

4

Tratamiento de datos

OBJETIVOS DEL CAPÍTULO

- ✓ Aprender a realizar inserciones, modificaciones y borrados de datos en las bases de datos.
- ✓ Conocer los efectos de la integridad referencial sobre la manipulación de datos.
- ✓ Aprender a hacer exportaciones e importaciones de datos.
- ✓ Conocer la implementación de transacciones en bases de datos.
- ✓ Estudiar las políticas de bloqueo en bases de datos.

En este capítulo trataremos la parte del lenguaje SQL relacionado con la edición de información y las operaciones que incluye, es decir, inserción (*INSERT*), modificación (*UPDATE*) y eliminación (*DELETE*) de los datos de nuestras bases de datos.

4.1 INSERCIÓN DE REGISTROS

Existen varias formas de agregar información. Podemos hacerlo mediante una sentencia *INSERT* normal, usando un *SELECT* o cargando los datos desde un fichero. Vemos todas ellas a continuación.

4.1.1 CLÁUSULA INSERT

Es el tipo más simple de inserción. Admite tres tipos de sintaxis.

Formato 1

```
INSERT [LOW_PRIORITY | DELAYED | HIGH_PRIORITY] [IGNORE]
    [INTO] tbl_name [PARTITION (partition_name,...)]
    [(col_name,...)]
    {VALUES | VALUE} ({expr | DEFAULT},...),(...),...
    [ ON DUPLICATE KEY UPDATE
        col_name=expr
        [, col_name=expr] ... ]
    INSERT [INTO] { nombre_tabla | nombre_vista } [(lista columnas)]
    { VALUES (expresion [, expresion ]...) | sentencia_SELECT }.
```

Donde:

- **LOW PRIORITY**: hace que la inserción se retrase mientras haya clientes leyendo de la tabla afectada. Sirve solo para tablas que admiten bloqueos, como son las *MyISAM*, *MEMORY* y *MERGE*.
- **DELAYED**: permite continuar con otras operaciones mientras la inserción se retrasa hasta que no haya clientes accediendo a la tabla, es decir, es como la anterior pero permite continuar trabajando.
- **HIGH PRIORITY**: deshabilita el efecto de la variable de sistema *-low-priority-updates* si está activa (=1). Esta variable hace que las operaciones de modificación tengan menor prioridad que las de consulta.
- **IGNORE**: obvia los errores en la inserción. Por ejemplo, no podremos insertar registros con claves repetidas pero tampoco nos informará.
- **INTO**: es una cláusula opcional para indicar el nombre de la tabla o vista.
- **PARTITION**: especifica las particiones donde se pretenden insertar los datos.
- **DEFAULT**: sirve para usar el valor del campo por defecto creado cuando se definió la tabla.

- **Lista columnas:** podemos incluir entre paréntesis grupos de valores para insertar más de una fila en la misma sentencia.
- **Values:** es el conjunto de valores expresados mediante expresiones (normalmente valores literales) o procedentes de una consulta.
- **ON DUPLICATE KEY UPDATE:** cuando el valor de una clave (primaria o secundaria) se repite, ésta cláusula permite actualizar uno o varios campos del registro correspondiente.



EJEMPLO 4.1

Insertar en la tabla jugadores a Antonio Martínez del equipo 6 cuyo *id* de capitán es el 13, fecha de alta uno de enero de 2010, salario 45.000, altura 2,16 y cuyo puesto es pivot:

```
INSERT INTO jugadores VALUES(0,'Antonio', 'Martinez', 'pivot', 13, '2010-10-01',  
45000, 6, 2.16);
```

El primer campo es de tipo *autonumérico* de manera que su valor vendrá dado por el valor del último jugador más uno.

Para insertar valores *autonuméricos* ponemos cero. El sistema calculará automáticamente el valor numérico correspondiente.

Los valores numéricos van sin comillas a diferencia del resto de tipos.

Formato 2

```
INSERT [LOW_PRIORITY | DELAYED | HIGH_PRIORITY] [IGNORE]  
[INTO] tbl_name [PARTITION (partition_name,...)]  
SET col_name={expr | DEFAULT}, ...  
[ ON DUPLICATE KEY UPDATE  
    col_name=expr  
    [, col_name=expr] ... ]
```

En este caso se permite especificar el nombre y valor de las columnas explícitamente con *SET*.



