Guía de ejercicios # 8:
Transacciones
Versión del 15/06/2010

Introducción a las bases de datos

UNQ

1. Sintaxis de comandos para manejo de stored procedures, y nota anterior

La nota anterior: cuando puedan elegir entre lockeo compartido y excusivo, elijan el compartido. Sólo usen lockeo exclusivo cuando un lockeo compartido no soluciona la cuestión.

1.1. SELECT ... INTO

Es

```
select atr1,atr2,...
from ...
where ...
group by ...
having ...
order by ...
into var1,var2,... ;
```

La consulta debe dar como resultado exactamente una fila (ni más ni menos) y las variables quedan asignadas de acuerdo al orden en que están en el `into`, a cada una se le asigna el valor del atributo que está en la misma posición.

P.ej.

```
declare nombreCandidato, apellidoCandidato varchar(50);

select nombre,apellido
from persona
where dni = 75840394
into nombreCandidato, apellidoCandidato;
```

2. Caja chica

El área de infraestructura de ACME Inc. quiere hacer un seguimiento de su operatoria de cajas chicas.

¿Cómo se manejan?

- Hay muchas cajas chicas, se las identifica por un número; cada caja está a cargo de un administrador.
- Una persona puede hacer un gasto adelantando el importe, y después lo rinde. Eso genera una fila en la tabla `gasto`.
- Una persona también puede hacer un retiro a rendir más adelante. Cuando hace el retiro se genera una fila en la tabla *retiro* donde se indica que está pendiente de ser rendido poniendo `estaRendido = 'N'`, y cuando la persona rinde el gasto se marca el retiro como rendido poniendo `estaRendido = 'S'` y se insertan los gastos correspondientes.
- El saldo de una caja se puede modificar por gastos, por retiros, y también por otras operaciones que no se registran en la BD; por eso hace falta conocer el saldo de cada caja explícitamente.

Quieren incluir en la misma BD un sistema de seguimiento de los toners de impresora láser: cuando se compra un toner se inserta una fila en la tabla `toner`, cuando se instala se le asignan a esa fila la `fechaInstalacion` y el `codImpresora` correspondientes, y cuando se cambia por uno nuevo, al que se está sacando se le asigna su `fechaReemplazo`.

No hay ninguna relación entre toners y cajas chicas, salvo que la compra de algunos toners (no necesariamente de todos) se hace mediante un gasto de caja chica.

El esquema completo es

```
caja <numCaja, nomAdmistrador, saldo>
retiro <numCaja, fecha, nomResp, importe, estaRendido>
gasto <numGasto, numCaja, fecha, nomResp, concepto, importe>
toner <modelo, numSerie, codImpresora, fechaCompra, fechaInstalacion,
fechaReemplazo>
modeloToner <modelo, precioAproximado>
```

La pk de `gasto` es un número que se asigna automáticamente para cada gasto.

1. Crear los siguientes *stored procedures*:

   a) `SPcomprartoner(numCaja,comprador,modelo,nroserie,precio)`: Registra el gasto para la caja que corresponde y lo agrega al stock (tabla de toner). Recordar restar del saldo de la caja. Como todavía no se sabe en qué impresora se va a instalar el toner, ese dato debe ir en null. Tomar la fecha del sistema `date(sysdate)` como fecha de compra.

   b) `SPinstalartoner(modelo,numSerie,codImpresora)`: Registra que se instala un toner en una impresora; se debe registrar que el toner que actualmente tiene la impresora (si es que tiene alguno) fue reemplazado. Tomar la fecha del sistema como fecha de instalación.

c) `SPretirardecaja(caja,responsable,importe)`: Retira un importe de una caja. Debe actualizar el saldo de la caja y agregar un elemento en la tabla retiro. Tomar la fecha del sistema como fecha de retiro.

