10. Control de la concurrencia

10.1. Introducción

En este tema describiremos el comportamiento del SGBD PostgreSQL cuando dos o más sesiones intentan acceder a los mismos datos al mismo tiempo. El objetivo en esa situación es permitir a todas las sesiones un acceso eficiente a los datos, mientras se mantiene una integridad estricta de los mismos.

10.2. Transacciones

10.2.1. Introducción

10.2.2. `BEGIN`, `COMMIT` y `ROLLBACK`

10.2.3. Transacciones en PostgreSQL

A diferencia de otros SGBD que usan los bloqueos para controlar la concurrencia, PostgreSQL mantiene la consistencia de los datos mediante el uso de un modelo multiversión (Multiversion Concurrency Control, MVCC). Esto significa que mientras se realiza una consulta a una base de datos, cada transacción ve una instantánea de los datos (una versión de la base de datos) como era un tiempo antes, en lugar de ver el estado actual de los datos subyacentes. Esto protege a la transacción de los problemas que se producirían si pudieran ver datos inconsistentes que podrían ser causadas por actualizaciones de otras transacciones actualmente en curso sobre las mismas filas, proporcionando aislamiento entre transacciones para cada sesión.

La ventaja principal de usar el modelo MVCC de control de concurrencia en lugar de los bloqueos es que en MVCC los bloqueos adquiridos para consultas (lecturas) de datos no entran en conflicto con los bloqueos adquiridos para escrituras de datos, y por tanto las lecturas nunca bloquean a las escrituras, y las escrituras nunca bloquean a las lecturas.

Los bloqueos a nivel de tabla y de fila también están disponibles en PostgreSQL para aplicaciones que no puedan adaptarse fácilmente al comportamiento del MVCC. No ob-

stante, un uso apropiado del MVCC proporcionará generalmente mayores prestaciones que el uso de los bloqueos.

10.3. Aislamiento entre transacciones

El estándar SQL define cuatro niveles de aislamiento entre transacciones, en términos de tres fenómenos que deben ser evitados entre transacciones concurrentes. Estos **fenómenos indeseables** son:

1. **Lectura sucia**: las transacciones leen datos escritos por otras transacciones concurrentes sin que éstas hayan sido confirmadas[1].
2. **Lectura irrrepetible**: una transacción vuelve a leer datos que previamente había leído y encuentra que los datos han sido modificados por otra transacción que confirmó los cambios (es decir, que aplicó un `COMMIT` después de la lectura inicial).
3. **Lectura fantasma**: una transacción vuelve a ejecutar una consulta que devuelve un conjunto de filas que satisfacen una cierta condición, y encuentra que el conjunto de filas que satisfacen la condición ha cambiado debido a que otra transacción concurrente ha confirmado los cambios (ha aplicado un `COMMIT`).

Los cuatro niveles de aislamiento posibles, y sus correspondientes comportamientos, se describen en la siguiente tabla:

Nivel de aislamiento	Lectura sucia	Lectura irrepetible	Lectura fantasma
Lectura no confirmada	Posible	Posible	Posible
Lectura confirmada	Imposible	Posible	Posible
Lectura repetible	Imposible	Imposible	Posible
Serializable	Imposible	Imposible	Imposible

PostgreSQL ofrece los niveles de aislamiento «*Lectura confirmada*» y «*Serializable*».

10.3.1. Lectura confirmada

La *lectura confirmada* es el nivel de aislamiento por defecto en PostgreSQL. Cuando una transacción se ejecuta en este nivel de aislamiento, una consulta `SELECT` sólo ve los datos confirmados antes de que comezara la consulta; nunca podrá ver datos no confirmados o cambios confirmados por otras transacciones concurrentes durante la ejecución de la consulta. (No obstante, la `SELECT` ve los efectos de las actualizaciones previamente ejecutadas dentro de su propia transacción, incluso aunque éstas no hayan sido confirmadas.) En efecto, una consulta `SELECT` ve una instantánea de la base de datos tal y como

[1] Es decir, sin que hayan aplicado un COMMIT.

estaba en el instante antes de que la consulta empezara a ejecutarse. Obsérvese que dos comandos `SELECT` sucesivos pueden ver datos diferentes, incluso estando dentro de la misma transacción, si otras transacciones confirman cambios durante la ejecución del primer `SELECT`.

