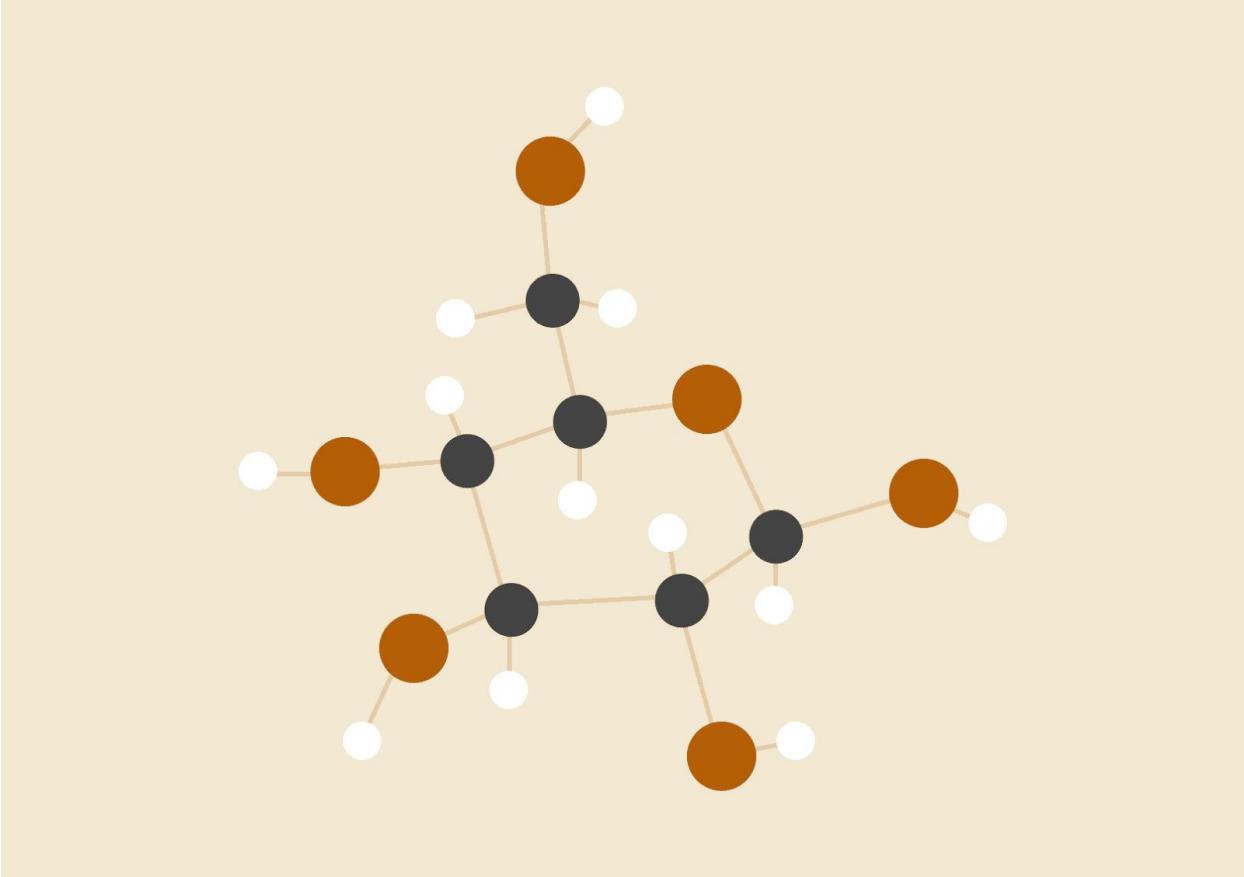
TRABAJO PRÁCTICO ESPECIAL

Díaz Gargiulo, Agustín. García Reinhold, Arturo.



Bases de Datos

25/06/2018

2.º TUDAI.

MOTIVACIÓN

A continuación presentaremos el análisis del trabajo práctico especial de la materia de Introducción a las Bases de Datos. Se nos pedía resolver un conjunto de controles y servicios sobre una base de datos que debíamos implementar, de acuerdo a un sistema de "Reservas de Departamentos Premium" y al script dado por la cátedra.

