

Titulación: Grado en Ingeniería Informática y Sistemas de Información
Curso: 2019-2020. Convocatoria Ordinaria de Junio
Asignatura: Bases de Datos Avanzadas – Laboratorio

Practica 3: Seguridad, Usuarios y Transacciones.

ALUMNO 1:

Nombre y Apellidos: Javier Martín Gómez _____

DNI: 47231977M _____

ALUMNO 2:

Nombre y Apellidos: Alberto González Martínez _____

DNI: : 09072311F _____

Fecha: 7-6-2020 _____

Profesor _____
Gutiérrez _____

Responsable: _____ **Oscar**

Mediante la entrega de este fichero los alumnos aseguran que cumplen con la normativa de autoría de trabajos de la Universidad de Alcalá, y declaran éste como un trabajo original y propio.

En caso de ser detectada copia, se puntuará TODA la asignatura como Suspense – Cero.

Plazos

Tarea online: Semana 13 de Abril, Semana 20 de Abril y semana 27 de Abril.

Entrega de práctica: **Día 18 de Mayo (provisional).** Aula Virtual

Documento a entregar: Este mismo fichero con las respuestas a las cuestiones planteadas, con el código SQL utilizado en cada uno de los aparatos. Si se entrega en formato electrónico se entregará en un ZIP comprimido: **DNI ' sdelosAlumnos_PECL3.zip**

AMBOS ALUMNOS DEBEN ENTREGAR EL FICHERO EN LA PLATAFORMA.

Introducción

El contenido de esta práctica versa sobre el manejo de las transacciones en sistemas de bases de datos, así como el control de la concurrencia y la recuperación de la base de datos frente a una caída del sistema. Las transacciones se definen como una unidad lógica de procesamiento compuesta por una serie de operaciones simples que se ejecutan como una sola operación. Entre las etiquetas BEGIN y COMMIT del lenguaje SQL se insertan las operaciones simples a realizar en una transacción. La sentencia ROLLBACK sirve para deshacer todos los cambios involucrados en una transacción y devolver a la base de datos al estado consistente en el que estaba antes de procesar la transacción. También se verá el registro diario o registro histórico del sistema de la base de datos (en PostgreSQL se denomina WAL: Write Ahead Login) donde se reflejan todas las operaciones sobre la base de datos y que sirve para recuperar ésta a un estado consistente si se produjera un error lógico o de hardware. La versión de postgres a utilizar deberá ser la versión 12.

Actividades y Cuestiones

En esta parte la base de datos **TIENDA** deberá de ser nueva y no contener datos. Además, consta de 5 actividades:

- Conceptos generales.
- Manejo de transacciones.
- Concurrencia.
- Registro histórico.
- Backup y Recuperación

Cuestión 1: Arrancar el servidor Postgres si no está y determinar si se encuentra activo el diario del sistema. Si no está activo, activarlo. Determinar cuál es el directorio y el archivo/s donde se guarda el diario. ¿Cuál es su tamaño? Al abrir el archivo con un editor de textos, ¿se puede deducir algo de lo que guarda el archivo?

Partimos de la base de datos Tienda vacía (borrando los datos de la práctica anterior).

Ahora, arrancamos el servidor. El diario del sistema (WAL, Write-ahead logging) se encuentra activo por defecto en PostgreSQL. Se encuentra en el directorio de la ruta "C:\Program Files\PostgreSQL\12\data\pg_wal". El archivo donde se encuentra el diario sería el más reciente (en nuestro caso 0000000100000009000000Do), aunque podría haber más de un archivo.

Como se puede comprobar en la siguiente captura, los archivos son de 16384 KB (16 MB)

000000010000000A00000000	03/04/2020 20:53	Archivo	16.384 KB
000000010000000A00000001	03/04/2020 20:53	Archivo	16.384 KB
0000000100000009000000D0	29/04/2020 19:59	Archivo	16.384 KB
0000000100000009000000D1	03/04/2020 20:50	Archivo	16.384 KB
0000000100000009000000D2	03/04/2020 20:50	Archivo	16.384 KB
0000000100000009000000D3	03/04/2020 20:50	Archivo	16.384 KB
0000000100000009000000D4	03/04/2020 20:50	Archivo	16.384 KB
0000000100000009000000D5	03/04/2020 20:50	Archivo	16.384 KB
0000000100000009000000D6	03/04/2020 20:50	Archivo	16.384 KB
0000000100000009000000D7	03/04/2020 20:50	Archivo	16.384 KB
0000000100000009000000D8	03/04/2020 20:50	Archivo	16.384 KB
0000000100000009000000D9	03/04/2020 20:50	Archivo	16.384 KB
0000000100000009000000DA	03/04/2020 20:50	Archivo	16.384 KB
0000000100000009000000DB	03/04/2020 20:50	Archivo	16.384 KB
0000000100000009000000DC	03/04/2020 20:50	Archivo	16.384 KB
0000000100000009000000DD	03/04/2020 20:50	Archivo	16.384 KB
0000000100000009000000DE	03/04/2020 20:50	Archivo	16.384 KB

Este sería el último archivo abierto con un editor de texto:

```

Ñ  - - - - -  ð  - - - - -  ð  q  - - - - -  ,  <  +  +  ;  '  %  %  $  )  -  (  °
,i-
  %b 0
7  ý  ð  ð  +b  ð  )b  ð  a  H  4  `kñð  ð  Eñ°x  ð  ð  %b  q
  ý  7  ð  ð  +b  ð  ð  H  4  °kñð  ð  'ðü™  ð  ð  %b  r
)  ý  7  ð  ð  )b  ð  ð  P  4  økñð
p|²€  ð  %b 0
7  ý  ð  ð  +b  ð  )b  ð  a,  H  4  @lñð  ð  $n
,  ð  ð  %b  q
  ý  7  ,  ð  ð  +b  ð  à  H  4  lñð  ð  e|0Š  ð  ð  %b  r
)  ý  7  ,  ð  ð  )b  ð  š  4  0lñð  °  YÖR=  ð  P  @  ð  ð  %b  +b  @  ðð  b1  ð
{|¹  ð  ð  %b  ð  ý  )b  pg_toast_25126  ð  ð  %  4  "mñð  p  c  *b
)b  ð  pt|  ð  ð  4  ð  ð
|Äv+  ð  ð  %b  ð  ý  +b  pg_toast_25126_index  ð  ð  P  4  hnñð  @  c
"  +b  ð  ð  pi  ð  ð
|WÄY  ð  ð  %b  ð  ý  !  !€  (  ýýý?  )b4  ~  ð  P  4  (oñð
%çR  ð  %b 0
7  ý  ð  ð  )b  ð  &b  if  H  4  xoñð  ð  ²ðN!  ð  ð  %b  q
  ý  7  f  ð  ð  )b  ð  ð  H  4  Èoñð  ð  N!  !  ð  ð  %b  r
)  ý  7  f  ð  ð  &b  ð  ~  *  4  |pñð  ð  |$|aý|  ð  %b  ,b  *  4  Xpñð  ð  -|,-ý  4  %b  ,b  È
|Ša@  š  ð  %b  ð  ý  !  ð  ýýý?  ,b  Tienda_pk  ð  @  4  .pñð  ð  ÄO,  !  ð  ð  %b  f
"  ,b  pi  ð  ð
  ý  ð  ,b  v  H  4  ^qñð  ð  W"  €  ð  ð  %b  g
  ý  ð  |@Tienda_pk  ~  ð  @  4  Èqñð  ð  „PÚ\  ð  ð  %b  ð
  ý  ð  ,b  4  -  4  |rñð
çs@
~  ð  %b  á  &  ý  ð  ýý  ,b  Id_tienda  ð  ýýýý  ð  ýýýýýýý|pi  ð
  ý  & 0  |@,b  Id_tienda  7  @  4  sñð  ð  S|ð  ð  ð  %b  c

  ý  & 0  ,b  ð  0  Ä  4  Hsñð
%/°i  "  ð  %b  2
  ý  ð  ýý  ,b  &b  ð  ð  ð  ð  h  ð  ð  ð  p  ð  ð  ð  p  ð  ð  ð  ð  h  ð  ð
  ý  ð  ð  &b  @  4  Ptñð  ð  ð  ð  ð  %b  w
  ý  ð  ,b  i  "  4  tñð
ÿða  ð  |ð$  ð  ð  %b  .

```

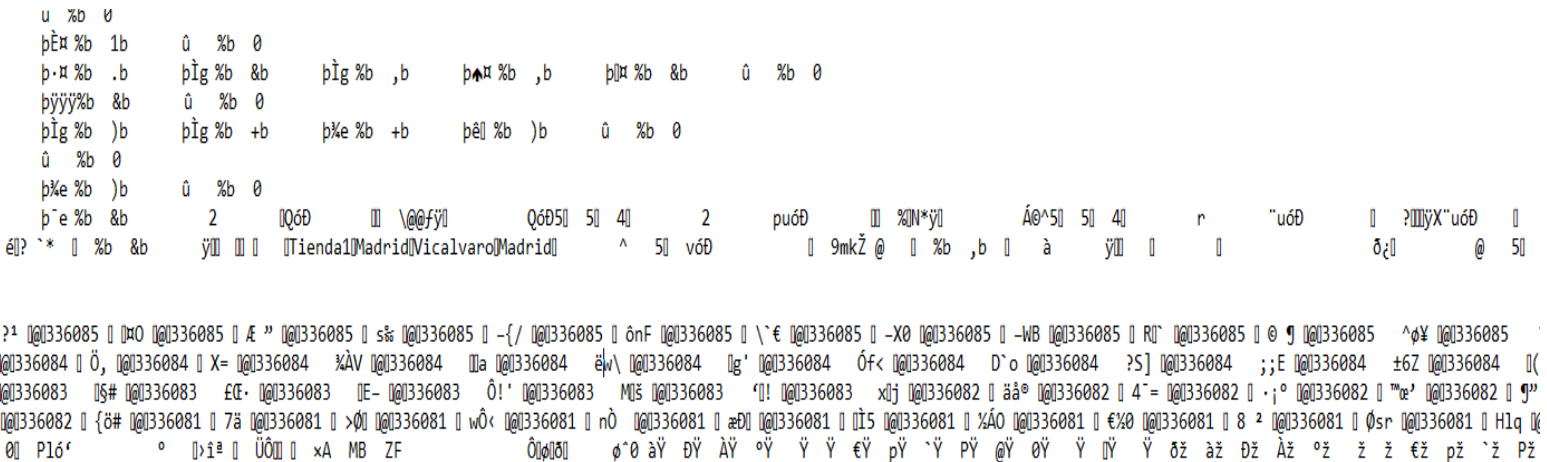
No se puede entender gran cosa, pero por las pocas letras que recoge el archivo, parece que ha recogido la creación de la nueva base de datos Tienda vacía.

Cuestión 2: Realizar una operación de inserción de una tienda sobre la base de datos **TIENDA**. Abrir el archivo de diario ¿Se encuentra reflejada la operación en el archivo del sistema? ¿En caso afirmativo, por qué lo hará?

Se realiza la siguiente inserción:

```
Tienda/postgres@PostgreSQL 12
Query Editor  Query History
1 insert into "Tienda" values(1,'Tienda1', 'Madrid','Vicalvaro','Madrid');
```

En la siguiente captura, se puede comprobar (con el texto que se entiende) que ha recogido la inserción.



Refleja la operación en el diario, por si se produce una desconexión/caída del servidor antes del checkpoint y necesita recuperar la operación.