Los comandos `UPDATE`, `DELETE` y `SELECT FOR UPDATE` se comportan de la misma manera que `SELECT` a la hora de buscar filas objetivo: sólo encontrarán filas objetivo que estuvieran confirmadas en el momento de la ejecución del comando. No obstante, puede ser que una fila objetivo haya sido ya actualizada (o borrada, o marcada para ser actualizada) por otra transacción concurrente en el momento en que es encontrada. En ese caso, la que intenta actualizar tendrá que esperar a que la primera transacción actualizadora confirme o deshaga[2] sus cambios (si todavía están pendientes). Si la primera actualizadora deshace sus cambios, entonces sus efectos son negados y la segunda actualizadora podrá proceder con la actualización de la fila encontrada originariamente. Si la primera actualizadora confirma sus cambios, la segunda actualizadora ignorará la fila si la primera actualizadora la borra, y en caso contrario intentará aplicar su operación sobre la versión actualizada de la fila. La condición de búsqueda del comando (es decir, la cláusula `WHERE`) será reevaluada para ver si la versión actualizada de la fila todavía cumple la condición de búsqueda. Si es así, la segunda actualizadora procederá con su operación, comenzando por la versión actualizada de la fila.

Debido a la regla anterior, es posible que un comando actualizador logre ver una instantánea inconsistente: puede ver los efectos de comandos actualizadores concurrentes que afecten a las mismas filas que está intentando actualizar, pero no podrá ver los efectos de esos comandos en otras filas de la base de datos. Este comportamiento hace que la *lectura confirmada* sea inapropiada para aquellos comandos que involucren complejas condiciones de búsqueda. No obstante, es apropiada para casos más sencillos. Por ejemplo, considérese la actualización de los balances de un banco con transacciones como

```
BEGIN;
UPDATE cuentas
  SET balance = balance + 100.00
  WHERE num_cuenta = 12345;
UPDATE cuentas
  SET balance = balance - 100.00
  WHERE num_cuenta = 7534;
COMMIT;
```

Si dos transacciones de este tipo intentan cambiar concurrentemente el balance de la cuenta 12345, claramente querremos que la segunda transacción comience desde la versión actualizada de la fila de esa cuenta. Como cada comando afecta sólo a una fila predeterminada, hacer que la segunda transacción pueda ver la versión actualizada de la fila no crea ningún problema de inconsistencia.

[2] *roll back*.

Como en la *lectura confirmada* cada nuevo comando comienza con una nueva instantánea, que incluye todas las transacciones confirmadas hasta ese instante, los sucesivos comandos de la misma transacción verán en todo caso los efectos de las transacciones concurrentes confirmadas. La cuestión aquí es si dentro de un sólo comando tenemos o no una visión absolutamente consistente de la base de datos.

El aislamento parcial de transacciones que proporciona la *lectura confirmada* es adecuado para muchas aplicaciones, y este modo de trabajo es rápido y fácil de usar. No obstante, para aplicaciones que realicen complicadas consultas y actualizaciones, puede ser necesario garantizar una visión más rigurosamente consistente de la base de datos de la que nos proporciona la lectura confirmada.

10.3.2. Serializable

El nivel *serializable* proporciona el más estricto aislamiento entre transacciones. Este nivel simula la ejecución secuencial de las transacciones, como si las transacciones se hubieran ejecutado una después de la otra, en secuencia, en lugar de concurrentemente. No obstante, las aplicaciones que utilicen este nivel de aislamiento deben prepararse ante posibles reintentos de transacciones debidas a fallos en la serialización.

Cuando una transacción se encuentra en el nivel *serializable*, una consulta `SELECT` sólo ve los datos confirmados antes de que comenzara la transacción; nunca ve ni datos no confirmados ni cambios confirmados durante la ejecución de la transacción por otras transacciones concurrentes. (No obstante, la `SELECT` ve los efectos de las actualizaciones previamente ejecutadas dentro de su propia transacción, incluso aunque no hayan sido aún confirmadas.) Este comportamiento es diferente de el de la *lectura confirmada*, porque la `SELECT` ve una instantánea del comienzo de la transacción, y no del comienzo de la consulta actual dentro de la transacción. Por tanto, los sucesivos comandos SELECT dentro de una misma transacción siempre verán los mismos datos.