2. Tenemos estos dos stored procedures (agregar los `declare` de las variables correspondientes)

```
SPtransferir(cajaDesde, cajaHacia, cuanto)
begin
  select saldo from caja where numCaja = cajaHacia into shant;
  select saldo - cuanto from caja where numCaja = cajaDesde into sd;
  select saldo + cuanto from caja where numCaja = cajaHacia into sh;
  update caja set saldo = sh where numCaja = cajaHacia;
  if (sd >= 0) then
    update caja set saldo = sd where numCaja = cajaDesde;
  else
    update caja set saldo = shant where numCaja = cajaHacia;
  end if;
end

SPgasto(laCaja, elResp, elConcepto, elImporte)
begin
  select saldo from caja where numCaja = laCaja into sant;
  if (sant >= elImporte) then
    update caja set saldo = saldo - elImporte where numCaja = laCaja;
    select max(numGasto) + 1 from gasto into nroNuevoGasto;
    insert into gasto(numGasto,numCaja,fecha,nomResp,concepto,importe)
    values (nroNuevoGasto, laCaja, date(sysdate()),
            elResp, elConcepto, elImporte);
  end if;
end
```

Observar que no se están delimitando transacciones.

La caja 1 tiene un saldo de 1000 pesos, la caja 2 tiene un saldo de 2000 pesos. El usuario 1 quiere pasar 200 pesos de la caja 1 a la 2.

a) Indicar un intercalado de la transacción del usuario 1 con otra ejecución concurrente de `SPtransferir` que haga que el saldo de la caja 2, después de haber terminado los dos, quede mal.

b) Lo mismo, pero con una ejecución concurrente de `SPgasto`.

c) Indicar un intercalado entre una ejecución del usuario 1 y una de un gasto en el que se gaste una plata que no se tiene, quedando probablemente una caja con saldo negativo.

3. Vayamos a un caso bien concreto: el usuario 1 quiere pasar 300 pesos de la caja 1 a la caja 2, y al mismo tiempo el usuario 2 quiere pasar 500 pesos de la caja 2 a la caja 1. Antes de las transferencias, la caja 1 tiene 1000 pesos de saldo y la caja 2 tiene 2000.
Ambos usan el stored procedure `SPtransferir`. Seguimos trabajando sin

delimitar transacciones.
Queda claro que después de la ejecución de las transaciones, si sumo el saldo de las cajas 1 y 2, debe dar 3000; el saldo de la caja 1 debe ser 1200, y el de la caja 2 debe ser 1800.

Encontrar intercalados que

a) terminen con un saldo total (caja 1 + caja 2) = 2500.

b) terminen con un saldo total (caja 1 + caja 2) = 3300.

c) terminen con un saldo total (caja 1 + caja 2) = 2700.

4. Ahora a los dos stored procedures les delimito transacción, o sea, se les pone un `start transaction` al principio y un `commit` al final.

   Pasemos a nivel de aislamiento `READ COMMITTED`.

   a) En SPtransferir hay una parte que se puede cambiar, ahora que nos manejamos con transacciones. Hacer los cambios correspondientes. Ayuda: uno de los `select` ya no va a hacer falta.

   b) Los casos de intercalado que generaban un saldo incorrecto final en la caja 2, ahora que pusimos transacciones, ¿se arreglaron?.

   c) Encontrar un intercalado entre una ejecución de `SPtransferir` y `SPgasto` que puede hacerse sin transacciones y no con transacciones, porque una transacción queda trabada.

   d) Encontrar un intercalado de dos ejecuciones de `SPtransferir` que resulte en un deadlock.

   e) Cambiar `SPtransferir` para que el saldo de las cajas quede siempre bien. No se pueden agregar lockeos en este punto, lo que se puede hacer es cambiar los `select` y los `update`. Sí puede pasar que permita hacer transacciones que no debería (quedando algún saldo negativo) pero no que las cuentas se hagan mal.