Cuestión 3: ¿Para qué sirve el comando pg_waldump.exe? Aplicarlo al último fichero de WAL que se haya generado. Obtener las estadísticas de ese fichero y comentar qué se está viendo.

El ejecutable pg_waldump.exe, se encuentra en el directorio “C:\Program Files\PostgreSQL\12\bin”. Sirve para crear una versión de un archivo WAL que se pueda entender, ya que al abrir el diario con un editor de texto no se puede entender prácticamente nada.

Lo ejecutamos en la terminal y se lo aplicamos al último fichero creado (0000000100000009000000Do). En la terminal, ejecutamos lo siguiente, para mostrar las estadísticas de manera legible:

“pg_waldump.exe -p "C:\Program Files\PostgreSQL\12\data\pg_wal" -z 0000000100000009000000Do”. Introducimos el parámetro -p que nos encuentra el camino y el parámetro -z para que nos muestre un resumen de las estadísticas:

```
C:\Program Files\PostgreSQL\12\bin>pg_waldump.exe -p "C:\Program Files\PostgreSQL\12\data\pg_wal" -z 000000010000000900000000
Type      N      (%)      Record size      (%)      FPI size      (%)      Combined size      (%)
-----
XLOG      19 ( 0,61)      1302 ( 0,56)      960 ( 0,01)      2262 ( 0,01)
Transaction      5 ( 0,16)      9485 ( 4,09)      0 ( 0,00)      9485 ( 0,06)
Storage      23 ( 0,74)      966 ( 0,42)      0 ( 0,00)      966 ( 0,01)
CLOG      0 ( 0,00)      0 ( 0,00)      0 ( 0,00)      0 ( 0,00)
Database      2 ( 0,06)      76 ( 0,03)      0 ( 0,00)      76 ( 0,00)
Tablespace      0 ( 0,00)      0 ( 0,00)      0 ( 0,00)      0 ( 0,00)
MultiXact      0 ( 0,00)      0 ( 0,00)      0 ( 0,00)      0 ( 0,00)
RelMap      0 ( 0,00)      0 ( 0,00)      0 ( 0,00)      0 ( 0,00)
Standby      41 ( 1,32)      1990 ( 0,86)      0 ( 0,00)      1990 ( 0,01)
Heap2      0 ( 0,00)      0 ( 0,00)      0 ( 0,00)      0 ( 0,00)
Heap      2309 ( 74,08)      167525 ( 72,17)      15597640 ( 99,53)      15765165 ( 99,13)
Btree      718 ( 23,03)      50768 ( 21,87)      73120 ( 0,47)      123888 ( 0,78)
Hash      0 ( 0,00)      0 ( 0,00)      0 ( 0,00)      0 ( 0,00)
Gin      0 ( 0,00)      0 ( 0,00)      0 ( 0,00)      0 ( 0,00)
Gist      0 ( 0,00)      0 ( 0,00)      0 ( 0,00)      0 ( 0,00)
Sequence      0 ( 0,00)      0 ( 0,00)      0 ( 0,00)      0 ( 0,00)
SPGist      0 ( 0,00)      0 ( 0,00)      0 ( 0,00)      0 ( 0,00)
BRIN      0 ( 0,00)      0 ( 0,00)      0 ( 0,00)      0 ( 0,00)
CommitTs      0 ( 0,00)      0 ( 0,00)      0 ( 0,00)      0 ( 0,00)
ReplicationOrigin      0 ( 0,00)      0 ( 0,00)      0 ( 0,00)      0 ( 0,00)
Generic      0 ( 0,00)      0 ( 0,00)      0 ( 0,00)      0 ( 0,00)
LogicalMessage      0 ( 0,00)      0 ( 0,00)      0 ( 0,00)      0 ( 0,00)
-----
Total      3117      232112 [1,46%]      15671720 [98,54%]      15903832 [100%]
```

En el resumen de las estadísticas se puede ver el número y tamaño de Transacciones, almacenamiento, Bases de datos, índices BTree, etc, que recoge el archivo que hemos ejecutado. Por ejemplo, se puede ver que el diario recoge 2 Bases de Datos, 5 transacciones o 718 índices BTree con sus respectivos tamaños.

Cuestión 4: Determinar el identificador de la transacción que realizó la operación anterior. Aplicar el comando anterior al último fichero de WAL que se ha generado y mostrar los registros que se han creado para esa transacción. ¿Qué se puede ver? Interpretar los resultados obtenidos.

Para determinar el identificador, realizamos una serie de pasos. Primero ejecutamos el siguiente comando para ver las transacciones realizadas: "pg_waldump.exe -p "C:\Program Files\PostgreSQL\12\data\pg_wal" -x 000000010000000900000000". El parámetro -x seguido de un identificador hace acceder a los datos de la transacción con ese identificador. Al introducir el 0, nos aparecen todas las transacciones con sus correspondientes ID. Este sería el resultado:

```
C:\Program Files\PostgreSQL\12\bin>pg_waldump.exe -p "C:\Program Files\PostgreSQL\12\data\pg_wal" -x 0 0000001000000090000000D0
rmgr: Standby len (rec/tot): 54/ 54, tx: 0, lsn: 9/D008D8E8, prev 9/D008B8B0, desc: RUNNING_XACTS nextXid 8
rmgr: Standby len (rec/tot): 54/ 54, tx: 0, lsn: 9/D0509898, prev 9/D0507860, desc: RUNNING_XACTS nextXid 8
rmgr: Standby len (rec/tot): 54/ 54, tx: 0, lsn: 9/D08415D8, prev 9/D083F5A0, desc: RUNNING_XACTS nextXid 8
rmgr: Standby len (rec/tot): 54/ 54, tx: 0, lsn: 9/D0C2C6A0, prev 9/D0C2A668, desc: RUNNING_XACTS nextXid 8
rmgr: Standby len (rec/tot): 54/ 54, tx: 0, lsn: 9/D0E94D08, prev 9/D0E92CD0, desc: RUNNING_XACTS nextXid 8
rmgr: Standby len (rec/tot): 50/ 50, tx: 0, lsn: 9/D0EFB8C8, prev 9/D0EFB8A0, desc: RUNNING_XACTS nextXid 8
rmgr: Standby len (rec/tot): 50/ 50, tx: 0, lsn: 9/D0EFB900, prev 9/D0EFB8C8, desc: RUNNING_XACTS nextXid 8
rmgr: XLOG len (rec/tot): 114/ 114, tx: 0, lsn: 9/D0EFB938, prev 9/D0EFB900, desc: CHECKPOINT_ONLINE redo
st multi 1 in DB 1; oldest/newest commit timestamp xid: 0/0; oldest running xid 817; online
rmgr: Standby len (rec/tot): 50/ 50, tx: 0, lsn: 9/D0EFB9B0, prev 9/D0EFB938, desc: RUNNING_XACTS nextXid 8
rmgr: Standby len (rec/tot): 54/ 54, tx: 0, lsn: 9/D0EFC310, prev 9/D0EFB9E8, desc: RUNNING_XACTS nextXid 8
rmgr: Standby len (rec/tot): 54/ 54, tx: 0, lsn: 9/D0EFC348, prev 9/D0EFC310, desc: RUNNING_XACTS nextXid 8
rmgr: XLOG len (rec/tot): 114/ 114, tx: 0, lsn: 9/D0EFC380, prev 9/D0EFC348, desc: CHECKPOINT_ONLINE redo
st multi 1 in DB 1; oldest/newest commit timestamp xid: 0/0; oldest running xid 818; online
rmgr: Standby len (rec/tot): 50/ 50, tx: 0, lsn: 9/D0EFC468, prev 9/D0EFC420, desc: RUNNING_XACTS nextXid 8
rmgr: XLOG len (rec/tot): 30/ 30, tx: 0, lsn: 9/D0EFC4A0, prev 9/D0EFC468, desc: NEXTOID 33317
rmgr: Standby len (rec/tot): 54/ 54, tx: 0, lsn: 9/D0EFC4C0, prev 9/D0EFC4A0, desc: RUNNING_XACTS nextXid 8
rmgr: Standby len (rec/tot): 54/ 54, tx: 0, lsn: 9/D0EFD178, prev 9/D0EFD060, desc: RUNNING_XACTS nextXid 8
rmgr: XLOG len (rec/tot): 114/ 114, tx: 0, lsn: 9/D0EFD180, prev 9/D0EFD178, desc: CHECKPOINT_ONLINE redo
st multi 1 in DB 1; oldest/newest commit timestamp xid: 0/0; oldest running xid 819; online
rmgr: Standby len (rec/tot): 54/ 54, tx: 0, lsn: 9/D0EFD258, prev 9/D0EFD228, desc: RUNNING_XACTS nextXid 8
rmgr: XLOG len (rec/tot): 114/ 114, tx: 0, lsn: 9/D0EFD290, prev 9/D0EFD258, desc: CHECKPOINT_ONLINE redo
st multi 1 in DB 1; oldest/newest commit timestamp xid: 0/0; oldest running xid 819; online
rmgr: Standby len (rec/tot): 50/ 50, tx: 0, lsn: 9/D0EFD350, prev 9/D0EFD308, desc: RUNNING_XACTS nextXid 8
rmgr: Storage len (rec/tot): 42/ 42, tx: 0, lsn: 9/D0EFD388, prev 9/D0EFD350, desc: CREATE base/25125/25126
rmgr: Standby len (rec/tot): 90/ 90, tx: 0, lsn: 9/D0F1E528, prev 9/D0F1E4A0, desc: LOCK xid 820 db 25125 r
25134
rmgr: Standby len (rec/tot): 54/ 54, tx: 0, lsn: 9/D0F1E588, prev 9/D0F1E528, desc: RUNNING_XACTS nextXid 8
rmgr: Standby len (rec/tot): 50/ 50, tx: 0, lsn: 9/D0F37570, prev 9/D0F35110, desc: RUNNING_XACTS nextXid 8
rmgr: Standby len (rec/tot): 50/ 50, tx: 0, lsn: 9/D0F375A8, prev 9/D0F37570, desc: RUNNING_XACTS nextXid 8
rmgr: XLOG len (rec/tot): 114/ 114, tx: 0, lsn: 9/D0F375E8, prev 9/D0F375A8, desc: CHECKPOINT_ONLINE redo
st multi 1 in DB 1; oldest/newest commit timestamp xid: 0/0; oldest running xid 821; online
rmgr: Standby len (rec/tot): 50/ 50, tx: 0, lsn: 9/D0F37658, prev 9/D0F375E0, desc: RUNNING_XACTS nextXid 8
rmgr: Standby len (rec/tot): 50/ 50, tx: 0, lsn: 9/D0F37788, prev 9/D0F37790, desc: RUNNING_XACTS nextXid 8
rmgr: Standby len (rec/tot): 50/ 50, tx: 0, lsn: 9/D0F377F0, prev 9/D0F37788, desc: RUNNING_XACTS nextXid 8
rmgr: XLOG len (rec/tot): 114/ 114, tx: 0, lsn: 9/D0F37828, prev 9/D0F377F0, desc: CHECKPOINT_ONLINE redo
st multi 1 in DB 1; oldest/newest commit timestamp xid: 0/0; oldest running xid 822; online
rmgr: Standby len (rec/tot): 50/ 50, tx: 0, lsn: 9/D0F378A0, prev 9/D0F37828, desc: RUNNING_XACTS nextXid 8
```