Los comandos `UPDATE`, `DELETE` y `SELECT FOR UPDATE` se comportan de igual manera que `SELECT` a la hora de buscar filas objetivo: sólo buscarán filas objetivo que hayan sido confirmadas en el momento en que comenzó la transacción. No obstante, puede ser que una fila objetivo haya sido ya actualizada (o borrada, o marcada para actualizar) por otras transacciones concurrentes en el momento en que es encontrada. En tal caso, la transacción serializable esperará a que la primera transacción actualizable confirme o deshaga los cambios. Si la primera actualizadora deshace sus cambios, entonces sus efectos serán negados y la transacción serializable podrá proceder con la actualización de la fila originalmente encontrada. Pero si la primera actualizadora confirma sus cambios (y por tanto actualiza o borra la fila, no sólo la selecciona para ser actualizada) entonces la transacción serializable se deshará con el siguiente mensaje:

```
ERROR: could not serialize access due to concurrent update
```

porque una transacción serializable no puede modificar filas cambiadas por otras transacciones después de que la transacción serializable comience.

Cuando la aplicación reciba este mensaje de error, debería abortar la transacción actual y reintentar la transacción completa desde el principio. La segunda vez, no obstante, la transacción verá los cambios previamente confirmados como parte de su visión incial de la base de datos, por lo que no habrá ningún conflicto lógico en usar la nueva versión de las filas como punto de comienzo de la nueva transacción actualizable.

Obsérvese que sólo las transacciones actualizables pueden necesitar reintentos; las transacciones que sólo leen nunca tendrán conflictos por serialización.

El modo *serializable* proporciona una garantía rigurosa de que cada transacción tendrá una visión totalmente consistente de la base de datos. No obstante, la aplicación deberá estar preparada para los posibles reintentos de transacción que tendrá que llevar a cabo cuando las actualizaciones concurrentes hagan imposible sostener la ilusión de la ejecución serializada. Ya que el coste de rehacer transacciones completas puede ser significativo, este modo sólo está recomendado cuando las transacciones actualizables contengan lógica lo suficiente compleja como para dar lugar a posibles respuestas erróneas en el modo de *lectura confirmada*. Lo normal es que el modo *serializable* sea necesario sólo cuando una transacción ejecute varios comando sucesivos que deban tener visiones idénticas de la base de datos.

10.3.3. SET TRANSACTION

```
SET TRANSACTION
    [ ISOLATION LEVEL { READ COMMITTED | SERIALIZABLE } ]
    [ READ WRITE | READ ONLY ];

SET SESSION CHARACTERISTICS AS TRANSACTION
    [ ISOLATION LEVEL { READ COMMITTED | SERIALIZABLE } ]
    [ READ WRITE | READ ONLY ];
```

La orden `SET TRANSACTION` fija las características de la transacción actual. No tiene ningún efecto sobre las transacciones posteriores. `SET SESSION CHARACTERISTICS` fija las características por defecto para todas las transacciones de una sesión. `SET TRANSACTION` cambia tales características para una transacción individual.

Las únicas características disponibles en una transacción son el nivel de aislamiento y el modo de acceso (lectura/escritura o sólo lectura).

El nivel de aislamiento de una transacción determina qué datos puede ver la transacción cuando otras transacciones se están ejecutando concurrentemente.

READ COMMITTED

Modo de aislamiento *lectura confirmada*. Una sentencia sólo puede ver las filas que se confirmaron antes de que empezara. Es la opción por defecto.

SERIALIZABLE

Modo de aislamiento *serializable*. La transacción actual sólo puede ver las filas confirmadas antes de que se ejecutara la primera consulta o sentencia de manipulación de datos en esta transacción.

Intuitivamente, *serializable* significa que dos transacciones concurrentes dejarán la base de datos en el mismo estado en el que quedaría si se hubieran ejecutado estrictamente una después de la otra en cualquier orden.

El nivel de aislamiento no puede fijarse después de que se haya ejecutado la primera consulta o sentencia de manipulación de datos (`SELECT`, `INSERT`, `DELETE`, `UPDATE`, `FETCH`, `COPY`) de una transacción.