5. Pasemos a nivel de aislamiento `REPEATABLE READ`.

   a) Indicar un intercalado usando estos dos stored procedures (dos transacciones de uno, o una de cada) en el cual el saldo de una de las cajas quede negativo.

   b) Agregar los lockeos que hagan falta para garantizar que eso no pase en ningún caso.

   c) Indicar un caso de deadlock entre dos ejecuciones de `SPtransferir`.

   d) Indicar por qué una ejecución de `SPtransferir` y una de `SPsaldo` no pueden entrar en deadlock.

3. Stock

Se parte de este esquema de BD

Producto

nomProd	stkMinimo
'Anaflex'	1
'Jorgito'	83
'Menganol'	30
'Motik'	28
'Pepin'	8
'Perek'	3
'Susik'	20

Stock

nomProd	stkActual
'Anaflex'	10
'Menganol'	38
'Motik'	24
'Perek'	10
'Susik'	32

El script de creación de esta instancia es `stock.sql`.
Cada ítem parte de las tablas como están indicadas.

1. hago la siguiente secuencia de operaciones

```
1. START TRANSACTION;
2. UPDATE stock SET stkActual = stkActual + 7
   WHERE nomProd = 'Perek';
3. INSERT INTO stock (nomProd, stkActual) VALUES ('Jorgito',4);
4. DELETE FROM stock WHERE nomProd = 'Anaflex';
5. COMMIT;
6. SELECT * FROM stock WHERE stkActual < 15;
```

   a) ¿ Cuál es el resultado del último `SELECT` ?

   b) Lo mismo, cambiando el paso 5 por `ROLLBACK`

2. El usuario 1 hace la siguiente secuenca de operaciones

```
1. START TRANSACTION;
2. SELECT * FROM stock;
3. INSERT INTO stock (nomProd, stkActual) VALUES ('Jorgito',4);
4. UPDATE stock SET stkActual = stkActual + 7
   WHERE nomProd = 'Perek';
5. COMMIT;
```

   simultáneamente, el usuario 2 abre una transacción y hace las siguientes operaciones

```
SELECT stkActual FROM stock WHERE nomProd = 'Perek';
SELECT sum(stkActual) FROM stock;
```

   todo antes que el usuario 1 abra su transacción. Después vuelve a hacer estas operaciones cuatro veces más, en los siguientes momentos

I. entre los pasos 2 y 3 del usuario 1.

II. entre los pasos 4 y 5 del usuario 1.

III. después que el usuario 1 hace COMMIT.

IV. después de hacer COMMIT el usuario 2, en una transacción distinta.

Indicar, para cada uno de los niveles de aislamiento READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ, SERIALIZABLE de acuerdo a cómo los define MySQL, qué resultados ve el usuario 2 en cada una de las cuatro repeticiones del par de SELECT.

En algún caso, alguno de las dos transacciones va a tener que esperar que la otra termine, por lo tanto no pueden ocurrir los 4 casos. Indicar de qué caso se trata y qué pasa.

4. Venta de entradas

Hay que manejar la venta de entrada a recitales en una sala. Hay recitales en los que la entrada se vende con ubicación, y otros que son sin ubicación. Para los primeros no vale vender dos veces la misma ubicación para el mismo recital, y para los segundos no se puede superar la capacidad de la sala.

Las tablas son

```
sala <numSala, capacidadSinSillas>
funcion <idFuncion, numSala, fechaHora, banda>
entradaConUbicacion <numeroEntrada, idFuncion, filaAsiento,
numeroAsiento, fechaVenta>
entradaSinUbicacion <numeroEntrada, idFuncion, fechaVenta>
```

El script de creación sin `entradaConUbicacion` está en `funciones.txt`. La tabla que falta, es porque nos olvidamos :S.

Para una función, son todas con ubicación o todas sin ubicación. Para las funciones con ubicación, cuando se crea la función se crean también todas las filas correspondientes de `entradaConUbicacion`, una para cada ubicación en la sala, todas con `fechaVenta = null`.

Las transacciones para la venta de una entrada son las siguientes.

