BASES DE DATOS

3. Control de transacciones y concurrencias

Una **transacción** es una secuencia de instrucciones SQL que el SGBD gestiona como una unidad. Las sentencias `COMMIT`y `ROLLBACK`permiten indicar un fin de transacción.

Transacciones en MySQL

Las transacciones en MySQL sólo tienen sentido bajo el motor de almacenamiento InnoDB, que es el único motor transaccional de MySQL. Recuerde que los otros sistemas de almacenamiento son no transaccionales y, por tanto, cada instrucción que se ejecuta es independiente y funciona, siempre, de manera autocommitiva.

Ejemplo de transacción: operación en el cajero automático

Una transacción típica es una operación en un cajero automático, por ejemplo: si vamos a un cajero automático a sacar dinero de una cuenta bancaria esperamos que si la operación termina bien (y obtenemos el dinero extraídos) se refleje esta operación en la cuenta , y, en cambio, si ha habido algún error y el sistema no nos ha podido dar el dinero, esperamos que no se refleje esta extracción en nuestra cuenta bancaria. La operación, pues, necesario que se considere como una unidad y termine bien (*commit*), sin embargo, que si acaba mal (*rollback*) todo quede como estaba inicialmente antes de empezar.

Una transacción habitualmente comienza en la primera sentencia SQL que se produce después de establecer conexión en la base de datos, después de una sentencia COMMIT o después de una sentencia ROLLBACK.

Una transacción finaliza con la sentencia COMMIT, con la sentencia ROLLBACK o con la desconexión (intencionada o no) de la base de datos.

Los cambios realizados en la base de datos en el transcurso de una transacción sólo son visibles para el usuario que los ejecuta. Al ejecutar una COMMIT, los cambios realizados en la base de datos pasan a ser permanentes y, por tanto, visibles para todos los usuarios.

Si una transacción finaliza con ROLLBACK, se deshacen todos los cambios realizados en la base de datos para las sentencias de la transacción.

Recordemos que MySQL tiene la autocommit definido por defecto, por lo que se efectúa un **COMMIT** automático después de cada sentencia SQL de manipulación de datos. Para

```
SET autocommit = 0 ;
```

Con el fin de volver a activar el sistema de autocommit:

```
SET autocommit = 1 ;
```

Hay que tener en cuenta que una transacción sólo tiene sentido si no está definido el autocommit.

El funcionamiento de transacciones no es el mismo en todos los SGBD y, por tanto, habrá que averiguar el tipo de gestión que proporciona antes de querer trabajar.

3.1. Sentencia START TRANSACTION en MySQL

> **START TRANSACTION**define explícitamente el inicio de una transacción. Por lo tanto, el código que haya entre **start transaction**y **commit**o **rollback**formará la transacción.

Iniciar una transacción implica un bloqueo de tablas (**LOCK TABLES**), así como la finalización de la transacción provoca el desbloqueo de las tablas (**UNLOCK TABLES**).

En MySQL **start transaction**es sinónimo de **begin**y, también, de **begin work**. Y la sintaxis para **start transaction**es la siguiente:

```
START TRANSACTION [ WITH CONSISTENTE SNAPSHOT ] | BEGIN [ WORK ]
```

La opción **WITH CONSISTENT SNAPSHOT**inicia una transacción que permite lecturas consistentes de los datos.

3.2. Sentencias COMMIT y ROLLBACK en MySQL

> **COMMIT**define explícitamente la finalización esperada de una transacción. La sentencia **ROLLBACK**define la finalización errónea de una transacción.

La sintaxis de **commit**y **rollback**en MySQL es la siguiente:

```
COMMIT [ WORK ] [ AND [ NO ] CHAIN | [ NO ] RELEASE ]
ROLLBACK [ WORK ] [ AND [ NO ] CHAIN | [ NO ] RELEASE ]
```

La opción **AND CHAIN**provoca el inicio de una nueva transacción que comenzará apenas termine la actual.