Para desarrollar la base de datos de este sistema tuvimos que tomar decisiones acerca de la implementación de cada una de las restricciones de integridad que se nos pedían. Estas restricciones debían implementarse tanto en el estándar de SQL como en la forma procedural que soporta PostgreSQL. Por lo que tuvimos que realizar un previo análisis del Diagrama de Entidad-Relación Extendido dado por la cátedra, cuyo contenido nos brindaba información acerca de las relaciones entre las tablas, sus restricciones de integridad y demás. También se nos proveyó del script de creación de las tablas del esquema, por lo que inicialmente tuvimos que introducir datos a cada una de las tablas según correspondía para el desarrollo de cada restricción y vista que se nos pedía de acuerdo a las reglas de negocio dadas y los servicios requeridos.

DESARROLLO DEL TRABAJO

Primera Parte, Ajuste del Esquema

Para completar la primera parte de la consiga, básicamente tuvimos que crear un script llamado G10_Creacion.sql donde realizamos las inserciones en cada una de las tablas, se nos pedía 5 inserciones en cada tabla, teniendo en cuenta que en algunas tablas esa cantidad era mayor. Además cabe destacar que creamos tablas adicionales al esquema brindado, las mismas fueron: "gr10_tipo_doc" la cual contiene el identificador del registro y una descripción del tipo de documento. Y "gr10_ciudad" la cual contiene el identificador de la ciudad y el nombre de la misma.

Segunda Parte, Elaboración de las Restricciones.

En esta parte del trabajo tuvimos que realizar los siguientes controles en el esquema de datos:

1) *Las fechas de las reservas deben ser consistentes.*

De acuerdo a la consigna tuvimos en cuenta primero que la fecha en la cual se está efectuando la reserva (de ahora en más fecha_reserva) de un departamento no podía ser mayor al momento en el cual se está llevando a cabo. Estas restricciones claramente eran a nivel atributo por lo que pudimos usar los CHECK para evaluar estas condiciones, siendo las restricciones declarativas las mismas en estos casos que soporta PostgreSQL.

```
alter table gr10_reserva add constraint gr10_reserva_check_fecha_reserva
check ( fecha_reserva <= now());
```

Segundo tuvimos en cuenta que la fecha_reserva debería ser menor o igual a la fecha_desde (fecha en la cual comienza la reserva).

```
alter table gr10_reserva add check (fecha_reserva <= fecha_desde);
```

Por último consideramos controlar que la fecha_desde debería ser menor a la fecha_hasta (fecha de finalización de la reserva).

```
alter table gr10_reserva add check ( fecha_desde < fecha_hasta);
```

2) *Que el detalle de las habitaciones sea consistente con el tipo de departamento.*

Para este punto debíamos tener en cuenta el uso de tres tablas: Departamento, Tipo_Dpto y Habitación, ya que había que corroborar por cada departamento que tenga la cantidad exacta de habitaciones que especificaba su tipo_dpto, y eso lo hacíamos consultando los identificadores de los departamentos en la tabla Habitacion. Teniendo en cuenta esto la restricción indicada es a nivel de base de datos, por lo que usamos una ASSERTION.

```
create assertion ck_cant_hab_dpto
check (not exists(select *
                  from gr10_departamento d
                  join gr10_tipo_dpto t on (d.id_tipo_depto=t.id_tipo_dpto)
                  where t.cant_habitaciones = (select count(*)
                                               from gr10_habitacion h
                                               where h.id_dpto= d.id_dpto)
       ))
```

Para la implementación en PostgreSQL primero evaluamos las situaciones en las cuales habría que aplicar está restricción, y llegamos a la conclusión que se debería proceder ante las siguientes situaciones:

Primero ante un cambio el id_tipo_depto en la tabla departamento ya que altera las cantidades de habitaciones permitidas en el mismo, segundo si se agrega o se actualiza (en este último caso un id_dpto) un registro en la tabla habitación, lógicamente esto cambia el número de habitaciones que puede poseer un departamento de un determinado tipo_dpto. Por último ante un cambio en las cant_habitaciones que permite un tipo_dpto, el cambio se puede realizar siempre y cuando no haya reservas en el departamento, para no permitir alterar esa consistencia.