Para entradas con ubicación

```
venderEntradaConUbicacion(elIdFuncion,laFila,elNumero)
  declare laFecha date;

  start transaction;

  select fechaVenta
  from entradaConUbicacion
  where idFuncion = elIdFuncion
    and filaAsiento = laFila and numeroAsiento = elNumero
  into laFecha;

  if (laFecha is not null) then
    update entradaConUbicacion
    set fechaVenta = date(sysdate())
    where idFuncion = elIdFuncion
      and filaAsiento = laFila and numeroAsiento = elNumero;
  end if;

  commit;
end
```

Para entradas sin ubicación

```
venderEntradaSinUbicacion(elIdFuncion)
  declar cant,capac date;

  start transaction;
```

```
  select capacidadSinSillas
  from funcion natural join sala
  where idFuncion = elIdFuncion
  into capac;
  select count(*) from entradaSinUbicacion
   where idFuncion = elIdFuncion into cant;

  if (cant < capac) then
    insert into entradaSinUbicacion (numeroEntrada,idFuncion,fechaVenta)
    values (xyz, elIdFuncion, date(sysdate()))
  end if;
  commit;
end
```

Estamos en nivel de aislamiento `READ COMMITTED`
Se pide

1. Para cada una de las transacciones, encontrar un intercalado entre dos ejecuciones de la misma (o sea, dos con ubicación por un lado, dos sin ubicación por el otro) que genere una anomalía de concurrencia. Inventar los datos que hagan falta.

2. Para el caso de con ubicación, si cambio el nivel de aislamiento a `SERIALIZABLE` de acuerdo a la definición de MySQL, ¿se arregla el problema?

3. Para cada caso, agregar lockeos que solucionen la anomalía. Para el caso sin ubicación, suponer que estamos trabajando con un motor de BD que no lockea condición sino filas. Vale agregar un `SELECT` adicional sólo para lockear.

4. Supongamos que queremos mantener un contador de cantidad de entradas vendidas por día. Entonces ponemos en ambas transacciones la siguiente sentencia inmediatamente después del `start transaction`

   ```
   update totalesPorDia set cant = cant + 1 where fecha = date(sysdate());
   ```

   Esto arregla el problema de concurrencia sin necesidad de lockeos, explicar por qué.

   Por otro lado genera casos en los que una transacción debe esperar que otra termine, mientras que con la solución de los lockeos eso no pasa, pueden ir en simultáneo. Mostrar un caso que se comporte de esta manera, inventando los datos que sean necesarios.

5. Micros

Tenemos que armar una BD y una aplicación para una empresa de micros, para que carguen los servicios que van a hacer. Tenemos choferes, micros y servicios. Un chofer no puede hacer más de dos servicios al día.

Las tablas son

```
chofer <legajo, nombre>
micro <patente, peso>
servicio <idServicio, legajoChofer, patenteMicro, fecha,
origen, destino>
```

Y la transacción para crear un servicio es

```
crearServicio(elLegajo,laPatente,laFecha)
  declare cantServiciosDelChoferEnElDia int;
  start transaction;
  select count(*) from servicio
  where legajoChofer = elLegajo and patenteMicro = laPatente and fecha = laFecha
  into cantServiciosDelChoferEnElDia;
  if (cantServiciosDelChoferEnElDia < 2) then
    insert into servicio ... ;
  end if;
  commit;
end
```

Estamos en nivel de aislamiento `READ COMMITTED`
Se pide

1. Encontrar un intercalado entre dos ejecuciones de `crearServicio` que genere una anomalía de concurrencia, inventando los datos que hagan falta.

2. Solucionar con un lockeo, suponiendo que el motor de BD con el que trabajamos no lockea condición sino filas. Ayuda: vale agregar un `SELECT`.