El modo de acceso de la transacción determina si la transacción es de lectura/escritura o de sólo lectura. Por defecto, todas las transacciones son de lectura/escritura. Cuando una transacción es de sólo lectura, no puede usar los siguientes comandos SQL: `INSERT`, `UPDATE`, `DELETE` y `COPY TO`; todos los comandos `CREATE`, `ALTER` y `DROP`; `COMMENT`, `GRANT`, `REVOKE` y `TRUNCATE`; y `EXPLAIN ANALYZE` y `EXECUTE` si el comando que quieren ejecutar está entre los ya listados.

10.3.4. START TRANSACTION

```
START TRANSACTION
    [ ISOLATION LEVEL { READ COMMITTED | SERIALIZABLE } ]
    [ READ WRITE | READ ONLY ];
```

Este comando comienza una nueva transacción. Si se especifica un nivel de aislamiento o un modo de lectura/escritura, la nueva transacción tendrá tales características, como si se hubiera ejecutado un `SET TRANSACTION`. Por lo demás, el comportamiento de este comando es idéntico al del comando `BEGIN`.

Esencialmente, equivale a un `BEGIN` seguido de un `SET TRANSACTION`.

10.4. Bloqueos explícitos

PostgreSQL proporciona varios modos de bloqueo para controlar el acceso concurrente a los datos de las tablas. Estos modos puden usarse para crear bloqueos controlados por la aplicación, en situaciones en los que MVCC no da el comportamiento deseado. Además, muchos comandos de PostgreSQL adquieren automáticamente bloqueos de modos apropiados para asegurar que las tablas referenciadas no son borradas o modificadas de una manera incompatible mientras el comando se ejecuta. (Por ejemplo, `ALTER TABLE` no puede ejecutarse concurrentemente con otras operaciones sobre la misma tabla.)

Para examinar una lista de bloqueos actualmente pendientes en un servidor de bases de datos, puede usarse la vista del sistema `pg_locks`.

10.4.1. Bloqueos a nivel de tabla

La siguiente lista muetra los modos de bloqueo disponibles y el contexto en el que son usados automáticamente por PostgreSQL. Recordar que todos esos modos de bloqueo son bloqueos a nivel de tabla, aunque el nombre contenga la palabra «row» (fila); los nombres de los modos de bloqueo son históricos. Por extendernos un poco, los nombres reflejan el uso típico de cada modo de bloqueo (pero la semántica es siempre la misma). La única diferencia real entre un modo de bloqueo y otro es el conjunto de modos de bloqueo con los que entra en conflicto. Dos transacciones no pueden mantener bloqueos de modos conflictivos sobre la misma tabla al mismo tiempo. (No obstante, una transacción nunca entra en conflicto consigo misma. Por ejemplo, puede adquirir un bloqueo `ACCESS EXCLUSIVE` y posteriormente adquirir un bloqueo `ACCESS SHARE` sobre la misma tabla.) Varias transacciones pueden mantener bloqueos no conflictivos concurrentemente. Nótese en particular que algunos modos de bloqueo son auto-conflictivos (por ejemplo, un bloqueo `ACCESS EXCLUSIVE` no puede ser mantenido por más de una transacción al mismo tiempo) mientras que otros no son auto-conflictivos (por ejemplo, un bloqueo `ACCESS SHARE` puede ser mantenido por varias transacciones). Una vez adquirido, un bloqueo se mantiene hasta el final de la transacción.

10.4.1.1. Modos de bloqueo a nivel de tabla

ACCESS SHARE

Entra en conflicto sólo con el modo `ACCESS EXCLUSIVE`.

Los comandos `SELECT` y `ANALYZE` adquieren un bloqueo de este modo sobre las tablas referenciadas. En general, cualquier consulta que sólo lee una tabla y no la modifica adquirirá este modo de bloqueo.

ROW SHARE

Entra en conflicto con los modos `EXCLUSIVE` y `ACCESS EXCLUSIVE`.

El comando `SELECT FOR UPDATE` adquiere un bloqueo de este modo sobre la(s) tabla(s) destino (también bloquea con `ACCESS SHARE` cualesquiera otras tablas que son referenciadas pero no seleccionadas para ser actualizadas —`FOR UPDATE`—).

ROW EXCLUSIVE

Entra en conflicto con los modos `SHARE`, `SHARE ROW EXCLUSIVE`, `EXCLUSIVE` y `ACCESS EXCLUSIVE`.

Los comandos `UPDATE`, `DELETE` e `INSERT` adquieren un bloqueo de este modo sobre la tabla objetivo (también bloquea con `ACCESS SHARE` cualesquiera otras tablas

referenciadas). En general, este modo de bloqueo será adquirido por cualquier comando que modifique los datos de una tabla.