El procedimiento que se ejecutaba en las primeras dos situaciones contaba la cantidad de habitaciones permitidas según el tipo_dpto del departamento reservado y contaba las cantidades de habitaciones que tenía efectivamente, siempre teniendo en cuenta el objeto que estoy

insertando, es decir en el caso de las habitaciones, el new.id_dpto. Si en la comparación entre ambos valores resulta haber más habitaciones que las permitidas no se efectúa la operación y se muestra el mensaje de error correspondiente.

La última situación llama a otro procedimiento, el mismo toma todos los departamentos que sean del tipo_dpto que se está actualizando y por cada uno se cuentan las habitaciones, si hay una inconsistencia, es decir hay más habitaciones de las permitidas, la operación no se lleva a cabo.

3) *Huéspedes en reserva (incluido el que la efectúa) no sean dueños de ese dpto.*

En este punto simplemente debíamos corroborar que el propietario no estuviese entre los huéspedes (incluido el que efectúa la reserva). Para la forma declarativa debíamos buscar lo que estuviese mal, es decir chequear que no exista un propietario entre los huéspedes de un departamento, sea en la tabla Huesped_Reserva o el huésped que realiza la reserva en la tabla Reserva, al tener que utilizar varias tablas de nuestro esquema la restricción es a nivel de base de datos, por ende el ASSERTION es la opción correcta.

```
create assertion ck_propietarios_huespedes
check(not exists(select *
                 from gr10_reserva r
                 join gr10_huesped_reserva h on (r.id_reserva=h.id_reserva)
                 where (h.tipo_doc, h.nro_doc) IN(select d.tipo_doc,d.nro_doc
                                                  from gr10_departamento d
                                                  where id_dpto!=d.id_dpto)
     )           )
```

Por el otro lado para la implementación de la versión de la restricción con soporte en PostgreSQL procedimos primero a definir los trigger, es decir las situaciones en las cuales interviene la restricción, las mismas fueron:

Primero una inserción en una reserva, o una actualización de la persona que la está efectuando, ya que no podía ser el dueño, la segunda situación es ante una inserción o alteración de la tabla donde se registran los huéspedes de la reserva, ya que el propietario tampoco puede estar entre ellos. Por último también hay que considerar que un departamento no puede cambiar de dueño si esa misma persona se encuentra alojada en él en ese momento.

Para la primera situación simplemente comprobamos que el propietario no sea la persona que efectúa la reserva, esto fijándonos en los datos del propietario en el departamento en cuestión. Para la segunda situación controlamos lo mismo, unimos los huéspedes, con sus reservas, de allí nos fijamos el departamento en cuestión y comparamos los datos del huésped y del propietario. Por último nos fijamos que al cambiar el propietario de un departamento el mismo no este en este momento como huésped de una reserva o tenga una reserva a su nombre. Lo mismo lo

conseguimos con dos variables booleanas, la primera comprobaba la existencia del propietario entre los huéspedes de las reservas de su departamento (en este momento) y la otra variable comprueba la existencia del propietario entre los datos de un huésped que efectúa una reserva. Si alguna de las dos resultaba verdadera se marcaba el error.