6. Ingresos y egresos de stock

Tenemos una BD que maneja stock, para lo cual tengo estas tablas

```
producto <codProducto, nomProducto>
deposito <codDeposito, nomDeposito>
stock <codProducto, codDeposito, cantidad>
```

Hay dos transacciones: despachar productos, y recibir productos. Son así:

```
despacharProductos(prod1,cant1,prod2,cant2,dep)
  declare stock1, stock2 int;
  start transaction;

  select cantidad from stock
   where codProducto = prod1 and codDeposito = dep
    into stock1;
  select cantidad from stock
   where codProducto = prod2 and codDeposito = dep
    into stock2;

  if (stock1 >= cant1 and stock2 >= cant2) then
    update stock set cantidad = stock1 - cant1
     where codProducto = prod1 and codDeposito = dep;
    update stock set cantidad = stock2 - cant2
     where codProducto = prod2 and codDeposito = dep;
  end if;
  commit;
end

recibirProductos(prod1,cant1,prod2,cant2,dep)
  declare stock1, stock2 int;
  start transaction;

  select cantidad from stock
   where codProducto = prod1 and codDeposito = dep
    into stock1;
  select cantidad from stock
   where codProducto = prod2 and codDeposito = dep
    into stock2;

  update stock set cantidad = stock1 + cant1
   where codProducto = prod1 and codDeposito = dep;
  update stock set cantidad = stock2 + cant2
   where codProducto = prod2 and codDeposito = dep;
  commit;
end
```

Tenemos dos usuarios, uno quiere despachar y otro recibir los productos pr1 y pr2 del depósito d1. El stock en ese depósito es de 100 unidades de cada producto. El usuario 1 quiere despachar 10 unidades de pr1 y 20 de pr2. El usuario 2 quiere recibir 50 unidades de pr1 y 30 de pr2. Está claro que después

de estas transacciones, el stock va a quedar en 140 para pr1 y 110 para pr2, 250 en total.

Estudiemos varios escenarios, en nivel de aislamiento `READ COMMITTED`

1. Encontrar intercalados con los que

   - el stock pr1 + pr2 dé 250, o sea el valor correcto
   - el stock pr1 + pr2 dé 280
   - se entre en deadlock

   ayuda: influye el orden en que entran los productos en los parámetros de los stored procedures.

2. Encontrar un intercalado sacando los `start transaction` y `commit` en el que el stock pr1 + pr2 dé 220.

3. Si cambio en los `update` las variables `stock1` y `stock2` por el atributo `cantidad`, no pueden quedar mal los valores, aunque no se solucionan los deadlock. ¿Por qué?

   También puede haber otra anomalía de concurrencia, indicar cuál y presentar un ejemplo.

4. Encontrar una forma de, modificando los `select` y lockeando, se evite tanto la otra anomalía de concurrencia **como la posibilidad de deadlocks**.

7. Clientes de operaciones bancarias

Tenemos un sistema de registro de operaciones bancarias. Para hacer una operación, el cliente tiene que estar habilitado. Entonces tenemos esta transacción

```
registrarOperacion(elCodCliente,elImporte)
  declare statusCliente varchar(2);
  start transaction;
  select xstatus from cliente where codCliente = elCodCliente into statusCliente;
  if (statusCliente = 'OK') then
    insert into operacion ...
  end if;
  commit;
end
```

Estamos en modo de aislamiento `REPEATABLE READ`
Se pide

1. Indicar dos ejemplos de operaciones simultáneas con la que se indica que podrían provocar anomalías de concurrencia. Ayuda: una es un `update`, la otra no.

2. Si cambio a `SERIALIZABLE` con la definición de MySQL, el problema se arregla. Explicar por qué.

8. Apuestas de caballos

Lo que sigue es parte del esquema de una BD para una empresa que recibe apuestas a caballos en distintos hipódromos

```
caballo<nomCaballo, importeMaximoApuestasPorCarrera>
carrera<nomCarrera, hipodromo, fecha, hora>
participanteEnCarrera<nomCarrera, nomCaballo, numero>
apuesta<nomCarrera, nomCaballo, nomApostador, importe, idCajero>
```

A fin de evitar especulaciones, se define para cada caballo un tope en el importe total de apuestas que puede recibir en cada carrera en la que participa, ese es el `importeMaximoApuestasPorCarrera`.