BASES DE DATOS

Cuando se hace un **`rollback`**es posible que el sistema procese de manera lenta las operaciones, ya que un **`rollback`**es una instrucción lenta. Si se quiere visualizar el conjunto de procesos que se ejecutan se puede ejecutar la orden **`SHOW PROCESSLIST`**y visualizar los procesos que se están *deshaciendo* debido al **`rollback`**.

El SGBDR *MySQL* realiza una `COMMIT`implícita antes de ejecutar cualquier sentencia LDD (lenguaje de definición de datos) o `LCD`(lenguaje de control de datos), o en ejecutar una desconexión que no haya sido precedida de un error. Por tanto, no tiene sentido incluir este tipo de sentencias dentro de las transacciones.

3.3. Sentencias SAVEPOINT y ROLLBACK TO SAVEPOINT en MySQL

> Existe la posibilidad de marcar puntos de control (`savepoints`) en medio de una transacción, por lo que si se efectúa ROLLBACK este pueda ser total (toda la transacción) o hasta uno de los puntos de control de la transacción.

La instrucción **`SAVEPOINT`**permite crear puntos de control. Su sintaxis es la siguiente:

```
SAVEPOINT  < nom_punt_control > ;
```

La sentencia `ROLLBACK`para deshacer los cambios hasta un determinado punto de control tiene esta sintaxis:

```
ROLLBACK  [ WORK ]  TO   [ SAVEPOINT ]  < nom_punt_control > ;
```

Si en una transacción se crea un punto de control con el mismo nombre que un punto de control que ya existe, este queda sustituido por el nuevo.

Si se quiere eliminar el punto de control sin ejecutar un **`commit`**ni un **`rollback`**podemos ejecutar la siguiente instrucción:

```
release SAVEPOINT  < nom_punt_control >
```

Ejemplo de utilización de puntos de control

Consideremos la situación siguiente:

```
1
```

```
SQL > instrucció_A;
SQL >  SAVEPOINT PB;
SQL > instrucció_B;
SQL >  SAVEPOINT PC;
SQL > instrucción C;
SQL > instrucció_consulta_1;
SQL >  ROLLBACK  TO PC;
SQL > instrucció_consulta_2;
```

La instrucción de consulta 1 ve los cambios efectuados por las instrucciones A, B y C, pero el `ROLLBACK TO PC`deshace los cambios producidos desde el punto de control `PC`, por lo que la instrucción de consulta 2 sólo ve los cambios efectuados por las instrucciones A y B (los cambios para C han desaparecido), y el último `ROLLBACK`deshace todos los cambios efectuados por a y B, mientras que el último COMMIT los dejaría como permanentes.

3.4. Sentencias LOCK TABLES y UNLOCK TABLES

concurrencia

La concurrencia en la ejecución de procesos implica la ejecución simultánea de varias acciones, lo que habitualmente tiene como consecuencia el acceso simultáneo a unos datos comunes, que habrá que tener en cuenta a la hora de diseñar los procesos individuales a fin de evitar inconsistencia en los datos.

Con el fin de prevenir la modificación de ciertas tablas y vistas en algunos momentos, cuando se requiere acceso exclusivo a las mismas, en sesiones paralelas (o concurrentes), es posible bloquear el acceso a las tablas.

El bloqueo de tablas (**`LOCK TABLES`**) protege contra accesos inapropiados de lecturas o escrituras de otras sesiones.

La sintaxis para bloquear algunas tablas o vistas, e impedir que otros accesos puedan cambiar simultáneamente los datos, es la siguiente:

```
LOCK  TABLES  < nom_taula1 >  [ [ AS ]  < alies1 > ]  READ  |  WRITE
  [ ,  < Nom_taula1 >  [ [ AS ]  < alies1 > ]  READ  |  WRITE  ]  ...
```

La opción **`read`**permite leer sobre la mesa, pero no escribir. La opción **`write`**permite que la sesión que ejecuta el bloqueo pueda escribir sobre la mesa, pero el resto de sesiones sólo la puedan leer, hasta que termine el bloqueo.