SHARE UPDATE EXCLUSIVE

Entra en conflicto con los modos SHARE UPDATE EXCLUSIVE, SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE y ACCESS EXCLUSIVE.

Este modo protege una tabla contra cambios concurrentes del esquema y ejecuciones de VACUUM.

Adquirido por el comando VACUUM (sin la cláusula FULL).

SHARE

Entra en conflicto con los modos ROW EXCLUSIVE, SHARE UPDATE EXCLUSIVE, SHARE ROW EXCLUSIVE, EXCLUSIVE y ACCESS EXCLUSIVE.

Este modo protege una tabla contra cambios concurrentes de datos.

Adquirido por CREATE INDEX.

SHARE ROW EXCLUSIVE

Entra en conflicto con los modos ROW EXCLUSIVE, SHARE UPDATE EXCLUSIVE, SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE y ACCESS EXCLUSIVE.

Este modo no es adquirido automáticamente por ningún comando de PostgreSQL.

EXCLUSIVE

Entra en conflicto con los modos ROW SHARE, ROW EXCLUSIVE, SHARE UPDATE EXCLUSIVE, SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE y ACCESS EXCLUSIVE.

Este modo permite únicamente bloqueos concurrentes de tipo ACCESS SHARE, es decir, sólo las lecturas pueden proceder en paralelo con una transacción que haya adquirido un bloqueo de este modo.

Este modo no es adquirido automáticamente por ningún comando de PostgreSQL.

ACCESS EXCLUSIVE

Entra en conflicto con todos los modos (ACCESS SHARE, ROW SHARE, ROW EXCLUSIVE, SHARE UPDATE EXCLUSIVE, SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE y ACCESS EXCLUSIVE).

Este modo garantiza que quien lo adquiere es la única transacción que puede acceder a la tabla, sea de la forma que sea.

Este modo es adquirido por los comandos ALTER TABLE, DROP TABLE, REINDEX, CLUSTER y VACUUM FULL. Este es además el modo de bloqueo por defecto cuando se usa el comando LOCK TABLE sin especificar un modo explícitamente.

Sólo un bloqueo ACCESS EXCLUSIVE puede bloquear una sentencia SELECT (sin cláusula FOR UPDATE).

10.4.1.2. LOCK TABLE

```
LOCK [ TABLE ] tabla [, ...]
    [ IN modo_bloqueo MODE ];
```

donde *modo_bloqueo* es:

```
ACCESS SHARE | ROW SHARE | ROW EXCLUSIVE | SHARE UPDATE EXCLUSIVE |
SHARE | SHARE ROW EXCLUSIVE | EXCLUSIVE | ACCESS EXCLUSIVE
```

`LOCK TABLE` adquiere un bloqueo a nivel de tabla, esperando si es necesario a que se liberen otros bloqueos que entren en conflicto con aquel que se quiere adquirir. Una vez obtenido, el bloqueo se mantiene hasta final de la transacción actual. (No existe un comando `UNLOCK TABLE`; los bloqueos siempre se liberan al final de la transacción.)

Cuando se adquieren bloqueos automáticamente por el uso de ciertos comandos que hacen referencias a tablas, PostgreSQL siempre utiliza el modo de bloqueo menos restrictivo posible. `LOCK TABLE` se proporciona para los casos en los que se necesitan bloqueos más restrictivos. Por ejemplo, supóngase que una aplicación ejecuta una transacción en el nivel de aislamiento *lectura confirmada* y necesita asegurarse de que los datos de una tabla permanecen estables durante todo el tiempo que dure la transacción. Para lograr esto podría obtenerse un bloqueo `SHARE` sobre la tabla antes de consultarla. Esto previene de cualquier cambio concurrente de los datos y asegura que las posteriores lecturas sobre la tabla tendrán siempre una visión estable de los datos confirmados, porque el modo de bloqueo `SHARE` entra en conflicto con el bloqueo `ROW EXCLUSIVE` adquirido por los escritores, y tu sentencia `LOCK TABLE` *table* `IN SHARE MODE` esperará hasta que cualquier poseedor concurrente de bloqueos `ROW EXCLUSIVE` confirme o deshaga sus cambios. Así, una vez que hayas obtenido el bloqueo, no habrá pendiente ninguna escritura no confirmada; además, no podrá comenzar ninguna hasta que se libere el bloqueo.