EJEMPLO 4.2

Para insertar un nuevo jugador con nombre Juan, *id* 16 y el resto de valores por defecto:

```
INSERT INTO jugador SET id=16, nombre='Juan';
```

Formato 3

```
INSERT [LOW_PRIORITY | HIGH_PRIORITY] [IGNORE]
[INTO] tbl_name [PARTITION (partition_name,...)]
[(col_name,...)]
SELECT ...
[ ON DUPLICATE KEY UPDATE
  col_name=expr
  [, col_name=expr] ... ]
```

Ahora podemos insertar los valores procedentes de otra tabla o vista especificada mediante un *SELECT*.

**EJEMPLO 4.3**

Insertar en la tabla *jugadores_histórico* (con los mismos campos que la tabla *jugadores*) los datos de jugadores que se dieron de alta antes del año actual:

```
INSERT INTO jugadores_histórico SELECT * FROM jugadores WHERE YEAR(fecha_alta) <
YEAR(cur_date());
```

Ahora hemos incluido los registros cuyo año de alta es menor que el actual.

4.1.2 CLÁUSULA REPLACE

Funciona igual que *INSERT* excepto por el hecho de que si el valor de la clave primaria del registro a insertar coincide con un valor existente, éste se borrará para insertar el nuevo.

La sintaxis tiene también tres formas.

Formato 1

```
REPLACE [LOW_PRIORITY | DELAYED]
[INTO] tbl_name
[PARTITION (partition_name,...)]
[(col_name,...)]
{VALUES | VALUE} ({expr | DEFAULT},...),(...),...
```

Formato 2

```
REPLACE [LOW_PRIORITY | DELAYED]
[INTO] tbl_name
[PARTITION (partition_name,...)]
SET col_name={expr | DEFAULT}, ...
```

Formato 3

```
REPLACE [LOW_PRIORITY | DELAYED]
[INTO] tbl_name
[PARTITION (partition_name,...)]
[(col_name,...)]
SELECT ...
```

4.1.3 EXPORTACIÓN/IMPORTACIÓN DE DATOS

A veces nos interesará exportar datos a otras bases de datos o ficheros, o insertar grandes cantidades de registros procedentes de un fichero de datos o de un fichero que incluye sentencias *INSERT* o *REPLACE*. Para exportar disponemos del programa de copias de seguridad *mysqldump* y de la sentencia *SELECT*, mientras que para la inserción de datos disponemos de la cláusula *LOAD DATA* y el comando *source*, que nos permite ejecutar ficheros de comandos o instrucciones SQL.

LOAD DATA

Es una cláusula SQL para insertar datos a gran velocidad a partir de un fichero con cierto formato.

Su sintaxis es:

```
LOAD DATA [LOW_PRIORITY | CONCURRENT] [LOCAL] INFILE 'fichero'
[REPLACE | IGNORE]
INTO TABLE tbl_name
[PARTITION (partition_name,...)]
[FIELDS | COLUMNS]
[TERMINATED BY 'string']
[[OPTIONALLY] ENCLOSED BY 'char']
[ESCAPED BY 'char']]
[LINES
[STARTING BY 'string']
[TERMINATED BY 'string']]
[IGNORE number LINES]
```

1) *SELECT * FROM jugador INTO outfile '/home/ales/jugador.sql'*
 2) *LOAD DATA LOCAL INFILE 'jugador.sql'*
INTO TABLE jugador;

Show VARIABLES like '%secure_file_priv%';

Donde:

- **LOW_PRIORITY**: indica que su ejecución se realizará cuando no haya clientes leyendo la tabla. Solo sirve para el caso de tablas *MyISAM*.
- **CONCURRENT**: permite que otros procesos o clientes accedan a la lectura de datos de la tabla en la que tienen lugar las inserciones.
- **LOCAL**: hace que el fichero especificado sea leído desde el cliente que realiza la conexión. En caso de no estar presente el servidor entiende que debe leer el fichero en el propio servidor. La ruta al fichero debe ser absoluta o relativa en cuyo caso se toma como directorio base el de la base de datos activa (para Windows debe usarse doble barra en la especificación de la ruta).