A veces, los bloqueos se utilizan para simular transacciones (en motores de almacenamiento que no sean transaccionales, por ejemplo) o bien para conseguir acceso más rápido a la hora de actualizar las tablas.

Cuando se ejecuta **`lock tables`**se hace un **`commit`**implícito, por lo tanto, si había alguna transacción abierta, ésta termina. Si termina la conexión (normalmente o anormalmente) antes de desbloquear las tablas, automáticamente se desbloquean las tablas.

Es posible, también, en MySQL bloquear todas las tablas de todas las bases de datos del SGBD, para hacer, por ejemplo, copias de seguridad. La sentencia que lo permite es **`FLUSH`**

Para desbloquear las tablas (todas las que estuvieran bloqueadas) hay que ejecutar la sentencia **`UNLOCK TABLES`**.

3.4.1. Funcionamiento de los bloqueos

Cuando se crea un bloqueo para acceder a una mesa, dentro de esta zona de bloqueo no se puede acceder a otras tablas (a excepción de las tablas del diccionario del SGBD -**`information_schema`**-) hasta que no finalice el bloqueo. Por ejemplo:

```
mysql > LOCK TABLES t1 READ ;
mysql > SELECT COUNT ( * ) FROM t1;
+ ---------- +
| COUNT ( * ) |
+ ---------- +
| 3 |
+ ---------- +
mysql > SELECT COUNT ( * ) FROM t2;
ERROR 1100 ( HY000 ) : TABLE 't2' was NOT locked WITH LOCK TABLES
```

No se puede acceder más de una vez en la tabla bloqueada. Si se necesita acceder dos veces en la misma mesa, hay que definir un alias para el segundo acceso a la hora de hacer el bloqueo.

```
mysql > LOCK TABLE t WRITE , t AS t1 READ ;
mysql > INSERT INTO t SELECT * FROM t;
ERROR 1100 : TABLE 't' was NOT locked WITH LOCK TABLES
mysql > INSERT INTO t SELECT * FROM t AS t1;
```

Si se bloquea una tabla especificando un alias, hay que hacer referencia con este alias. Provoca un error acceder directamente con su nombre:

```
mysql > LOCK TABLE t AS myalias READ ;
mysql > SELECT * FROM t;
ERROR 1100 : TABLE 't' was NOT locked WITH LOCK TABLES
mysql > SELECT * FROM t AS myalias;
```

Si se quiere acceder a una tabla (bloqueada) con un alias, hay que definir el alias en el momento de establecer el bloqueo:

```
mysql > LOCK TABLE t READ ;
mysql > SELECT * FROM t AS myalias;
ERROR 1100 : TABLE 'myalias' was NOT locked WITH LOCK TABLES
```

3.5. Sentencia SET TRANSACTION

MySQL permite configurar el tipo de transacción con la sentencia **`set transaction`**.

La sintaxis para configurar las transacciones es:

```
SET [ GLOBAL | SESSION ] TRANSACTION Isolation LEVEL
  { READ UNCOMMITTED | READ Committed | REPEATABLE READ | Serializable }
```

afectar globalmente (**`GLOBAL`**) o bien afectar a la sesión en curso (**`SESSION`**).

- **`READ UNCOMMITTED`**: Se permite acceder a los datos de las tablas, aunque no se haya hecho un **`commit`**. Por lo tanto, es posible acceder a datos no consistentes (*dirty read*).
- **`READ COMMITTED`**: Sólo se permite acceder a datos que se hayan aceptado (**commit**).
- **`REPEATABLE READ`**(opción por defecto): permite acceder a los datos de manera consistente dentro de las transacciones, de manera que todas las lecturas de los datos, dentro de una transacción de tipo **`REPEATABLE READ`**, permitirán obtener los datos como al inicio de la transacción, aunque ya hubieran cambiado.
- **`SERIALIZABLE`**: Permite acceder a los datos de manera consistente en cualquier lectura de los datos, aunque no nos encontramos dentro de una transacción.