Analizando cada una de las transacciones a través de su ID, intuimos que ID de la transacción que realizó la inserción anterior es 821 como podemos observar en la siguiente captura, al ver operaciones de inserción, cambiando el o de la consulta anterior por 821 (ID):

```
C:\Program Files\PostgreSQL\12\bin>pg_waldump.exe -p "C:\Program Files\PostgreSQL\12\data\pg_wal" -x 821 0000001000000090000000D0
rmgr: Heap len (rec/tot): 91/ 91, tx: 821, lsn: 9/D0F37690, prev 9/D0F37658, desc: INSERT+INIT off 1 flags 0x00, b
rmgr: Btree len (rec/tot): 94/ 94, tx: 821, lsn: 9/D0F376F0, prev 9/D0F37690, desc: NEWROOT lev 0, blkref #0: rel 1
rmgr: Btree len (rec/tot): 64/ 64, tx: 821, lsn: 9/D0F37750, prev 9/D0F376F0, desc: INSERT_LEAF off 1, blkref #0: r
rmgr: Transaction len (rec/tot): 34/ 34, tx: 821, lsn: 9/D0F37790, prev 9/D0F37750, desc: COMMIT 2020-04-30 16:53:39.7294
pg_waldump: fatal: error en registro de WAL en 9/D0F378A0: invalid record length at 9/D0F378D8: wanted 24, got 0
```

Igual que en la pregunta anterior, podemos usar el parámetro -z para que nos muestre las estadísticas:

```
C:\Program Files\PostgreSQL\12\bin>pg_waldump.exe -p "C:\Program Files\PostgreSQL\12\data\pg_wal" -x 821 -z 0000001000000090000000D0
Type N (%) Record size (%) FPI size (%) Combined size (%)
---- - -
XLOG 0 ( 0,00) 0 ( 0,00) 0 ( 0,00) 0 ( 0,00)
Transaction 1 ( 25,00) 34 ( 12,01) 0 ( 0,00) 34 ( 12,01)
Storage 0 ( 0,00) 0 ( 0,00) 0 ( 0,00) 0 ( 0,00)
CLOG 0 ( 0,00) 0 ( 0,00) 0 ( 0,00) 0 ( 0,00)
Database 0 ( 0,00) 0 ( 0,00) 0 ( 0,00) 0 ( 0,00)
Tablespace 0 ( 0,00) 0 ( 0,00) 0 ( 0,00) 0 ( 0,00)
MultiXact 0 ( 0,00) 0 ( 0,00) 0 ( 0,00) 0 ( 0,00)
RelMap 0 ( 0,00) 0 ( 0,00) 0 ( 0,00) 0 ( 0,00)
Standby 0 ( 0,00) 0 ( 0,00) 0 ( 0,00) 0 ( 0,00)
Heap2 0 ( 0,00) 0 ( 0,00) 0 ( 0,00) 0 ( 0,00)
Heap 1 ( 25,00) 91 ( 32,16) 0 ( 0,00) 91 ( 32,16)
Btree 2 ( 50,00) 158 ( 55,83) 0 ( 0,00) 158 ( 55,83)
Hash 0 ( 0,00) 0 ( 0,00) 0 ( 0,00) 0 ( 0,00)
Gin 0 ( 0,00) 0 ( 0,00) 0 ( 0,00) 0 ( 0,00)
Gist 0 ( 0,00) 0 ( 0,00) 0 ( 0,00) 0 ( 0,00)
Sequence 0 ( 0,00) 0 ( 0,00) 0 ( 0,00) 0 ( 0,00)
SPGist 0 ( 0,00) 0 ( 0,00) 0 ( 0,00) 0 ( 0,00)
BRIN 0 ( 0,00) 0 ( 0,00) 0 ( 0,00) 0 ( 0,00)
CommitTs 0 ( 0,00) 0 ( 0,00) 0 ( 0,00) 0 ( 0,00)
ReplicationOrigin 0 ( 0,00) 0 ( 0,00) 0 ( 0,00) 0 ( 0,00)
Generic 0 ( 0,00) 0 ( 0,00) 0 ( 0,00) 0 ( 0,00)
LogicalMessage 0 ( 0,00) 0 ( 0,00) 0 ( 0,00) 0 ( 0,00)
-----
Total 4 283 [100,00%] 0 [0,00%] 283 [100%]
```

Podemos observar que se ha creado un registro de tamaño 34.

Cuestión 5: Se va a crear un backup de la base de datos **TIENDA**. Este backup será utilizado más adelante para recuperar el sistema frente a una caída del sistema. Realizar solamente el backup mediante el procedimiento descrito en el apartado 25.3 del manual (versión 12 es *"Continuous Archiving and point-in-time recovery (PITR)"*).

Como nos indica en el apartado 25.3, vamos a usar la herramienta pg_basebackup.

En postgresql.conf cambiamos el archive_command para guardar el pg_wal correspondiente en un directorio nuevo.

```
archive_mode = on          # enables archiving; off, on, or always
                           # (change requires restart)
archive_command = 'copy "%p" "C:\\wal_backup\\%f"' # command to use to archive a logfile segment
```

Reiniciamos el servidor para que se lleven a cabo los cambios en postgresql.conf.

Para hacer el backup, se introduce lo siguiente:

```
C:\Program Files\PostgreSQL\12\bin>pg_basebackup.exe -D "C:\pg_basebackup\backup" -Ft -z -U postgres -p 5433
Contraseña:
C:\Program Files\PostgreSQL\12\bin>_
```

El parámetro -D indica en qué directorio se va a crear el archivo, por lo que primero creamos en el disco C un directorio que se llama pg_basebackup:

PerfLogs	14/05/2020 20:32	Carpeta de archivos
pg_basebackup	31/05/2020 12:41	Carpeta de archivos
PostgreSQL	29/01/2019 12:32	Carpeta de archivos
TDM-GCC-64	22/12/2017 21:08	Carpeta de archivos
Usuarios	23/09/2019 15:52	Carpeta de archivos
wal_backup	01/06/2020 13:32	Carpeta de archivos
Windows	14/05/2020 23:31	Carpeta de archivos

Ponemos su ruta y al final el nombre del archivo que va a crear (backup), podemos ver que se ha creado:

Nombre	Fecha de modificación	Tipo	Tamaño
backup	03/05/2020 12:03	Carpeta de archivos	

Después los parámetros -Ft y -z hacen que el backup se guarde en formato .gz.tar y el parámetro -U es para indicar el usuario (elegimos postgres, ya que tiene todos los permisos). Aquí vemos que se han creado las copias correctamente:

e equipo > OS (C:) > pg_basebackup > backup

Nombre	Fecha de modificación	Tipo	Tamaño
base.tar.gz	01/06/2020 13:32	8 Zip archive	3.930 KB
pg_wal.tar.gz	01/06/2020 13:32	8 Zip archive	17 KB

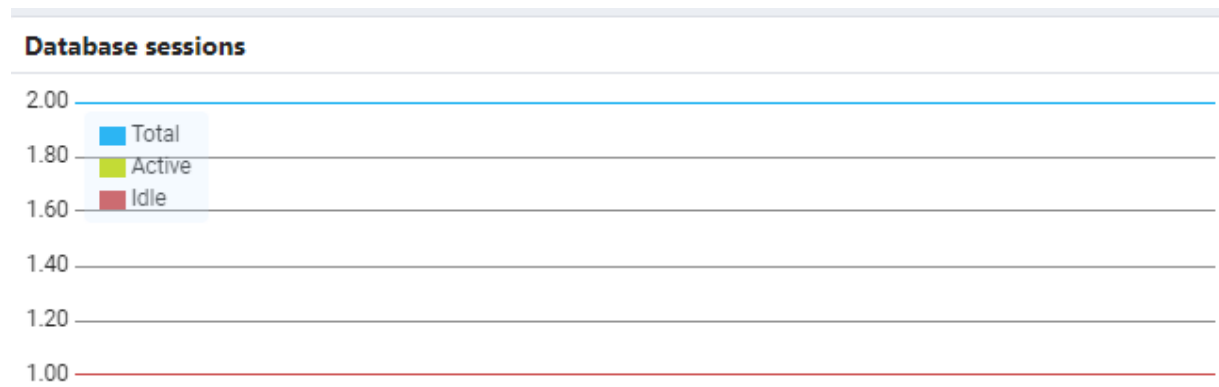
Se crean copias de seguridad de las bases de datos y del diario

Cuestión 6: Qué herramientas disponibles tiene PostgreSQL para controlar la actividad de la base de datos en cuanto a la concurrencia y transacciones? ¿Qué

información es capaz de mostrar? ¿Dónde se guarda dicha información? ¿Cómo se puede mostrar?

La herramienta principal para controlar la actividad de la base de datos es el Dashboard, donde se pueden ver las transacciones actuales del sistema y su información (tablas que está utilizando, si están activas o bloqueadas...). Se pueden ver las estadísticas en tiempo real de lo que está ocurriendo en el clúster de la base de datos.

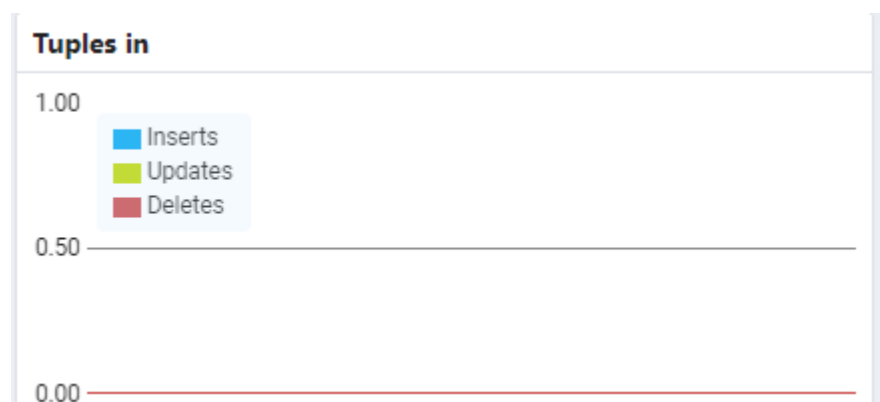
Se pueden ver las diferentes sesiones de la base de datos (totales, activas y paradas):



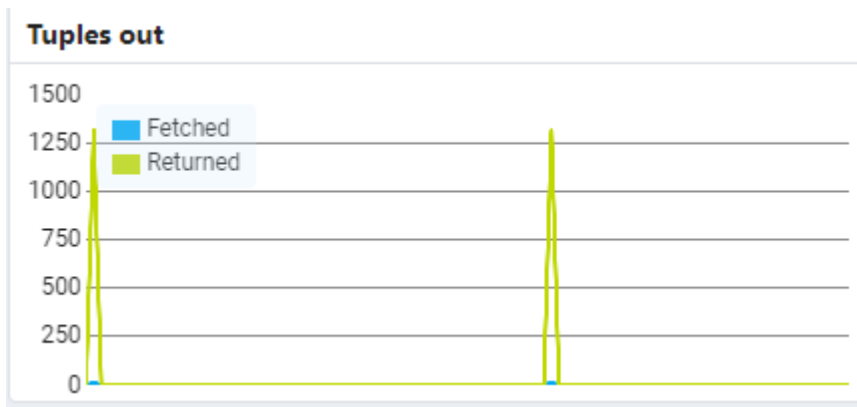
También, se pueden ver las transacciones que se están generando por segundo, cuántas han hecho commit o cuántas han abortado (RollBack).