- **INFILE 'fichero'**: indica el fichero que contiene los datos.
- **REPLACE | IGNORE**: hacen que en caso de existir el valor de una clave ésta se actualice (*REPLACE*) o se omita continuando con el resto de inserciones (*IGNORE*).
- **LINES STARTING BY**: indica que las líneas del fichero que contiene los datos comienzan por cierta cadena.
- **LINES TERMINATED BY**: indica la cadena que hay al final de cada línea en el fichero. Sirve para delimitar cada registro de datos.
- **FIELDS | COLUMNS**: es lo mismo que *LINES*. Sirve para delimitar el final de cada campo de datos.
- **OPTIONALLY ENCLOSED BY**: indica que los campos pueden estar delimitados por un carácter como comillas dobles.
- **FIELDS ESCAPED BY**: indica el carácter que se usará para “escapar” los caracteres de manera que no se confundan con los usados para delimitar campos, indicar separador de campos y el carácter fin de línea.
Por ejemplo, algunos campos pueden estar delimitados por comillas dobles y al mismo tiempo puede haber campos que contengan comillas como parte del valor del campo. Si no hubiese carácter de escape MySQL entendería las comillas como el final del campo, cuando no es así.
- **IGNORE**: sirve para especificar el número de líneas a omitir al principio del fichero. Sirve para cuando hay cabeceras o datos que no corresponden con campos de las tablas.

Los valores por defecto, si omitimos las cláusulas *FIELDS* y *LINES* son equivalente a lo siguiente:

```
FIELDS TERMINATED BY '\t' ENCLOSED BY '' ESCAPED BY '\\'
LINES TERMINATED BY '\n' STARTING BY '';
```



EJEMPLO 4.4

Disponemos de un fichero con datos de partidos llamado *partidos.txt* y situado en la unidad C del servidor. En él cada fila o registro está separado por un salto de línea y cada campo por un punto y coma.

Indica el comando necesario para cargar los datos en la tabla de partidos omitiendo los errores en la repetición de claves así como las dos primeras líneas:

```
LOAD DATA INFILE 'C:\\partidos.txt' IGNORE INTO TABLE partido FIELDS TERMINATED
BY ';' LINES TERMINATED BY '\\n';
```

SOURCE

Podemos usar ficheros de comandos para ejecutar inserciones. Esto es habitual cuando hacemos copias de seguridad con *mysqldump* ya que el resultado se genera en forma de comandos *INSERT* para los datos de las tablas.

Para ello, creamos un fichero que contenga los comandos de inserción a ejecutar (aunque pueden incluir cualquier tipo de comandos SQL). Luego lo ejecutamos del siguiente modo:

```
C:\\> mysql -uusuario -ppassword < ruta_fichero_de_comandos
```

De este modo, redireccionamos los comandos del fichero al cliente usando las credenciales correspondientes. Para ello debemos asegurarnos de que haya un comando *USE* antes de procesar datos para una tabla para los casos en que creamos el fichero a mano.

Si estamos conectados como clientes podemos hacer lo mismo con el comando *source* de MySQL de este modo:

```
mysql> source ruta_fichero_de_comandos
```

O equivalentemente con \.:

```
mysql> \. ruta_fichero_de_comandos
```

Si queremos ver el progreso de los comandos de forma paginada podemos usar:

```
C:\> mysql -uusuario -ppassword < ruta_fichero_de_comandos |more
```

O si preferimos almacenar la salida en otro fichero, como *myoutput*:

```
C:\> mysql -uusuario -ppassword < ruta_fichero_de_comandos > myoutput
```

LOAD XML (solo a partir de versión > 6.0.3)

Este comando permite cargar datos procedentes de un fichero *xml* en una tabla. Este fichero puede generarse con el programa MySQL usando la opción *xml* que genera los datos de una tabla en formato *xml*. Para ello usaremos lo siguiente:

```
C:\>mysql -xml -e "SELECT * FROM tabla" > fichero.xml
```

El proceso contrario requeriría el uso de *LOAD XML*.