P.ej. si para el caballo Bucéfalo se define un tope de 1000 pesos por carrera, participa en la carrera Premio Limón, y 3 personas le apostaron $300 cada una a Bucéfalo en el Premio Limón, entonces sólo se pueden aceptar apuestas adicionales por un total de hasta $100 pesos a Bucéfalo en esa carrera; si Bucéfalo también corre el Premio Frutilla y nadie hizo apuestas para esta carrera, entonces se puede apostar a Bucéfalo para el Premio Frutilla hasta los $1000 de su tope por carrera.

Se codifica el siguiente stored procedure muy sencillo para agregar una apuesta

```
apostar(laCarrera varchar(45), elCaballo varchar(45), elApostador varchar(45),
        elImporte decimal(12,2), elCajero int)
begin
  insert into apuesta(nomCarrera, nomCaballo, nomApostador, importe, idCajero)
         values(laCarrera, elCaballo, elApostador, elImporte, elCajero);
end
```

Dada esta situación

1. Como está el sp, no hay ninguna validación que impida superar el importe máximo de apuestas por carrera de un caballo. Agregar la validación al sp apostar. Por ahora no hacer ningún lockeo.

2. Si la validación que hicieron es correcta, no se podrá superar el límite de apuestas ... salvo anomalías de concurrencia. Mostrar un schedule que provoque que se supere el límite, inventando los datos que hagan falta.

3. Agregar lo que haga falta para evitar las anomalías de concurrencia detectadas, suponiendo que el motor de BD que estamos usando lockea por conjunto de filas y no por condición. Mostrar que el schedule que provocaba la anomalía ya no es posible.

4. La solución que diste en el punto anterior ¿permite cargar dos apuestas concurrentes al mismo caballo en diferentes carreras? Indicar si sí o si no justificando, y en caso de que no se pueda, modificar el sp para permitir apuestas concurrentes al mismo caballo en diferentes carreras, y también apuestas concurrentes a distintos caballos en la misma carrera, aunque no (claro) apuestas concurrentes al mismo caballo en la misma carrera.

9. Pañol de herramientas

Lo que sigue es parte del esquema de la BD que se armó para una aplicación de reserva de herramientas en un pañol. Vamos a usar MySQL.
`herramienta<idHerramienta, tipo, marca, nomPersonaQueLaReservo>`
`persona<nomPersona, fnac>`

Si una herramienta no está reservada, el `nomPersonaQueLaReservo` está en `null`.

Las herramientas siempre se reservan de a pares (p.ej. una lijadora y una agujereadora, un martillo y un punzón, un martillo y una agujereadora, etc.), por esto se armó el siguiente stored procedure

```
reservar(idHerr1 int, idHerr2 int, nomPersonaQueQuiereReservar varchar(45))
begin
  start transaction;
  declare nomPersonaRes1, nomPersonaRes2 varchar(45);
  select nomPersonaQueLaReservo from herramienta where idHerramienta = idHerr1
    into nomPersonaRes1;
  select nomPersonaQueLaReservo from herramienta where idHerramienta = idHerr2
    into nomPersonaRes2;
  if (nomPersonaRes1 is null and nomPersonaRes2 is null) then
    update herramienta set nomPersonaQueLaReservo = nomPersonaQueQuiereReservar
     where idHerramienta = idHerr1;
    update herramienta set nomPersonaQueLaReservo = nomPersonaQueQuiereReservar
     where idHerramienta = idHerr2;
  end if;
  commit;
end
```

A partir de esta situación

1. Indicar un intercalado en el que el valor que lee el primer `select` para alguna de las transacciones incluidas es distinto según si elegimos nivel de aislamiento `READ COMMITTED` o `REPEATABLE READ`, inventando los datos que hagan falta.