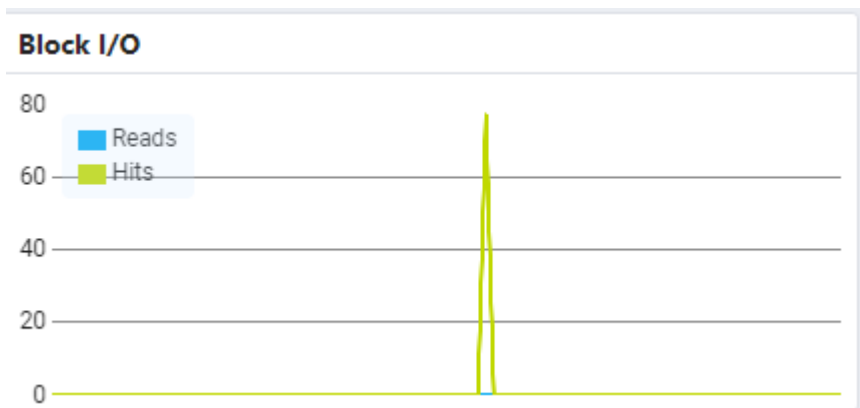
Las tuplas que se están insertando, borrando o modificando:



Las tuplas buscadas y devueltas:



Los bloques de entrada y salida que se han leído del disco o se han utilizado de la memoria (al estar ya cargados)



Se pueden ver las sesiones actuales:

	PID	User	Application	Client	Backend start	State	Wait event	Blocking PIDs
●	10940	postgres	pgAdmin 4 - DB:Tienda	::1	2020-05-17 18:18:57 CEST	active		
●	18728	postgres	pgAdmin 4 - CONN:1699128	::1	2020-05-17 18:20:28 CEST	idle	Client: ClientRead	

Se pueden ver los bloqueos, que indican las tuplas que utilizan (bloquean) cada tabla:

PID	Lock type	Target relation	Page	Tuple	vXID (target)	XID (target)	Class	Object ID	vXID (owner)	Mode	Granted?
10940	relation	pg_locks							6/922	AccessShareLock	true

Cuestión 7: Crear dos usuarios en la base de datos que puedan acceder a la base de datos **TIENDA** identificados como usuario1 y usuario2 que tengan permisos de lectura/escritura a la base de datos tienda, pero que no puedan modificar su estructura. Describir el proceso seguido.

Primero se crean los dos usuarios con el comando create user con sus correspondientes contraseñas:

```

6
7 create user usuario1 with password 'passusuario1';
8
9 create user usuario2 with password 'pasusuario2';

```

Ahora, para que tengan los permisos de lectura/escritura, les damos los permisos de select, update, insert y delete en todas las tablas de la base de datos Tienda:

```
grant select,update,delete,insert on all tables in schema public to usuario1;

grant select,update,delete,insert on all tables in schema public to usuario2;
```

Cuestión 8: Abrir una transacción que inserte una nueva tienda en la base de datos (NO cierre la transacción). Realizar una consulta SQL para mostrar todas las tiendas de la base de datos dentro de esa transacción. Consultar la información sobre lo que se encuentra actualmente activo en el sistema. ¿Qué conclusiones se pueden extraer?

Se realizan las siguientes consultas abriendo la transacción con begin, pero sin cerrarla:

```
begin;
insert into "Tienda" values(2,'Tienda2', 'Barcelona','Canaletas','Catalunya');
select * from "Tienda";
```

Nos muestra la tienda insertada pese a no haber hecho el commit:

Data Output

Explain

Messages

Notifications

	Id_tienda [PK] integer	Nombre text	Ciudad text	Barrio text	Provincia text
1	1	Tienda1	Madrid	Vicalvaro	Madrid
2	2	Tienda2	Barcelona	Canaletas	Catalunya

Al hacer el select dentro de la transacción, se puede ver que se muestra lo que acabamos de insertar, es decir, como se ya se hubiese hecho un commit del insert y no se hubiese volcado a memoria global. Esto es debido a que dentro de la transacción si puedes acceder a lo que se acaba de insertar pese a no estar todavía en memoria global.

Si consultamos el Dashboard, podemos ver lo que se está ejecutando en el sistema y podemos apreciar que se está ejecutando la transacción:

	PID	User	Application	Client	Backend start	State	Wait event
11788	11788	postgres	pgAdmin 4 - CONN:51298	::1	2020-05-20 19:23:33 CEST	idle in transaction	Client: ClientRead
16848	16848	postgres	pgAdmin 4 - CONN:863378	::1	2020-05-20 19:23:34 CEST	idle	Client: ClientRead
32692	32692	postgres	pgAdmin 4 - DB:Tienda	::1	2020-05-20 19:23:16 CEST	active	

Podemos ver que la transacción que se está ejecutando (ID 32692). Ahora, podemos comprobar los bloqueos que reciben la tabla usada (Tienda):

11788	relation	"Tienda_pk"					4/532	AccessShareLock	true
11788	relation	"Tienda"					4/532	AccessShareLock	true
11788	relation	"Tienda"					4/532	RowExclusiveLock	true

Vemos que se le conceden bloqueos a Tienda, por lo que si otra transacción intenta acceder a ella antes de terminar la actual, se bloqueará.

Cuestión 9: Cierre la transacción anterior. Utilizando pgAdmin o psql, abrir una transacción T1 en el usuario1 que realice las siguientes operaciones sobre la base de datos **TIENDA**. NO termine la transacción. Simplemente:

- Inserte una nueva tienda con ID_TIENDA 1000.
- Inserte un trabajador de la tienda anterior.
- Inserte un nuevo ticket del trabajador anterior con número 54321.

Cerramos la transacción anterior con commit:

```
commit;
```

COMMIT

Query returned successfully in 57 msec.

Establecemos la conexión con el usuario1:

```
set role usuario1;
```

Con el comando “select current_user” vemos que el usuario actual es el usuario1:

	current_user	
	name	
1	usuario1	

Hacemos la transacción con los inserts correspondientes:

```
begin;
insert into "Tienda" values(1000,'Tienda3', 'Sevilla','Giralda','Andalucia');
insert into "Trabajador" values(1,'46345677H', 'Siro','Lopez','Dependiente',2000,1000);
insert into "Ticket" values(54321,15, '05/05/2020','1');
```

Cuestión 10: Realizar cualquier consulta SQL que muestre los datos anteriores insertados para ver que todo está correcto.

Realizamos la siguiente consulta para comprobar que todo se ha insertado correctamente:

```
--Pregunta 10
select "fecha","Trabajador"."Nombre","Ciudad","Id_tienda" from "Tienda" inner join "Trabajador" on "Id_tienda_Tienda"="Id_tienda"
inner join "Ticket" on "codigo_trabajador" = "codigo_trabajador_Trabajador";
```

Vemos que se muestra todo correctamente:

	fecha	Nombre	Ciudad	Id_tienda
	date	character varying	text	integer
1	2020-05-05	Siro	Sevilla	1000

Cuestión 11: Establecer una **nueva conexión** con pgAdmin o psql a la base de datos con el usuario2 (abrir otra sesión diferente a la abierta actualmente que pertenezca al usuario2) y realizar la misma consulta. ¿Se nota algún cambio? En caso afirmativo, ¿a qué puede ser debido el diferente funcionamiento en la base de datos para ambas consultas? ¿Qué información de actividad hay registrada en la base de datos en este momento?

Establecemos una nueva conexión en una consola nueva con el usuario2:

```
set role usuario2;
```

Vemos que el usuario2 es el actual en la consola nueva:

current_user	
name	
1	usuario2

Y realizamos otra vez la consulta:

```
select "fecha", "Trabajador"."Nombre", "Ciudad", "Id_tienda" from "Tienda" inner join "Trabajador" on "Id_tienda_Tienda"="Id_tienda" inner join "Ticket" on "codigo_trabajador" = "codigo_trabajador_Trabajador";
```

Vemos que está vacía:

Data Output		Explain	Messages	Notifications
fecha		Nombre	Ciudad	Id_tienda
date		character varying	text	integer

Este cambio se debe a que la anterior transacción no ha hecho commit por lo que se encuentra en la memoria local de esa transacción y aún no ha volcado a memoria global. En el momento que cambiamos de usuario, no estamos en la misma transacción por lo que no nos muestra nada hasta que no hagamos el commit en la anterior.

La información de la actividad es la siguiente:

		PID	User	Application	Client	Backend start	State	Wait event	B
+	■	11788	postgres	pgAdmin 4 - CONN:51298	::1	2020-05-20 19:23:33 CEST	idle in transaction	Client: ClientRead	
+	■	16848	postgres	pgAdmin 4 - CONN:863378	::1	2020-05-20 19:23:34 CEST	idle	Client: ClientRead	
+	■	32692	postgres	pgAdmin 4 - DB:Tienda	::1	2020-05-20 19:23:16 CEST	active		
11788	relation	"Tienda_pk"					4/534	AccessShareLock	true
11788	relation	"Tienda"					4/534	AccessShareLock	true
11788	relation	"Tienda"					4/534	RowExclusiveLock	true

Vemos que se está ejecutando la transacción que hemos iniciado y se ha concedido el bloqueo a la tabla Tienda.

Cuestión 12: ¿Se encuentran los nuevos datos físicamente en las tablas de la base de datos? Entonces, ¿de dónde se obtienen los datos de la cuestión 2.10 y/o de la 2.11?

No se encuentran los datos físicamente, ya que el resultado desde la consola del primer usuario (pregunta 10) se obtiene de la memoria local de la transacción que se está ejecutando y no vuelcan a memoria global al no haber hecho commit.

En la pregunta 11 se cambia de usuario, por lo que va a obtener los datos almacenados en la memoria global de la base de datos. Los datos de la transacción anterior no estarán, ya que no se ha hecho commit por lo que no pasan a memoria global. Hasta que no se comprometa, solo tendrá acceso a esos datos el usuario de la pregunta 10.

Cuestión 13: Finalizar con éxito la transacción T1 y realizar la consulta de la cuestión 2.10 y 2.11 sobre ambos usuarios conectados. ¿Qué es lo que se obtiene ahora? ¿Por qué?

Se finaliza con éxito la transacción con commit.

```
commit;
```

En el usuario1 se mostrará lo mismo, aunque ahora los datos estarán en memoria global:

Data Output	Explain	Messages	Notifications
fecha	Nombre	Ciudad	Id_tienda
date	character varying	text	integer
1	2020-05-05	Siro	Sevilla
			1000

En el usuario2 sí se mostrarán los datos cargados en la transacción

Data Output	Explain	Messages	Notifications
fecha	Nombre	Ciudad	Id_tienda
date	character varying	text	integer
1	2020-05-05	Siro	Sevilla
			1000

Esto se debe a que al hacer el commit los datos pasan a memoria global, por lo que todos los usuarios con los permisos correspondientes podrán ver la información.

Cuestión 14: Sin ninguna transacción en curso, abrir una transacción en un usuario cualquiera y realizar las siguientes operaciones:

- Insertar una tienda nueva con ID_TIENDA a 2000.
- Insertar un trabajador de la tienda 2000.
- Insertar un ticket del trabajador anterior con número 54300.
- Hacer una modificación del trabajador para cambiar el número de tienda de 2000 a 1000.
- Cerrar la transacción.

¿Cuál es el estado final de la base de datos? ¿Por qué?