La sintaxis es la siguiente:

```
LOAD XML [LOCAL] INFILE 'file_name' [REPLACE | IGNORE]
    INTO TABLE [db_name.]tbl_name
    [ROWS IDENTIFIED BY '<tagname>']
    [IGNORE number [LINES | ROWS]]
    [(column_or_user_var,...)]
    [SET col_name = expr,...]
```

Las opciones son las mismas que el resto de comandos comentados salvo *ROWS IDENTIFIED BY '<tagname>'*, que sirve para indicar la etiqueta que iniciará cada fila de la tabla.

Este comando soporta tres formatos XML:

Formato 1

El fichero *xml* se compone de filas que comienzan por *row*, cada columna y sus valores aparecen en forma de *atributo=valor*:

```
<row column1="value1" column2="value2" .../>
```

Formato 2

Las columnas aparecen como etiquetas y sus valores como el contenido de las mismas.

```
<row>
  <column1>value1</column1>
  <column2>value2</column2>
</row>
```

Formato 3

Los nombres de las columnas son los valores de los atributos de las etiquetas '*field*' y el contenido son los valores de dichas etiquetas.

```
<row>
  <field name='column1'>value1</field>
  <field name='column2'>value2</field>
</row>
```



EJEMPLO 4.5

Generar un fichero *xml* con los datos de los equipos:

```
C:\>mysql --xml -e "SELECT * FROM liga.equipos" > equipos.xml
```

Cargar los datos del fichero *equipo.xml* en la tabla equipo.

Si el fichero contiene lo siguiente:

```
<regequipo>
  <id_equipo>7</column1>
  <nombre>Unicaja</nombre>
  <ciudad>Málaga</ciudad>
  <web>Unicaja</web>
  <puntos>Unicaja</puntos>
</regequipo>
```

El comando sería:

```
LOAD XML INFILE 'equipo.xml' INTO TABLE equipo ROWS IDENTIFIED BY regequipo;
```

Exportar datos

Para la exportación de datos disponemos de la cláusula *SELECT* con la siguiente sintaxis:

```
SELECT...[INTO OUTFILE 'file_name' export_options
          | INTO DUMPFILE 'file_name'
```

Es decir, el resultado de la consulta se envía a un fichero llamado *filename* con las opciones de exportación correspondientes. Éstas son las mismas que para el caso de *LOAD DATA INFILE*. De hecho son comandos complementarios.

Si queremos algo más elaborado y potente disponemos de *mysqldump*, programa que incorpora MySQL para copias de seguridad.

Tiene tres usos básicos muy intuitivos:

```
mysqldump [opciones] nombre_de_base_de_datos [tablas]
mysqldump [opciones] --databases DB1 [DB2 DB3...]
mysqldump [opciones] --all-databases
```

Admite muchas opciones pero en este momento solo nos interesa la *-t* que provoca el volcado de todos los datos de las tablas seleccionadas en forma de órdenes *INSERT*.

Así para volcar por pantalla todos los datos de la base *liga* ejecutaríamos lo siguiente:

```
C:\>mysqldump -t liga
```

Si preferimos hacer el volcado a un fichero usaremos redirección:

```
C:\>mysqldump -t liga > C:\>liga.sql
```

ACTIVIDADES 4.1



- ▶ Haga una copia de seguridad de todos los datos de sus bases de datos usando *mysqldump*.
- ▶ Cree un fichero con registros de datos para tres nuevos equipos de la ACB. Separe los campos por guiones y las filas por espacios en blanco. Indique el comando necesario para cargar dicho fichero en la tabla *equipos*.
- ▶ Haga una copia de la base de datos *liga* en formato *xml*. Elimine los datos de equipo y recuperélos usando *LOAD XML*.
- ▶ Averigüe qué es un fichero de tipo *CSV*.
- ▶ Genere un *CSV* usando *Libreoffice Calc* con los datos de jugadores de la base *liga*.

4.2 MODIFICACIÓN DE REGISTROS

La modificación de información implica cambiar algunos o todos los valores de las columnas de una o varias tablas.

