parte al aplicar un "Alter Table" respecto a la restricción $CHECK$ con la intención de operar para identificar los menores y mayores de edad en la tabla clientes, es posible notar que la transacción tampoco se puede ejecutar.

Con lo mencionado anteriormente la herramienta puede ser de gran ayuda al momento de controlar el flujo de datos en la base de datos, dando alternativas a las opciones tradicionales de condición y resguardando la consistencia de estos en la base de datos gracias a las respuestas del servidor.

Cabe destacar el caso de que las sentencias $CHECK$ colisionen al momento de establecer comunicación o alguna relación con otra entidad o tabla; Generando así una mayor complejidad escalable.

Cuando se da uso de llaves foráneas, muchas de las relaciones que pueden surgir pueden optimizar el tiempo de ejecución y así mismo permite una mejor cardinalidad de unión entre entidades. También reduce la posibilidad de la redundancia de filas en tablas con elementos duplicados, además al usar Foreign Key genera un comportamiento más lógico de la bases de datos al momento de agregar, actualizar o eliminar filas, tal es el caso que se vio al momento de aplicarla con clientes; como clientes es la tabla principal, se debe de eliminar o actualizar en cascada, aunque lo correcto sería hacer el cambio o borrado desde la referencia en adelante (ver figura 6 Comprobación Caso 1, página 7).

Por otra parte la restricción "Assertion" tiene un efecto similar a un "debug", esto quiere decir que hace la revisión sobre alguna condición asociada a una query, la cual almacena una variable que cumpla con la limitación. Para el caso presente, se analiza que hayan o no personas beneficiarias. Dando así la posibilidad de que el *Assert* pueda usarse como clausula similar al *exception* en diferentes ámbitos y operaciones, ya que es un posible mensaje de error que entrega la razón de la no realización del proceso.

El uso de Trigger's en ciertos aspectos llega a ser muy provechoso, tal es el caso de realizar operaciones antes o después de la inserción de elementos en la tabla clientes (ver figura 10: Comprobación del TRIGGER, página 10). De esta forma puede trabajar también con consultas que relacionen a más de una entidad o tabla, agilizando procesos predeterminados al momento de ejecutar alguna sentencia como lo son el $INSERT$, el $UPDATE$ o el $DELETE$.

5. Conclusión

Es posible confirmar el éxito de la presente actividad,debido al cumplimiento de los casos y pruebas solicitadas, y a partir de esto se presentan las siguientes observaciones.

Para cargar la base de datos, la página Mockaroo otorga la facilidad de la creación de los datos, en este caso generando mil datos respectivos para ambas tablas, teniendo presente realizar un cambio al archivo en cuestión, pasando de un archivo "csv" a "txt". Porque como se vio en cátedra el formato "csv" corresponde a un tipo de archivo perteneciente al trabajar con Windows, pero en este caso en que se usa directamente Linux conviene usar un formato que no desperdicie recursos o más bien correspondiente a la actividad siendo este último ".txt" .

NotNull otorga la utilidad de impedir que se cree data nula dentro de la base de datos. Caso que se violente el caso de ingresar data nula, ha de entregar un error en pantalla. Esto vendría a ser muy útil al momento de ingresar elementos nuevos a las entidades sobre todo porque es elemental que el campo asociado a este atributo sea rellenado de forma correcta, tal es el caso aplicado en una página web con forms, etc.

La herramienta *Check* tiene utilidad al momento de generar restricciones para el ingreso de datos, teniendo como funcionalidad una condición "boolean", de esta forma se puede mejorar el control de los datos que van llenando y permite un monitoreo más eficiente.

ForeingKey tiene la función de agilizar el rendimiento y disminuir la redundancia en el orden de la base de datos, es decir, no han de existir datos duplicados, de esta forma se tiene un manejo mucho más controlado sobre la misma. Los problemas que se encuentran en este punto vienen dados por agregar, actualizar y eliminar, siendo este el último el que presenta mayores problemas. Tal es el caso de que, al momento de generar un eliminar, si se toma primero la tabla padre, ha de generar un error indicando que la tabla hija esta asociada al dato que se busca eliminar, esto presenta la disyuntiva de si se elige realizar una eliminación por cascada o por referencia.

Por otro lado, el caso de *Assertion* ha de ayudar en temas de "debugger", teniendo como funcionamiento un "Try" y "Exception", lo cual ha de indicarle al usuario de la base de datos que algo no se cumplió, en las condiciones preestablecidas, y termina entregando un mensaje de porqué no funciono.

En *Trigger* se aprecia la agilización de los procesos, tanto como puede ser antes o después de la inserción de los elementos que ocurre dentro de la base de datos. Cabe resaltar lo visto en clases, el uso de muchos triggers ha de empeorar el rendimiento de la respectiva base de datos, llegando a tener problemas tales como bajo rendimiento o perdidas de datos.

6. Bibliografía

- Para elaborar Trigger
  `https://www.postgresql.org/docs/9.1/sql-createtrigger.html`
  Página consultada el 10 de Junio del 2021.

- Trigger Procedures
  `https://www.postgresql.org/docs/9.2/plpgsql-trigger.html`
  Página consultada el 10 de Junio del 2021.

- Check constraint
  `https://www.geeksforgeeks.org/postgresql-check-constraint/`
  Página consultada el 12 de Junio del 2021.

- Assert Statement
  `https://www.postgresqltutorial.com/plpgsql-assert/`
  Página consultada el 10 de Junio

- PL/pgsql Foreign Key
  `https://www.postgresqltutorial.com/postgresql-foreign-key/`
  Página consultada el 10 de Junio