Para conseguir un efecto similar cuando se está ejecutando una transacción en el nivel de aislamiento *serializable*, hay que ejecutar la sentencia `LOCK TABLE` antes de ejecutar cualquier sentencia de modificación de datos. La visión de los datos que tendrá la transacción *serializable* se congelará cuando comience su primera sentencia de modificación de datos. Una posterior sentencia `LOCK TABLE` todavía evitará las escrituras concurrentes, pero no asegurará que lo que la transacción lea se corresponda con los últimos valores confirmados.

Si una transacción de este tipo va a cambiar los datos de una tabla, entonces debería usar un bloqueo `SHARE ROW EXCLUSIVE` en lugar de un bloqueo `SHARE`. Esto aseguro que sólo una transacción de este tipo se ejecutará a la vez. Sin él, podría ocurrir un interbloqueo: dos transacciones que intentan adquirir un bloqueo `SHARE`, y después son incapaces de adquirir un `ROW EXCLUSIVE` para realizar efectivamente sus cambios. (Nótese que los bloqueos de una misma transacción nunca entran en conflicto, por lo que una transacción puede adquirir un bloqueo `ROW EXCLUSIVE` mientras mantiene un bloqueo

SHARE —pero no si cualquier otra tiene ya un bloqueo SHARE—.) Para evitar interbloqueos, hay que asegurarse que todas las transacciones consiguen sus bloqueos sobre los mismos objetos en el mismo orden, y que si se involucran varios modos de bloqueo sobre un mismo objeto, entonces que todas las transacciones adquieran primero los más restrictivos.

LOCK ... IN ACCESS SHARE MODE requiere privilegios de SELECT sobre la tabla destino. Todas las otras formas de LOCK requieren privilegios de UPDATE o DELETE.

LOCK sólo es útil dentro de un bloque de transacción (par BEGIN/COMMIT), puesto que el bloqueo se pierde tan pronto como finaliza la transacción. Un comando LOCK fuera de un bloque de transacción constituye una transacción auto-contenida, por lo que el bloqueo se perderá tan pronto como sea obtenido.

LOCK TABLE sólo obtiene bloqueos a nivel de tabla, y por tanto los modos que llevan la palabra ROW tienen un nombre engañoso. Estos modos deberían leerse como indicativos de la intención del usuario de adquirir bloqueos a nivel de fila dentro de la tabla bloqueada. Además, el modo ROW EXCLUSIVE es un bloqueo de tabla compartible. Téngase siempre en mente que todos los modos de bloqueo tienen idéntico significado en lo que respecta a LOCK TABLE, diferenciándose únicamente en las reglas que dictan qué modos son incompatibles con cuáles.

Ejemplos

Obtener un bloqueo SHARE sobre una tabla cuando se van a realizar inserciones en una tabla con clave ajena que apunta a la primera:

```
BEGIN WORK;
LOCK TABLE peliculas IN SHARE MODE;
SELECT id
    FROM peliculas
    WHERE nombre = 'Star Wars: Episodio I - La Amenaza Fantasma';
-- Hacer ROLLBACK si no se devuelve el registro
INSERT INTO comentarios_peliculas
    VALUES (id, '¡GENIAL! ¡La esperaba desde hacía mucho!');
COMMIT WORK;
```

Obtener un bloqueo SHARE ROW EXCLUSIVE sobre una tabla cuando se va a realizar una operación de borrado:

```
BEGIN WORK;
LOCK TABLE peliculas IN SHARE ROW EXCLUSIVE MODE;
DELETE FROM comentarios_peliculas
    WHERE id IN (SELECT id
                 FROM peliculas
                 WHERE puntos < 5);
```

```
DELETE FROM peliculas WHERE puntos < 5;
COMMIT WORK;
```

10.4.2. Bloqueos a nivel de fila

Además de los bloqueos a nivel de tabla, existen los bloqueos a nivel de fila. Un bloqueo a nivel de fila sobre una fila específica se adquiere automáticamente cuando la fila es actualizada (o borrada, o marcada para ser actualizada). El bloqueo se mantiene hasta que la transacción se confirma o se deshace. Los bloqueos a nivel de fila no afectan a las consultas; sólo bloquean a los escritores que intentan escribir en la misma fila. Para adquirir un bloqueo a nivel de fila sobre una fila sin modificarla realmente, se puede seleccionar la fila con `SELECT FOR UPDATE`. Nótese que una vez que se adquiere un bloqueo a nivel de fila en particular, la transacción puede actualizar la fila varias veces sin medio a conflictos.