Para ello disponemos del comando *UPDATE* que puede funcionar en dos modos: en una única tabla y en modo *multitabla*. La sintaxis para ambos es la siguiente:

Formato para una sola tabla

```
UPDATE [LOW_PRIORITY] [IGNORE] table_REFERENCE  
    SET col_name1={expr1|DEFAULT} [, col_name2={expr2|DEFAULT}] ...  
    [WHERE where_condition]  
    [ORDER BY ...]  
    [LIMIT row_count]
```

Formato para varias tablas

```
UPDATE [LOW_PRIORITY] [IGNORE] table_REFERENCES  
    SET col_name1={expr1|DEFAULT} [, col_name2={expr2|DEFAULT}] ...  
    [WHERE where_condition]
```

El significado de las cláusulas es similar al de las secciones anteriores.

La condición *WHERE* es la misma que estudiamos en el tema de consultas.

Para el caso de múltiples tablas se incluye la sentencia *table_references* que permite incluir una combinación o *JOIN* de dos o más tablas.



EJEMPLO 4.6

Suba el salario de los jugadores del equipo 5 en 1.000 euros:

```
UPDATE jugador SET salario=salario+1000 WHERE equipo=5;
```



EJEMPLO 4.7

Añada un campo *id_capitan* en la tabla *equipo* y actualice los valores según la información de la tabla *jugador*:

```
UPDATE equipo e JOIN jugador j ON e.id_equipo=j.id_jugador SET e.id_capitan=j.capitan;
```

4.3 BORRADO DE REGISTROS

La eliminación de datos de nuestras tablas funciona básicamente igual que la actualización con la siguiente sintaxis tanto para una sola tabla como para varias:

Formato para una sola tabla

```
DELETE [LOW_PRIORITY] [QUICK] [IGNORE] FROM tbl_name  
[WHERE where_condition]  
[ORDER BY ...]  
[LIMIT row_count]
```

Si no usamos el filtro *WHERE* se borran todos los registros de la tabla indicada. Otra forma, más eficiente, se hace con el comando:

```
TRUNCATE TABLE tbl_name
```

Formato 1 para varias tablas

```
DELETE [LOW_PRIORITY] [QUICK] [IGNORE]  
tbl_name[.*] [,tbl_name[.*]] ...  
FROM table_references  
[WHERE where_condition]
```

Formato 2 para varias tablas

```
DELETE [LOW_PRIORITY] [QUICK] [IGNORE]  
FROM tbl_name[.*] [,tbl_name[.*]] ...  
USING table_references  
[WHERE where_condition]
```

En este último caso, la cláusula *table_references* sirve como en el caso anterior para definir una combinación de dos o más tablas.

Con este tipo de borrados podemos borrar datos de más de una tabla incluyendo otras que sirve para filtrar las filas.

Por ejemplo, si disponemos de tres tablas *t1*, *t2* y *t3* podemos borrar datos de dos de ellas, *t1* y *t2*, usando cualquiera de los siguientes comandos:

```
DELETE t1, t2 FROM t1, t2, t3 WHERE t1.id=t2.id AND t2.id=t3.id;  
DELETE FROM t1, t2 USING t1, t2, t3 WHERE t1.id=t2.id AND t2.id=t3.id;
```

La opción *QUICK* hace que, para tablas *MyISAM* se acelere la eliminación ya que los índices creados sobre las filas eliminadas no se reciclan de modo que se evita el trabajo de reorganización de los mismos. Esto es útil cuando se prevé que las filas que se inserten después tengan índices parecidos.

El asterisco se usa por compatibilidad con Access.

El comando SQL de borrado admite también las cláusulas *ORDER BY* y *LIMIT*.



EJEMPLO 4.8

Eliminar todos los equipos que no hayan jugado partidos como locales:

```
DELETE equipo FROM equipo e LEFT JOIN partido ON e.id_equipo=p.elocal WHERE p.elocal IS NULL;
```

ACTIVIDADES 4.2



- Averigüe la utilidad del comando *TRUNCATE* y su diferencia con *DELETE*.
- Indique el comando necesario para aumentar el salario de los jugadores de más de dos metros un 10%. Use *REPLACE* y *UPDATE* y compruebe la diferencia.
- Agregue usando MySQL la columna *puntos_equipo* a la tabla *jugadores* para reflejar todos los puntos de su equipo. Actualice sus valores con los valores correctos en la tabla *equipos*.
- ¿Con qué comandos elimina todos los datos de las tablas en la base liga?
- Intente aumentar el *id_equipo* de la tabla *equipos* en una unidad para todos los registros. Explique lo que ocurre y cómo podría solucionarlo.
- Borre registros de equipos que no hayan jugado partidos.