Realizamos las tres inserciones y después la modificación, poniendo commit al final para cerrar la transacción:

```
--Pregunta 14
begin;
insert into "Tienda" values(2000,'Tienda4', 'Madrid','Barrio Salamanca','Madrid');
insert into "Trabajador" values(2,'46345688K', 'Roberto','Martinez','Encargado',3500,2000);
insert into "Ticket" values(54300,40, '06/05/2020','2');
update "Trabajador" set "Id_tienda_Tienda"=1000 where "Id_tienda_Tienda"=2000;
commit;
```

Hacemos la consulta anterior:

```
select "fecha","Trabajador"."Nombre","Ciudad","Id_tienda" from "Tienda" inner join "Trabajador" on "Id_tienda_Tienda"="Id_tienda"
inner join "Ticket" on "codigo_trabajador" = "codigo_trabajador_Trabajador";
```

Y nos da el siguiente resultado:

Data Output Explain Messages Notifications

	fecha date	Nombre character varying	Ciudad text	Id_tienda integer
1	2020-05-05	Siro	Sevilla	1000
2	2020-05-06	Roberto	Sevilla	1000

Vemos que nos lo ha insertado correctamente. Primero se inserta con el ID_Tienda=2000, pero al hacer la posterior modificación en el trabajador, para que trabaje en la tienda con ID 1000, por lo tanto, finalmente se muestra con la modificación.

Cuestión 15: Repetir la cuestión 9 con otra tienda, trabajador y ticket. Realizar la misma consulta de la cuestión 10, pero ahora terminar la transacción con un ROLLBACK y repetir la consulta con los mismos dos usuarios. ¿Cuál es el resultado? ¿Por qué?

Insertamos lo siguiente, pero ahora terminado con un rollback.

```
--Pregunta 15
begin;
insert into "Tienda" values(5,'Tienda5', 'Barcelona','Diagonal','Catalunya');
insert into "Trabajador" values(3,'53645688D', 'Diego','Alonso','Mozo',1750,5);
insert into "Ticket" values(10,85, '06/06/2020','3');
rollback;
```

Si ejecutamos la consulta de la 9 con el usuario actual:

	fecha date	Nombre character varying	Ciudad text	Id_tienda integer
1	2020-05-05	Siro	Sevilla	1000
2	2020-05-06	Roberto	Sevilla	1000

Si lo ejecutamos con el usuario2:

	fecha date	Nombre character varying	Ciudad text	Id_tienda integer
1	2020-05-05	Siro	Sevilla	1000
2	2020-05-06	Roberto	Sevilla	1000

Vemos que no se ha insertado nada. Esto es debido a que el RollBack hace que la transacción aborte, por lo que la base de datos vuelve a su estado inicial y no se harán las inserciones introducidas

Cuestión 16: Cerrar todas las sesiones anteriores. Abrir una sesión con el usuario1 de la base de datos **TIENDA**. Insertar la siguiente información en la base de datos:

- Insertar una tienda con id_tienda de 31145.
- Insertar un trabajador que pertenezca a la tienda anterior y tenga un código de 45678.

Se abre la sesión con el usuario1:

```
set role usuario1;
```

Insertamos la tienda y el trabajador con las claves pedidas:

```
insert into "Tienda" values(31145,'Tienda10', 'Cadiz','Puerto de Santa Maria','Andalucia');
insert into "Trabajador" values(45678,'24789688X', 'Ana','Gonzalez','Dependiente',2000,31145);
```

Cuestión 17: Abrir una sesión con el usuario2 a la base de datos **TIENDA**. Abrir una transacción T2 en este usuario2 y realizar una modificación de la tienda código 31145 para cambiar el nombre a “Tienda Alcalá”. ¿Qué actividad hay registrada en la base de datos? ¿Cuál es la información guardada en la base de datos? ¿Por qué?

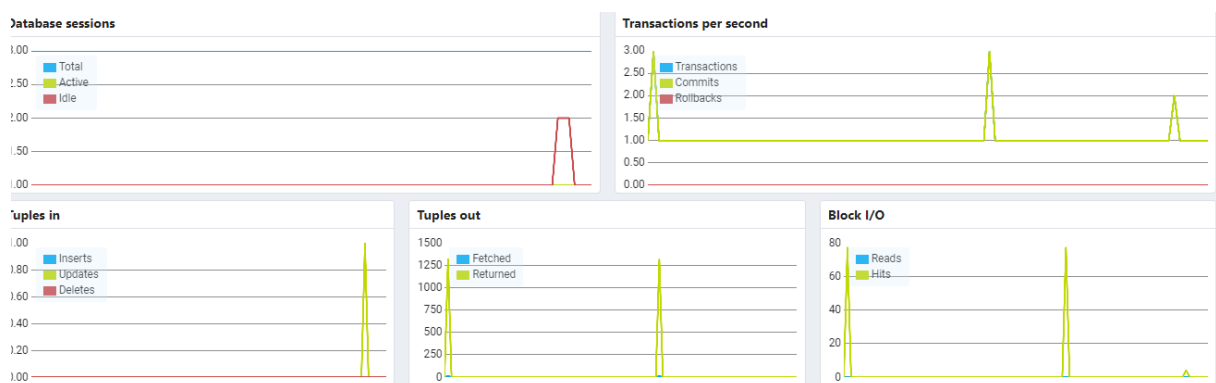
Se abre la sesión con el usuario2

```
set role usuario2;
```

Se hace la modificación:

```
begin;
update "Tienda" set "Nombre"='Tienda Alcalá' where "Id_tienda"=31145;
```

Consultamos el Dashboard de PgAdmin para ver la actividad:



	PID	User	Application	Client	Backend start	State	Wait event	E
✖	2636	postgres	pgAdmin 4 - CONN:3565623	::1	2020-05-08 10:57:28 CEST	idle in transaction	Client: ClientRead	
✖	36172	postgres	pgAdmin 4 - DB:Tienda	::1	2020-05-08 10:57:20 CEST	active		
✖	76040	postgres	pgAdmin 4 - CONN:6674819	::1	2020-05-08 10:57:28 CEST	idle	Client: ClientRead	

PID	Lock type	Target relation	Page	Tuple	vXID (target)	XID (target)	Class	Object ID	vXID (owner)	Mode	Granted?
2636	relation	"Tienda_pk"							8/3991	RowExclusiveLock	true
2636	relation	"Tienda"							8/3991	RowExclusiveLock	true
36172	relation	pg_locks							6/8884	AccessShareLock	true

Podemos observar que se ha iniciado la base de datos y, por lo tanto, está activa. También, se puede comprobar que el entorno está ejecutando una transacción y aún no ha terminado (no se ha hecho commit). En la pestaña locks, podemos ver las relaciones usadas (que por lo tanto se les asigna un bloqueo) en la transacción (Tienda).

Podemos ver ahora que desde la consola del usuario2 se ha hecho el update, realizando la siguiente consulta: "select "Trabajador"."Nombre","Tienda"."Nombre","Ciudad","Id_tienda" from "Tienda" inner join "Trabajador" on "Id_tienda_Tienda"="Id_tienda"":

Data Output	Explain	Messages	Notifications
Nombre character varying	Nombre text	Ciudad text	Id_tienda integer
1 Siro	Tienda3	Sevilla	1000
2 Roberto	Tienda3	Sevilla	1000
3 Ana	Tienda Alcala	Cadiz	31145

En cambio, si la ejecutamos desde la otra consola del otro usuario, podemos ver que aún no se ha actualizado:

Data Output	Explain	Messages	Notifications
Nombre character varying	Nombre text	Ciudad text	Id_tienda integer
1 Siro	Tienda3	Sevilla	1000
2 Roberto	Tienda3	Sevilla	1000
3 Ana	Tienda10	Cadiz	31145

Esto es debido a que, al no hacer el commit, todavía no se ha hecho la actualización en memoria global y solo está en la memoria local de la transacción del usuario2.

Cuestión 18. Abra una transacción T1 en el usuario1. Haga una actualización del trabajador con número 45678 para cambiar el salario a 3000. ¿Qué actividad hay registrada en la base de datos? ¿Cuál es la información guardada en la base de datos? ¿Por qué?

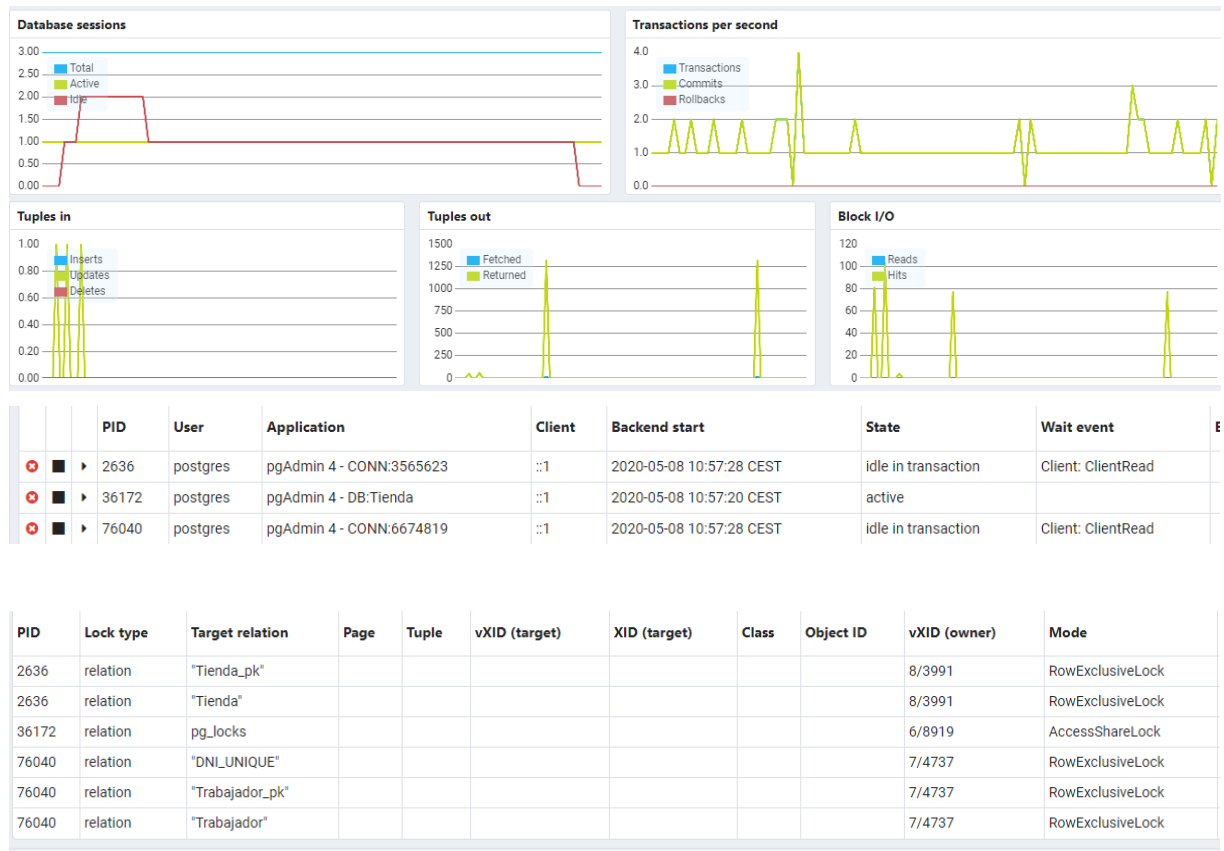
Sin cerrar la anterior transacción, abrimos conexión en otra consola con el usuario1:

```
set role usuario1;
```

Relizamos la actualización pedida:

```
begin;  
update "Trabajador" set "Salario"=3000 where "codigo_trabajador"=45678;
```

Vemos la actividad:



Podemos observar que están las dos transacciones en curso y en la pestaña locks, podemos ver las relaciones usadas (que por lo tanto se les asigna un lock) para cada transacción (Trabajador, Tienda y sus claves).