4) *La cantidad de huéspedes no puede superar los permitidos por la reserva.*

Para implementar esta restricción necesitábamos de varias tablas, sean las mismas: reserva, departamento, huesped_reserva y tipo_dpto. El ASSERTION era el indicado para usar en nuestra declaración, ya que aplicamos restricciones de más de una tabla, siendo el scope de la misma el esquema en general.

```
create assertion ck_cant_huespedes_permitidos
check(not exists(select *
                 from gr10_reserva r
                 join gr10_departamento d on (r.id_dpto= d.id_dpto)
                 join gr10_tipo_dpto t on (d.id_tipo_depto=
                      t.id_tipo_depto)
                 where t.cant_max_huespedes < (select count(*)
                                               from gr10_huesped_reserva h
                                               where r.id_reserva= h.id_reserva)
     )           )
```

Para la implementación de está restricción en PostgreSQL primero definimos las situaciones en las cuales se debía ejecutar la restricción, las mismas son:

Ante una inserción o un cambio en la tabla que alberga los registros de los huéspedes de una reserva, ya que eso cambia directamente el número de huéspedes que un departamento puede tener. La segunda situación es ante un cambio de departamento en la reserva, para ver si los huéspedes asignados a la misma “entran” en el nuevo departamento. Y por último ante un cambio de la cantidad de huéspedes que puede tener un tipo_dpto ya que había que controlar ante el cambio, si alguna reserva estaba vigente en ese momento (en el que se quiere efectuar el cambio) y si ese cambio provoca un exceso de gente en el departamento.

Para elaborar la función que se ejecutaba en las primeras dos situaciones comparamos la cantidad que huespedes que tenía esa reserva específica y la cantidad de huéspedes que tolera el tipo_dpto del departamento que se está reservando. Si la cantidad de huéspedes supera la capacidad se marcaba el error y no se procedía con la operación correspondiente.

Para la última situación tomamos todos los departamentos que tienen el nuevo id_dpto que se está actualizando y nos fijamos si había una reserva en ese momento la cual su número de huéspedes superen la nueva cantidad que permitimos.

Tercera Parte, Implementación de los Servicios.

Para el *primer servicio* se nos pedía que por cada departamento que estuviese en el sistema, demos el estado en la fecha determinada, siendo los estados posibles: “Ocupado” o “Libre”. Para esto utilizamos como recurso un procedimiento que dada una fecha como parámetro, almacena en una variable todos los departamentos que tengan alguna reserva para esa fecha. Esto se realiza con un left join ente departamento y reserva que me deja con todas las reservas sobre las cuales aplico el filtro del rango de fecha_desde y fecha_hasta dónde la fecha ingresada por parámetro se debía encontrar. Luego almacenados estos datos en una variable lo que resta es recorrerla y verificar si la reserva en el departamento es nula, en tal caso el estado del departamento sería libre, de lo contrario ocupado. Por cada una de esas iteraciones del recorrido íbamos generando la tabla resultante, que el procedimiento retorna. La misma está compuesta de un integer representando el id departamento y un varchar representando el estado del departamento en la fecha dada.// agregar justificacion del uso de procedimiento porque la funcion siempre se llama con distintas variables bla bla bla, la vista en cambio se usaria para algo más especifico bla bla bla.

En el caso del segundo servicio, bastante similar debíamos listar todos los departamentos libres en un rango de fechas provistas por el usuario y una ciudad de interés. Recurrimos a la misma opción, utilizamos un procedimiento que obtenía los datos del usuario por parámetro (fecha_desde, fecha_hasta y ciudad). Almacenamos en una variable los departamentos cuyas fechas de reserva no se encontraran en el rango provisto por el usuario y los departamentos mismos sean de la ciudad que se nos especificaba. Para ésta búsqueda realizamos un left join entre departamento y reservas, quedándonos con las reservas a las cuales les aplicamos el filtro por el cual las fecha_desde y fecha_hasta de la reserva no deben contener la fecha_desde y fecha_hasta que nos llegaba por parámetro. Una vez aplicado el filtro de fechas, le sumamos también que las ciudades de esos departamentos sean la que nos llegaba por parámetro. Finalmente agrupamos por departamento. También cabe destacar que al principio controlamos que el ingreso del usuario haya sido correcto, es decir que la fecha_desde que nos proveyó sea menor a la fecha_hasta en la consulta que desea hacer, de lo contrario le retornamos un error con la leyenda : “La fecha de inicio debe ser menor a la fecha de fin”.

Cuarta Parte, Definición de las Vistas.