PostgreSQL no recuerda ninguna información sobre filas modificadas en memoria, por lo que no existe límite en el número de filas bloqueadas a un mismo tiempo. No obstante, bloquear una fila puede causar una escritura en el disco; así, por ejemplo, un `SELECT FOR UPDATE` modificará las filas seleccionadas para ser marcadas y eso dará lugar a escrituras en el disco.

Además de los bloqueos a nivel de tabla y fila, también se usan los bloqueos compartidos/exclusivos a nivel de página para controlar el acceso en lectura/escritura a las páginas de tablas en la piscina del búffer compartido. Estos bloqueos son liberados inmediatamente después de que una fila es obtenida o actualizada. Los desarrolladores de aplicaciones normalmente no tienen que preocuparse por los bloqueos a nivel de página, pero se mencionan aquí por completitud.

10.5. Interbloqueos

El uso de bloqueos explícitos puede incrementar el riesgo de interbloqueos, en los cuales dos (o más) transacciones adquieren bloqueos que otros quieren. Por ejemplo, si la transacción 1 adquiere un bloqueo exclusivo sobre la tabla A y luego intenta adquirir un bloqueo exclusivo sobre la tabla B, mientras que la transacción 2 ya tiene bloqueada en exclusiva la tabla B y ahora quiere un bloqueo exclusivo sobre la tabla A, entonces ninguna de ellas podrá continuar. PostgreSQL detecta automáticamente las situaciones de interbloqueo y las resuelve abortando una de las transacciones involucradas, permitiendo continuar a la(s) otra(s). (Qué transacción exactamente será abortada es difícil de predecir y no deberían hacerse suposiciones sobre eso.)

Nótese que los interbloqueos también pueden ocurrir como resultado de bloqueos a nivel de fila (y por tanto, también pueden ocurrir incluso si no se usan bloqueos explíci-

tos.) Considérese el caso en el que hay dos transacciones concurrentes modificando una tabla. La primera transacción ejecuta:

```
UPDATE cuentas
   SET balance = balance + 100.00
   WHERE num_cuenta = 11111;
```

Ésta adquiere un bloqueo a nivel de fila sobre la fila especificada por el número de cuenta. Después, la segunda transacción ejecuta:

```
UPDATE cuentas
   SET balance = balance + 100.00
   WHERE num_cuenta = 22222;
UPDATE cuentas
   SET balance = balance - 100.00
   WHERE num_cuenta = 11111;
```

La primera sentencia `UPDATE` adquiere con éxito un bloqueo a nivel de fila sobre la fila especificada, por lo que actuliza perfectamente esa fila. No obstante, la segunda sentencia `UPDATE` encuentra que la fila que está intentando actualizar ya ha sido bloqueada, por lo que espera a que la transacción que posee el bloqueo se termine. La transacción dos está ahora esperando a que la transacción uno se complete para poder continuar su ejecución. Ahora, la transacción uno ejecuta:

```
UPDATE cuentas
   SET balance = balance - 100.00
   WHERE num_cuenta = 22222;
```

La transacción uno intenta adquirir un bloqueo a nivel de fila sobre la fila especificada, pero no puede: la transacción dos ya posee ese bloqueo. Así que espera a que la transacción dos se complete. Por tanto, la transacción uno bloquea a la transacción dos, y la transacción dos bloquea a la transacción uno: un interbloqueo. PostgreSQL detectará esta situación y abortará una de las transacciones.

Generalmente, la mejor defensa contra los interbloqueos es evitarlos asegurándonos de que todas las aplicaciones que usen la base de datos adquieran sus bloqueos sobre varios objetos de una manera consistente. Esa fue la razón del interbloqueo del ejemplo anterior: si las dos transacciones hubiesen actualizado las filas en el mismo orden, no se habría producido el interbloqueo. Uno debe también asegurarse de que el primer bloqueo adquirido sobre un objeto en una transacción es del modo más alto necesario para ese objeto. Si no fuera factible verificar esto por adelantado, entonces se podrían gestionar los interbloqueos reintentando las transacciones que son abortadas debido al interbloqueo.