Ahora, en la consola del usuario1 (T1), podemos ver los datos de Tienda y Trabajador:

Id_tienda	Nombre	Ciudad	Barrio	Provincia	codigo_trabajador	DNI	Nombre	Apellidos	Puesto	Salario
integer	text	text	text	text	integer	character varying (9)	character varying	character varying	character varying	integer
1000	Tienda3	Sevilla	Giralda	Andalucia		1 46345677H	Siro	Lopez	Dependiente	2000
1000	Tienda3	Sevilla	Giralda	Andalucia		2 46345688K	Roberto	Martinez	Encargado	3500
31145	Tienda10	Cadiz	Puerto de...	Andalucia	45678	24789688X	Ana	Gonzalez	Dependiente	3000

Podemos observar que se ha actualizado el salario a 3000 del trabajador con Id 31145, pero no el nombre de la tienda.

Ahora, en la consola del usuario2 (T2), podemos ver los datos de Tienda y Trabajador:

Id_tienda	Nombre	Ciudad	Barrio	Provincia	codigo_trabajador	DNI	Nombre	Apellidos	Puesto	Salario
integer	text	text	text	text	integer	character varying (9)	character varying	character varying	character varying	integer
1000	Tienda3	Sevilla	Giralda	Andalucia		1 46345677H	Siro	Lopez	Dependiente	2000
1000	Tienda3	Sevilla	Giralda	Andalucia		2 46345688K	Roberto	Martinez	Encargado	3500
31145	Tienda Alcala	Cadiz	Puerto de...	Andalucia	45678	24789688X	Ana	Gonzalez	Dependiente	2000

Podemos observar que se ha actualizado el nombre de la tienda a Tienda Alcala, pero no el salario.

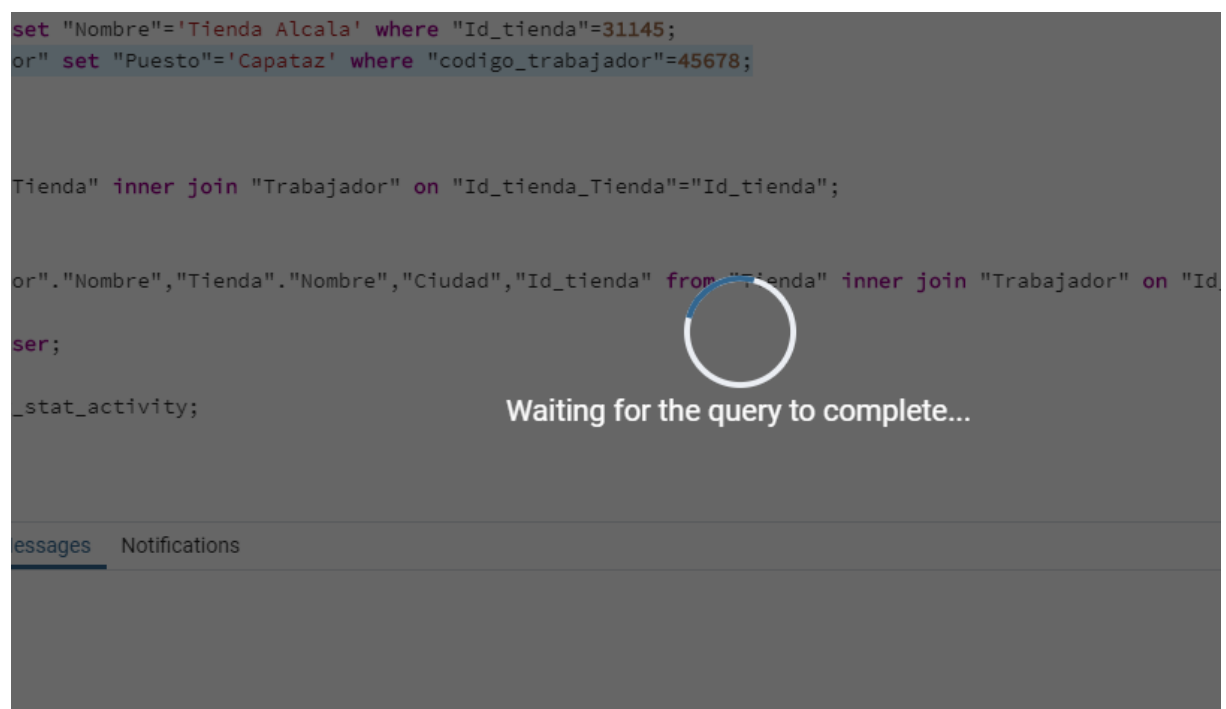
Esto se debe a que las actualizaciones se hacen en memoria local de cada transacción, por lo tanto, hasta que no se haga el commit, no pasarán a memoria global y solo se podrán ver a través de sus consolas correspondientes.

Cuestión 19: En la transacción T2, realice una modificación del trabajador con código 45678 para cambiar el puesto a “Capataz”. ¿Qué actividad hay registrada en la base de datos? ¿Cuál es la información guardada en la base de datos? ¿Por qué?

Hacemos la actualización dentro de T2 con el usuario2:

```
update "Trabajador" set "Puesto"='Capataz' where "codigo_trabajador"=45678;
```

No se puede realizar la actualización todavía, ya que la transacción T1 está escribiendo en la tabla Trabajador por lo que las dos transacciones entran en conflicto y T2 queda bloqueada:



Ahora vemos la actividad:

		PID	User	Application	Client	Backend start	State	Wait event
+	■	▶ 2636	postgres	pgAdmin 4 - CONN:3565623	::1	2020-05-08 10:57:28 CEST	active	Lock: transactionid
+	■	▶ 36172	postgres	pgAdmin 4 - DB:Tienda	::1	2020-05-08 10:57:20 CEST	active	
+	■	▶ 76040	postgres	pgAdmin 4 - CONN:6674819	::1	2020-05-08 10:57:28 CEST	idle in transaction	Client: ClientRead

PID	Lock type	Target relation	Page	Tuple	vXID (target)	XID (target)	Class	Object ID	vXID (owner)	Mode	Granted?
2636	relation	"Trabajador"							8/3991	RowExclusiveLock	true
2636	relation	"Tienda"							8/3991	RowExclusiveLock	true
2636	relation	"Tienda_pk"							8/3991	RowExclusiveLock	true
2636	relation	"DNI_UNIQUE"							8/3991	RowExclusiveLock	true
2636	relation	"Trabajador_pk"							8/3991	RowExclusiveLock	true
2636	tuple	"Trabajador"	0	32					8/3991	ExclusiveLock	true
36172	relation	pg_locks							6/8957	AccessShareLock	true
76040	relation	"Trabajador"							7/4737	RowExclusiveLock	true
76040	relation	"Trabajador_pk"							7/4737	RowExclusiveLock	true
76040	relation	"DNI_UNIQUE"							7/4737	RowExclusiveLock	true

Podemos observar que la transacción T2 (PID 2636) sigue activa en el sistema, pero la marca como bloqueada y en la pestaña locks, vemos que la transacción T2 quiere acceder a la tabla Trabajador.

En el usuario1 sigue sin actualizarse en nombre de la tienda, ya que aún no ha terminado la transacción del usuario2 pese a estar bloqueada:

Id_tienda	Nombre	Ciudad	Barrio	Provincia	codigo_trabajador	DNI	Nombre	Apellidos	Puesto	Salario	klr
1000	Tienda3	Sevilla	Giralda	Andalucia	1	46345677H	Siro	Lopez	Dependiente	2000	
1000	Tienda3	Sevilla	Giralda	Andalucia	2	46345688K	Roberto	Martinez	Encargado	3500	
31145	Tienda10	Cadiz	Puerto de...	Andalucia	45678	24789688X	Ana	Gonzalez	Dependiente	3000	

Cuestión 20: En la transacción T1, realice una modificación de la tienda con código 31145 para modificar el barrio y poner "El Ensanche". ¿Qué actividad hay registrada en la base de datos? ¿Cuál es la información guardada en la base de datos? ¿Por qué?

Hacemos la actualización en T1:

```
update "Tienda" set "Barrio"='El Ensanche' where "Id_tienda"=31145;
```

Al estar la transacción T2 modificando la tabla Tienda, se bloquea T1, pero al estar T2 también bloqueada se detecta un interbloqueo. La consola nos devuelve lo siguiente:

```
ERROR: se ha detectado un deadlock
DETAIL: El proceso 76040 espera ShareLock en transacción 967; bloqueado por proceso 2636.
El proceso 2636 espera ShareLock en transacción 968; bloqueado por proceso 76040.
HINT: Vea el registro del servidor para obtener detalles de las consultas.
CONTEXT: mientras se actualizaba la tupla (0,13) en la relación «Tienda»
SQL state: 40P01
```

Si vemos la actividad:

		PID	User	Application	Client	Backend start	State	Wait event	
+	■	▶ 2636	postgres	pgAdmin 4 - CONN:3565623	1	2020-05-08 10:57:28 CEST	idle in transaction	Client: ClientRead	
+	■	▶ 36172	postgres	pgAdmin 4 - DB:Tienda	1	2020-05-08 10:57:20 CEST	active		
+	■	▶ 76040	postgres	pgAdmin 4 - CONN:6674819	1	2020-05-08 10:57:28 CEST	idle in transaction (aborted)	Client: ClientRead	

PID	Lock type	Target relation	Page	Tuple	vXID (target)	XID (target)	Class	Object ID	vXID (owner)	Mode	Granted?
2636	relation	"DNI_UNIQUE"							8/3991	RowExclusiveLock	true
2636	relation	"Trabajador_pk"							8/3991	RowExclusiveLock	true
2636	relation	"Trabajador"							8/3991	RowExclusiveLock	true
2636	relation	"Tienda_pk"							8/3991	RowExclusiveLock	true
2636	relation	"Tienda"							8/3991	RowExclusiveLock	true
36172	relation	pg_locks							6/9302	AccessShareLock	true

Comprobamos que PostgreSQL ha decidido matar a T1 (vemos que está abortada) y T2 se ha desbloqueado y sigue activa.

Ahora, si mostramos la información de T2, vemos que la actualización que había la había bloqueado se ha realizado correctamente, ya que se ha desbloqueado sin problema:

Id_tienda	Nombre	Ciudad	Barrio	Provincia	codigo_trabajador	DNI	Nombre	Apellidos	Puesto	Salario
integer	text	text	text	text	integer	character varying (9)	character varying	character varying	character varying	integer
1000	Tienda3	Sevilla	Giralda	Andalucia		1 46345677H	Siro	Lopez	Dependiente	2000
1000	Tienda3	Sevilla	Giralda	Andalucia		2 46345688K	Roberto	Martinez	Encargado	3500
31145	Tienda Alcalá	Cádiz	Puerto de...	Andalucia	45678	24789688X	Ana	Gonzalez	Capataz	2000

Cuestión 21: Comprometa ambas transacciones T1 y T2. ¿Cuál es el valor final de la información modificada en la base de datos **TIENDA**? ¿Por qué?