Para está parte final del trabajo debíamos crear dos vistas a partir de los datos del esquema brindado, la primera vista requiere listar todos los departamentos del sistema junto con su recaudación de los últimos seis meses. Por ende creamos la vista denominada: *GR10_dpto_recaudacion()* la cual se generaba a partir de la siguiente búsqueda: primero nos quedamos con los pagos de cada reserva para cada departamento es decir un left join desde departamento con reserva para quedarnos con estas últimas y a su vez un left join con los pagos, para obtener de cada pago el importe correspondiente a la reserva y la fecha en la que se efectuó

el pago. Nosotros nos quedamos con la suma de todos estos importes, agrupados por departamento. Cabe destacar que para poder "recuperar" los pagos correspondientes a la fecha pedida usamos lo siguiente: *CURRENT_DATE - INTERVAL '6 months'* que nos da la fecha seis meses atrás a la fecha actual.

```
create or replace view GR10_dpto_recaudacion AS
    SELECT d.id_dpto AS "ID departamento", COALESCE(SUM(p.importe), 0) AS
    "recaudacion"
    FROM gr10_departamento d
    LEFT JOIN gr10_reserva r ON d.id_dpto = r.id_dpto
    LEFT JOIN gr10_pago p ON r.id_reserva = p.id_reserva
        AND p.fecha_pago > CURRENT_DATE - INTERVAL '6 months'
    GROUP BY d.id_dpto
    ORDER BY d.id_dpto;
```

La segunda vista pedía los departamentos ordenados por ciudad y por mejor rating. Para esto creamos la vista a partir de la siguiente búsqueda: primero obtenemos una vez más todos los departamentos, hacemos un left join con reservas y finalmente un left join con comentarios, quedándonos así con estos últimos, de los cuales nosotros pedimos el promedio del puntaje, para esto usamos la función AVG (y un cast a FLOAT para que me arroje un promedio más preciso) de PostgreSQL junto con el cast de COALESCE para los valores nulos. Estos resultados son agrupados primero por la ciudad a la que corresponde ese departamento y en segundo lugar por el promedio que arrojan los mismos.

```
create or replace view GR10_dpto_rating AS
       SELECT d.id_dpto, d.id_ciudad, COALESCE(AVG(CAST(c.estrellas AS FLOAT)), 0)
       FROM gr10_departamento d
       LEFT join gr10_reserva r ON d.id_dpto = r.id_dpto
       LEFT JOIN gr10_comentario c ON r.id_reserva = c.id_reserva
       GROUP BY d.id_dpto
       ORDER BY d.id_ciudad, AVG(c.estrellas);
```

CONCLUSIÓN

Para finalizar el trabajo podemos concluir que hemos desarrollado una base de datos consistente, que emula el sistema de una cadena de departamentos para alquilar. Hemos aplicado, a lo largo del seguimiento de la consigna algunas restricciones que se nos pedían dadas las reglas del negocio sobre las cuales trabajamos. Podemos enumerar muchas restricciones que podrían aplicarse al sistema de departamentos, pero escapamos a su vez de los dominios de la consigna.

Para cada una de las reglas del negocio a aplicar hemos desarrollado su código en estándar SQL a través de las sentencias CHECK y ASSERTION, aunque éstas no tienen soporte para PostgreSQL. También hemos desarrollado esas mismas restricciones con una sintaxis que soportara PostgreSQL a través del uso de TRIGGERS y FUNCIONES.

Con respecto a la definición de vistas no hemos de olvidar que profundizamos el concepto de una base de datos segura, en la cual hay una vista para cada tipo de usuario, podemos ver esta relación con un ejemplo fácil, suponer que la primera vista, la que asocia los departamentos con su rating podrían ser de interés para el futuro huésped o la persona interesada en alquilar, mientras que por otro lado la vista asociada a la recaudación de los departamentos podría ser de interés o bien para el administrador del sitio o para los propietarios en sí.

Finalmente hemos sido capaces de trabajar con las fechas de las reservas, extrayendo la información necesaria para la implementación de cada uno de los servicios.