Si no se detectara una situación de interbloqueo, una transacción que buscase conseguir un bloqueo a nivel de fila o de tabla esperaría indefinidamente a que se liberasen los bloqueos conflictivos. Esto significa que es una mala idea mantener las transacciones

abiertas por largos periodos de tiempo (por ejemplo, mientras se espera la entrada del usuario).

10.6. Comprobación de consistencia de datos a nivel de aplicación

Debido a que los lectores en PostgreSQL no bloquean los datos, con independencia del nivel de aislamiento de la transacción, los datos leídos por una transacción pueden ser sobreescritos por otra transacción concurrente. En otras palabras, si una `SELECT` devuelve una fila, eso no significa que la fila permanece igual en el instante en que es devuelta (es decir, en algún momento después de que la consulta comience). La fila podría haber sido modificada o borrada por una transacción ya confirmada que confirmó sus datos después de que aquella empezara. Incluso si la fila es todavía válida «ahora», podría haber sido cambiada o borrada antes de que la transacción actual hiciera un `COMMIT` o un `ROLLBACK`.

Otra manera de pensar sobre ésto es imaginar que cada transacción ve una «foto» del contenido de la base de datos, y las trnasacciones que se ejecutan concurrentemente pueden ver fotos diferentes. Por tanto, el concepto absoluto del «ahora» es cuanto menos sospechoso en todo caso. Esto no resulta normalmente un grave problema si las aplicaciones cliente están aisladas unas de otras, pero si los clientes pueden comunicarse mediante canales externos a la base de datos, entonces pueden darse confusiones serias.

Para asegurar la validez actual de una fila y protegerla contra actualizaciones concurrentes se debe usar `SELECT FOR UPDATE` o una sentencia `LOCK TABLE` apropiada. (`SELECT FOR UPDATE` bloquea sólo las filas devueltas contra actualizaciones concurrentes, mientras que `LOCK TABLE` bloquea la tabla completa.) Esto debe ser tenido en cuenta cuando se porten aplicaciones a PostgreSQL provenientes de otros entornos.

Las comprobaciones globales de validez requieren una reflexión extra bajo MVCC. Por ejemplo, una aplicación bancaria podría querer comprobar que la suma de todos los créditos de una tabla es igual a la suma de los débitos en otra tabla, cuando ambas tablas están siendo activamente actualizadas. Comparar los resultados de dos comandos `SELECT sum(...)` sucesivos no funcionará adecuadamente bajo el modo *lectura confirmada*, ya que la segunda consulta podría incluir resultados de transacciones que no se contaron en la primera. Hacer las dos sumas en una sóla transacción *serializable* proporcionará una imagen precisa de los efectos de las transacciones que fueron confirmadas antes de que la transacción *serializable* empezara (pero uno podría preguntarse con toda lógica si la respuesta sería todavía relevante en el momento en que fue entregada). Si la transacción *serializable* por sí misma aplica algunos cambios antes de intentar realizar la comprobación de consistencia, la utilidad del chequeo resulta todavía más

discutible, puesto que ahora incluye algunos pero no todos los cambios producidos después de comenzar la transacción. En tales casos una persona cuidadosa podría intentar bloquear todas las tablas necesarias para la comprobación, con vistas a conseguir una imagien indiscutible de la realidad actual. Un bloqueo `SHARE` (o superior) garantiza que no existen cambios no confirmados en la tabla bloqueada, salvo aquellos producidos por la transacción actual.

Nótese además que si uno se apoya en los bloqueos explícitos para prevenir cambios concurrentes, debería usar el modo *lectura confirmada*, o usar el modo *serializable* con la precaución de obtener los bloqueos antes de realizar las consultas. Un bloqueo explícito obtenido en una transacción *serializable* garantiza que no se está ejecutando ninguna otra transacción que modifica la tabla, pero si la instantánea vista por la transacción se pierde al obtener el bloqueo, también se pueden perder algunos cambios ya confirmados en la tabla. La instantánea de una transacción serializable en realidad se congela al comienzo de su primera consulta o comando de manipulación de datos (`SELECT`, `INSERT`, `UPDATE` o `DELETE`), por lo que es posible obtener bloqueos explícitos antes de que se congele la instantánea.