Hacemos el commit en ambas y vemos la información de cada una. En T2 la información queda así:

Id_tienda	Nombre	Ciudad	Barrio	Provincia	codigo_trabajador	DNI	Nombre	Apellidos	Puesto	Salario
integer	text	text	text	text	integer	character varying (9)	character varying	character varying	character varying	integer
1000	Tienda3	Sevilla	Giralda	Andalucia		1 46345677H	Siro	Lopez	Dependiente	2000
1000	Tienda3	Sevilla	Giralda	Andalucia		2 46345688K	Roberto	Martinez	Encargado	3500
31145	Tienda Alcalá	Cádiz	Puerto de...	Andalucia	45678	24789688X	Ana	Gonzalez	Capataz	2000

Vemos que la información se actualiza correctamente, ya que se ha cerrado con el commit sin ningún problema. Aunque vemos que salario sigue siendo 2000 (lo explicamos a continuación). Vemos la información de T1:

Id_tienda	Nombre	Ciudad	Barrio	Provincia	codigo_trabajador	DNI	Nombre	Apellidos	Puesto	Salario
integer	text	text	text	text	integer	character varying (9)	character varying	character varying	character varying	integer
1000	Tienda3	Sevilla	Giralda	Andalucia		1 46345677H	Siro	Lopez	Dependiente	2000
1000	Tienda3	Sevilla	Giralda	Andalucia		2 46345688K	Roberto	Martinez	Encargado	3500
31145	Tienda Alcalá	Cádiz	Puerto de...	Andalucia	45678	24789688X	Ana	Gonzalez	Capataz	2000

Podemos ver que el salario sigue siendo 2000, por lo que no se ha hecho la actualización que sí había hecho dentro de la transacción. Esto es debido a que PostgreSQL ha decidido matarla y por lo tanto ha hecho un RollBack obviando todos los cambios que se habían realizado en la transacción.

Cuestión 22: Cerrar todas las sesiones anteriores. Abrir una sesión con el usuario1 de la base de datos **TIENDA**. Insertar en la tabla tienda una nueva tienda con código 6789. Abrir una transacción T1 en este usuario y realizar una modificación de la tienda con código 6789 y actualizar el nombre a “Mediamarkt”. No cierre la transacción.

Iniciamos sesión con el usuario1 y hacemos la inserción pedida:

```
set role usuario1;
insert into "Tienda" values(6789,'Tienda13', 'Coslada','Central','Madrid');
```

Hacemos la actualización pedida abriendo una transacción sin cerrarla:

```
begin;
update "Tienda" set "Nombre"='Mediamarkt' where "Id_tienda"=6789;
```

Si hacemos una consulta de todas las tiendas podemos ver que se ha actualizado sin problema:

	Id_tienda [PK] integer	Nombre text	Ciudad text	Barrio text	Provincia text
1	1	Tienda1	Madrid	Vicalvaro	Madrid
2	31145	Tienda Alcalá	Cádiz	Puerto de...	Andalucía
3	6789	Mediamarkt	Coslada	Central	Madrid
4	2	Tienda2	Barcelona	Canaletas	Cataluña
5	1000	Tienda3	Sevilla	Giralda	Andalucía
6	2000	Tienda4	Madrid	Barrio Sal...	Madrid

Cuestión 23: Abrir una sesión con el usuario2 de la base de datos **TIENDA**. Abrir una transacción T2 en este usuario y realizar una modificación de la tienda con código 6789 y cambiar el nombre a “Saturn”. No cierre la transacción. ¿Qué es lo que ocurre? ¿Por qué? ¿Qué información se puede obtener de la actividad de ambas transacciones en el sistema? ¿Es lógica esa información? ¿Por qué?

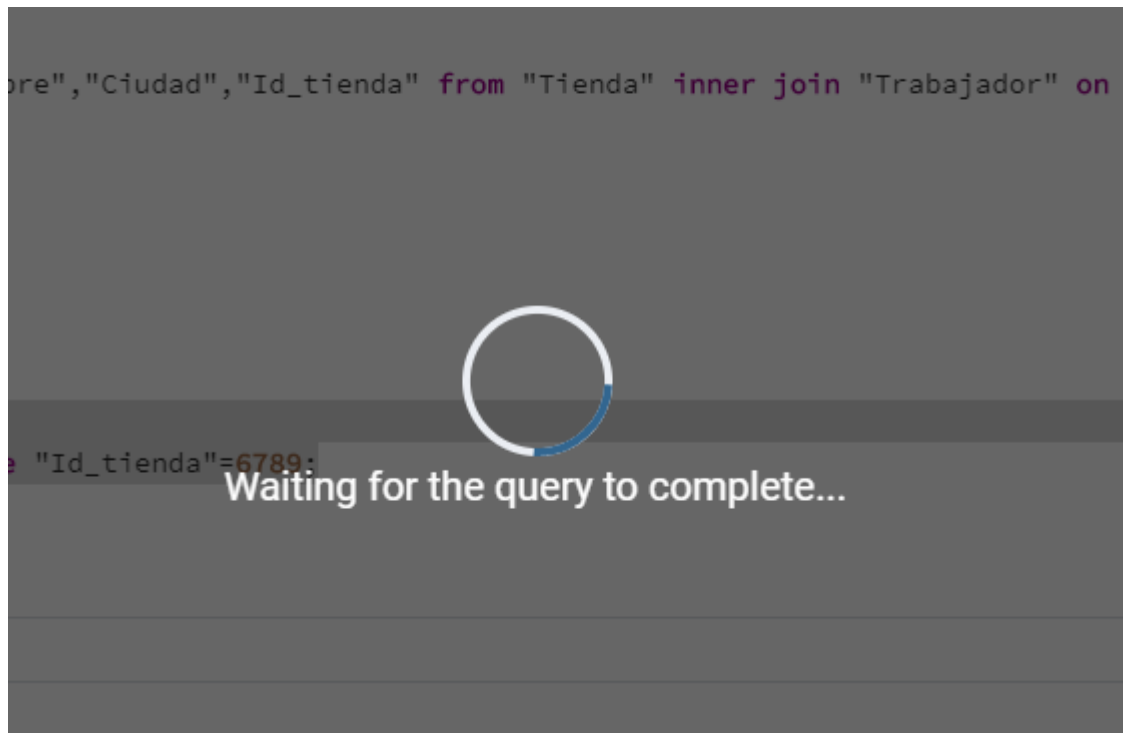
Abrimos otra consola con el usuario2:

```
set role usuario2;
```

Hacemos la actualización pedida dentro de una transacción sin cerrarla:

```
begin;
update "Tienda" set "Nombre"='Saturn' where "Id_tienda"=6789;
|
```

La transacción queda bloqueada:



Esto es debido a que la otra transacción (T1) está escribiendo en la tabla Tienda, por lo que las dos transacciones entran en conflicto y T2 queda bloqueada, ya que durante

la transacción T1 no puede acceder otra transacción a la tabla, por lo que seguirá bloqueada hasta que T1 cierre la transacción o se produzca un interbloqueo.

Vemos la actividad que hay en el sistema:

		PID	User	Application	Client	Backend start	State	Wait event
✖	■ ▶	20572	postgres	pgAdmin 4 - CONN:6914129	::1	2020-05-11 19:15:02 CEST	idle in transaction	Client: ClientRead
✖	■ ▶	54480	postgres	pgAdmin 4 - CONN:787579	::1	2020-05-11 19:15:01 CEST	active	Lock: transactionid
✖	■ ▶	66160	postgres	pgAdmin 4 - DB:Tienda	::1	2020-05-11 19:14:39 CEST	active	

Vemos que la transacción con PID 54480 (T2) ha sido bloqueada y ha intentado acceder a una tabla que ya tenía un bloqueo puesto por otra transacción (T1), por lo tanto, T2 está esperando a que T1 acabe para desbloquearse y poder acceder sin problemas a la tabla Tienda.

Cuestión 24: Comprometa la transacción T1, ¿Qué es lo que ocurre? ¿Por qué? ¿Cuál es el estado final de la información de la tienda con código 6789 para ambos usuarios? ¿Por qué?

Realizamos el commit en T1:

```
commit;
```

Se desbloquea la transacción T2 y se realiza la actualización con éxito:

Data Output	Explain	Messages	Notifications
UPDATE 1			
Query returned successfully in 10 min 20 secs.			

Al comprometer (commit) la transacción T1, se ha quitado el lock de la tabla Tienda, ya que, al terminar la transacción, ha terminado de escribir en la tabla, por lo que la transacción T2 es desbloqueada y puede hacer la actualización que no se le había permitido hacer todavía. La información en usuario1 sería la siguiente:

	Id_tienda [PK] integer	Nombre text	Ciudad text	Barrio text	Provincia text
1	1	Tienda1	Madrid	Vicalvaro	Madrid
2	31145	Tienda Alcala	Cadiz	Puerto de...	Andalucia
3	6789	Mediamarkt	Coslada	Central	Madrid
4	2	Tienda2	Barcelona	Canaletas	Catalunya
5	1000	Tienda3	Sevilla	Giralda	Andalucia
6	2000	Tienda4	Madrid	Barrio Sal...	Madrid

Vemos que se ha aplicado la actualización de T1, pero no de T2. Esto es debido a que T2 todavía no se ha cerrado, por lo que la actualización de la tabla solo se ha realizado en su memoria local. La actualización de T1 (Mediamarkt) se ha realizado

correctamente y se ha guardado en memoria global. Hasta que no se cierre T2, no se volcará la actualización realizada (Saturn) a memoria global.

Vemos la información de T2:

	Id_tienda [PK] integer	Nombre text	Ciudad text	Barrio text	Provincia text
1	1	Tienda1	Madrid	Vicalvaro	Madrid
2	31145	Tienda Alcala	Cadiz	Puerto de...	Andalucia
3	6789	Saturn	Coslada	Central	Madrid
4	2	Tienda2	Barcelona	Canaletas	Catalunya
5	1000	Tienda3	Sevilla	Giralda	Andalucia
6	2000	Tienda4	Madrid	Barrio Sal...	Madrid

Como acabamos de comentar, la actualización se ha realizado correctamente, pero solo se puede acceder a dicha información desde T2, ya que todavía no está en memoria global.

Cuestión 25: Comprometa la transacción T2, ¿Qué es lo que ocurre? ¿Por qué? ¿Cuál es el estado final de la información de la tienda con código 6789? ¿Por qué?

Realizamos el commit en T2:

```
43  commit;  
44
```

La transacción termina satisfactoriamente y el commit se puede hacer sin problemas. A diferencia del interbloqueo ocurrido anteriormente, la transacción T2 es desbloqueada sin problemas, ya que primero termina T1 satisfactoriamente sin ocurrir ningún RollBack y así, T2 puede terminar también sin problemas.

Desde cualquiera de los dos usuarios la información de la tabla Tienda es la siguiente:

	Id_tienda [PK] integer	Nombre text	Ciudad text	Barrio text	Provincia text
1	6789	Saturn	Coslada	Central	Madrid

Como hemos comentado anteriormente, primero se actualiza a MediaMarkt en T1 y después T2 intenta actualizarla a Saturn, pero queda bloqueada, ya que T1 no ha sido cerrada todavía y además, la actualización se guarda en la memoria local de T1. Una vez que se cierra T1, la tabla Tienda queda actualizada en memoria global y T2 es desbloqueada, por lo que ya puede hacer su actualización sin problemas, pero al no cerrarse, la actualización de Saturn se mantiene en la memoria local de T2, por lo que no es visible para el resto de usuarios. Una vez cerrada T2, la actualización de Saturn pasa a memoria global, por lo que ya es visible para cualquier usuario.