2. Hay una posible anomalía de concurrencia. Indicar de qué se trata, mostrarla mediante un intercalado inventando los datos que hagan falta, e indicar una forma sencilla de evitarla modificando el stored procedure.

3. Ya sea con el stored procedure original o con el modificado, pueden darse deadlocks. Armar un intercalado que entre en deadlock en alguna de las dos versiones (original o modificada), indicando cuál se está tomando.

4. Cambiar el sp para evitar la posibilidad de deadlock. **Ayuda** considerar que si una sentencia SQL debe lockear varias filas, lockea todas o ninguna.

10. Donaciones

Una agencia gubernamental maneja la asignación de fondos a distintas ONGs a los que los contribuyentes al Impuesto a las Ganancias pueden hacer donaciones. Lo que sigue es parte del esquema definido para la BD que va a manejar esta información

```
donacion<nroDonacion, idOng, cuitDonante, fecha, importe>
donante<cuitDonante, razonSocial, limiteCantDonacionesPorDia>
ong<idOng, saldo>
```

Algunos donantes tienen limitada la cantidad de donaciones que pueden hacer en un mismo día, eso lo pidió la gente que controla el lavado de dinero. A esos donantes se le pone el valor del tope en `limiteCantDonacionesPorDia`, al resto se le pone `null` como valor para ese atributo.

Tenemos este stored procedure para registrar una donación

```
donar(elIdOng int, elCuitDonante varchar(20), laFecha date, elImporte decimal(12,2))
begin
  start transaction;

  declare cantDonacionesDeHoy int;
  declare cantMaxDonaciones decimal(12,2);
  declare saldoAnterior decimal(12,2);

  select limiteCantDonacionesPorDia from donante
  where cuitDonante = elCuitDonante
  into cantMaxDonaciones;
  select count(*) from donacion
  where cuitDonante = elCuitDonante and fecha = laFecha
  into cantDonacionesDeHoy;
  select saldo from ong where idOng = elIdOng into saldoAnterior;

  if (cantMaxDonaciones is not null) and (cantDonacionesDeHoy < cantMaxDonaciones)
  then
    insert into donacion(nroDonacion, idOng, cuitDonante, fecha, importe)
           values(..., elIdOng, elCuitDonante, laFecha, elImporte);
    update ong set saldo = saldoAnterior + elImporte;
  end if;

  commit;
end
```

Un programador avispado se da cuenta de una anomalía de concurrencia que puede tener este stored procedure, entonces le cambia el código por lo que sigue

```
donar(elIdOng int, elCuitDonante varchar(20), laFecha date, elImporte decimal(12,2))
begin
  start transaction;

  declare cantDonacionesDeHoy int;
  declare cantMaxDonaciones decimal(12,2);

  select limiteCantDonacionesPorDia from donante
  where cuitDonante = elCuitDonante
  into cantMaxDonaciones;
  select count(*) from donacion
  where cuitDonante = elCuitDonante and fecha = laFecha
  into cantDonacionesDeHoy;

  if (cantMaxDonaciones is not null) and (cantDonacionesDeHoy < cantMaxDonaciones)
  then
    insert into donacion(nroDonacion, idOng, cuitDonante, fecha, importe)
           values(..., elIdOng, elCuitDonante, laFecha, elImporte);
    update ong set saldo = saldo + elImporte;
  end if;

  commit;
end
```

(para ver rápidamente la diferencia, mirar el `update` en las dos versiones)

Pero ... ¡ay! el programador no fue lo suficientemente avispado, hay otra anomalía de concurrencia que se le pasó, y puede ocurrir con cualquiera de las dos versiones.

A partir de la situación descripta, se pide

1. Indicar cuál es la anomalía que aparece en la primer versión y no en la segunda, dar un schedule de la primer versión en donde se manifieste el problema.

2. Indicar cuál es la anomalía que se presenta en las dos versiones, dar un schedule de la segunda versión en la que se manifieste el problema, hacer las modificaciones necesarias para solucionarla. Justificar por qué el agregado no produce ningún deadlock.