Cuestión 26: Cerrar todas las sesiones anteriores. Abrir una sesión con el usuario1 de la base de datos **TIENDA**. Abrir una transacción T1 en este usuario y realizar una modificación del ticket con número 54321 para cambiar su código a 223560. Abra otro usuario diferente del anterior y realice una transacción T2 que cambie la fecha del ticket con número 54321 a la fecha actual. No cierre la transacción.

Abrimos una sesión con el usuario1:

```
set role usuario1;
```

Abrimos la transacción y realizamos la actualización pedida:

```
begin;  
update "Ticket" set "Nº de ticket"=223560 where "Nº de ticket"=54321;
```

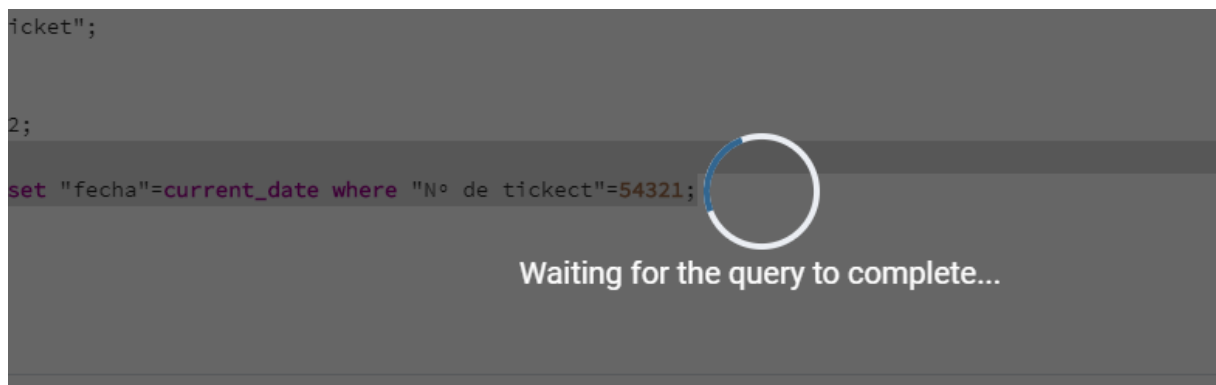
Abrimos en otra consola una sesión con el usuario2:

```
set role usuario2;
```

Abrimos otra transacción actualizando el ticket con la fecha actual:

```
begin;  
update "Ticket" set "fecha"=current_date where "Nº de ticket"=54321;
```

T2 se bloquea:



Esto es debido a que está intentando acceder a la misma tabla (Ticket) que T1.

Cuestión 27: Comprometa la transacción T1, ¿Qué es lo que ocurre? ¿Por qué? ¿Cuál es el estado de la información del ticket con código 54321 para ambos usuarios? ¿Por qué?

Comprometemos la transacción T1:

```
commit;
```

Se desbloquea T2, ya que T1 deja libre la tabla Ticket, por lo que T2 es desbloqueada y ya puede acceder a Ticket.

Para usuario1 la tabla ticket sería la siguiente:

	Nº de ticket [PK] integer	Importe integer	fecha date	codigo_trabajador_Trabajador integer
1	54300	40	2020-05-06	2
2	223560	15	2020-05-05	1

Vemos que el número de ticket se ha actualizado a 223560, por lo que la actualización se ha hecho correctamente. La fecha sigue la anterior y no la actualizada por T2, ya que esta última transacción no ha terminado por lo que el dato que ha actualizado no ha volcado a memoria global.

Vemos la información del usuario2:

	Nº de ticket [PK] integer	Importe integer	fecha date	codigo_trabajador_Trabajador integer
1	54300	40	2020-05-06	2
2	223560	15	2020-05-05	1

Vemos que el número de ticket es el mismo que en usuario1, ya que ya ha volcado a memoria global. La fecha no se ha modificado a la actual y sigue igual que anteriormente. Esto es debido a que, cuando se desbloquea, hace la actualización, pero ya no existe ningún ticket con número 54321 (T1 lo ha actualizado a 223560) en memoria global, por lo que no actualiza nada.

Cuestión 28: Comprometa la transacción T2, ¿Qué es lo que ocurre? ¿Por qué? ¿Cuál es el estado final de la información del ticket con número 54321 para ambos usuarios? ¿Por qué?

Realizamos el commit en T2:

```
commit;
```

La transacción termina correctamente, ya que no se ha producido ningún interbloqueo al ser desbloqueada T2 sin problemas.

La información para usuario1:

	Nº de ticket [PK] integer	Importe integer	fecha date	codigo_trabajador_Trabajador integer
1	54300	40	2020-05-06	2
2	223560	15	2020-05-05	1

Para usuario2:

	Nº de ticket [PK] integer	Importe integer	fecha date	codigo_trabajador_Trabajador integer
1	54300	40	2020-05-06	2
2	223560	15	2020-05-05	1

Vemos que es la misma. La razón es la misma que hemos comentado anteriormente, T1 actualiza el ticket 54321 a 223560 sin comprometerse y T2 intenta actualizar la fecha del ticket 54321. Al intentar acceder a la misma tabla que T1, se bloquea y espera a que T1 termine. T1 se compromete y ticket 54321 se actualiza a 223560 en memoria global. T2 se desbloquea e intenta actualizar la fecha del ticket 54321, pero ya no existe ningún ticket con ese código en memoria global, por lo que no actualiza nada y la tabla ticket se queda igual.

Cuestión 29: ¿Qué es lo que ocurre en el sistema gestor de base de datos si dentro de una transacción que cambia el importe del ticket con número 223560 se abre otra transacción que borre dicho ticket? ¿Por qué?

Iniciamos la transacción actualizando el importe:

```
begin;  
update "Ticket" set "Importe"=20 where "Nº de ticket"=223560;
```

Dentro de ella, abrimos otra intentando eliminar dicho ticket:

```
begin;  
delete from "Ticket" where "Nº de ticket"=223560;
```

Nos avisa de que ya hay una transacción activa, pero se borra:

```
WARNING: ya hay una transacción en curso  
DELETE 1
```

```
Query returned successfully in 51 msec.
```

Si vemos la información de ticket, podemos ver que se ha borrado perfectamente:

	Nº de ticket [PK] integer	Importe integer	fecha date	codigo_trabajador_Trabajador integer
1	54300	40	2020-05-06	2


PostgreSQL no soporta hacer dos transacciones anidadas y lo que se ha realizado en la transacción es borrar el ticket en la primera transacción que hemos abierto.

Cuestión 30: Suponer que se produce una pérdida del cluster de datos y se procede a restaurar la instancia de la base de datos del punto 6. Realizar solamente la restauración (recovery) mediante el procedimiento descrito en el apartado 25.3 del manual (versión, 12) "Continuous Archiving and point-in-time recovery (PITR)". ¿Cuál es el estado final de la base de datos? ¿Por qué?


Detenemos el servidor desde el administrador de tareas:

```
postgresql-x64-12 postgresql-x64-12 - PostgreSQL Serv... Detenido
```

Cambiamos de nombre al directorio del cluster (data):

 data_2	01/06/2020 13:42	Carpeta de archivos
--	------------------	---------------------


Restauramos la copia de seguridad que habíamos creado anteriormente.

 data	01/06/2020 13:49	Carpeta de archivos
--	------------------	---------------------

Cambiamos `restore_command` del `postgresql.conf` para restaurar el wal:

```
restore_command = 'copy "C:\\wal_backup\\%f" "%p"' # command to use to restore an archived logfile segment
```

Creamos el archivo `recovery.signal` para que nos indique el proceso de recuperación:

 recovery.signal	01/06/2020 13:49	Archivo SIGNAL	0 KB
---	------------------	----------------	------

Reiniciamos el servidor y se realiza el backup. Comprobamos en el log que el proceso de recuperación se ha realizado correctamente:

```
2020-06-01 14:14:48.394 CEST [8420] LOG: el sistema de bases de datos fue interrumpido; última vez en funcionamiento en 2020-06-01 14:09:48 CEST
2020-06-01 14:14:50.059 CEST [8420] LOG: comenzando proceso de recuperación
2020-06-01 14:14:50.103 CEST [8420] LOG: se ha restaurado el archivo «00000001000000090000000E» desde el área de archivado
2020-06-01 14:14:50.714 CEST [8420] LOG: redo comienza en 9/DE000060
2020-06-01 14:14:50.743 CEST [8420] LOG: el estado de recuperación consistente fue alcanzado en 9/DE000138
2020-06-01 14:14:50.746 CEST [9844] LOG: el sistema de bases de datos está listo para aceptar conexiones de sólo lectura
2020-06-01 14:14:50.786 CEST [8420] LOG: se ha restaurado el archivo «00000001000000090000000F» desde el área de archivado
2020-06-01 14:14:51.169 CEST [8420] LOG: se ha restaurado el archivo «00000001000000090000000E» desde el área de archivado
2020-06-01 14:14:51.484 CEST [8420] LOG: redo listo en 9/E0000070
2020-06-01 14:14:51.508 CEST [8420] LOG: última transacción completada al tiempo de registro 2020-06-01 14:11:12.096799+02
2020-06-01 14:14:51.538 CEST [8420] LOG: se ha restaurado el archivo «00000001000000090000000E» desde el área de archivado
2020-06-01 14:14:51.789 CEST [8420] LOG: seleccionado nuevo ID de timeline: 2
2020-06-01 14:14:52.142 CEST [8420] LOG: recuperación completa
2020-06-01 14:14:52.633 CEST [9844] LOG: el sistema de bases de datos está listo para aceptar conexiones
```

El estado final de la base de datos es el mismo que antes de realizar el proceso de recuperación. Esto es debido a que, al haber conservado los archivos del WAL, PostgreSQL rehace las operaciones realizadas después del recovery, y así, la base de datos se restaura a su punto más avanzado.

Cuestión 31: A la vista de los resultados obtenidos en las cuestiones anteriores, ¿Qué tipo de sistema de recuperación tiene implementado postgresQL? ¿Qué protocolo de gestión de la concurrencia tiene implementado? ¿Por qué? ¿Genera siempre planificaciones secuenciables? ¿Genera siempre planificaciones recuperables? ¿Tiene rollbacks en cascada? Justificar las respuestas.

El sistema de recuperación de PostgreSQL es modificación diferida (Redo), ya que los valores modificados en una transacción no vuelcan a memoria global hasta que se hace el commit, por lo que si hay un fallo tiene que rehacer las operaciones y no deshacer nada.

La gestión de la concurrencia se realiza a través del protocolo de dos fases riguroso refinado. En las preguntas 19 y 23 podemos ver que la transacción se bloquea al hacer update. Repasando los tipos de bloqueo de estas transacciones, los bloqueos concedidos son compartidos, por lo que si la transacción luego quiere escribir, tendrá que hacer un cambio de bloqueo (riguroso refinado). Todos los bloqueos se liberan tras comprometer a la transacción, por lo que esto indica que el protocolo es de dos fases.

En cuanto a las planificaciones, PostgreSQL general siempre planificaciones secuenciable y sin rollbacks en cascada, ya que utiliza el modelo multiversión de control de la concurrencia, MVCC. Cada transacción ve una versión de la base de datos anterior y no la real para evitar inconsistencias.

Bibliografía

- Capítulo 13: Concurrency Control.
- Capítulo 25: Backup and Restore.
- Capítulo 27: Monitoring Database Activity.
- Capítulo 29: Reliability and the Write-Ahead log.