4.4

BORRADOS Y MODIFICACIONES E INTEGRIDAD REFERENCIAL

En bases de datos en las que hemos implementado integridad referencial en algunas de sus tablas debemos considerar las reglas de borrado y modificación que impusimos en el diseño a la hora de eliminar o modificar registros. Hasta ahora hemos obviado esta parte y, sin embargo, puede ser una gran fuente de errores si no lo tenemos en cuenta.

En el caso particular del gestor MySQL hay que tener esto en cuenta para las tablas o motores del tipo *InnoDB* que son las que permiten integridad referencial.

Así, en MySQL, tanto cuando borramos como cuando modificamos registros tenemos cuatro posibilidades en la definición de la integridad referencial:

■ NO ACTION

En este caso se impide la eliminación o modificación de un registro en una tabla que tiene registros relacionados (mediante alguno de sus tributos comunes) en otras tablas.

■ RESTRICT

Es equivalente a *NO ACTION*.

■ SET NULL

Permite borrar o actualizar un registro en la tabla “padre” poniendo a *NULL* el campo o campos relacionados en la tabla “hija” (salvo que se hayan definido como *NOT NULL*).

■ CASCADE

Es el valor más habitual ya que propaga las modificaciones o borrados a los registros de la tabla hija relacionados con los de la tabla padre.

Trabajar con integridad referencial puede generar muchos problemas si no observamos cuidadosamente lo anterior. En particular podemos distinguir los siguientes:

Inserción de datos

Es posible que cuando queramos hacer inserciones individuales o masivas de datos la integridad referencial suponga una fuente de errores, especialmente cuando hay relaciones entre campos de dos tablas.

Por ejemplo, si en la tabla de jugadores de la base de datos *liga* definimos una clave ajena (equipo) hacia la clave principal de la tabla *equipo* y en la tabla *equipo* definimos la clave ajena *id_capitan* hacia la clave principal de la tabla *jugador*. En este caso si queremos hacer una inserción masiva de todos los datos del equipo y de los jugadores la integridad referencial nos lo impedirá debido a que si insertamos primero los datos de equipos, no podremos introducir un equipo cuyo capitán no existe en la tabla *jugador* y si lo hacemos al revés insertando primero los datos de jugadores tampoco podemos introducir un jugador cuyo equipo no existe en la tabla *equipo*.

Este tipo de problemas se pueden evitar insertando primero los datos e impidiendo después las restricciones de integridad.

En todo caso, para inserciones masivas en las que no se da este tipo de relaciones dobles debemos prestar atención al orden de inserción de los datos. Para ello, introduciremos primero los datos de las tablas principales (aquellas a las que apuntan las claves ajenas) o las que no tienen claves ajenas y después secundarias o las que contienen claves ajenas que apuntan a dichas tablas principales.

Modificación de datos

Las modificaciones de datos no suelen generar problemas ya que se suele definir la integridad referencial de tipo *CASCADE* de modo que cualquier cambio en atributos clave se propaga automáticamente a las claves ajenas con las que se relaciona.

En todo caso, si la restricción es de tipo *SET NULL* será una fuente de valores nulos en los campos relacionados, además de que deberemos tener en cuenta que los campos de las claves ajenas se hayan definido como de tipo *NULL*.

Borrado de datos

Cuando borramos registros debemos tener en cuenta el tipo de borrado definido. Si, como suele ocurrir se definió el borrado en modo *CASCADE*, los datos de registros relacionados se borrarán automáticamente en todas las tablas de forma que ésta operación debe hacerse cuando se está muy seguro de sus implicaciones.

Para el caso de borrados masivos de todos los datos ocurre a la inversa que en el caso de la inserción.

A modo de ejemplo veamos qué posibilidades se darían si quisieramos borrar todos los datos de una base de datos:

- Si el modo es *CASCADE*: debemos eliminar en primer lugar los registros de las tablas principales. Por ejemplo, al eliminar un departamento desaparecerían también los registros de profesores que trabajen en él.
- Si el modo es *NO ACTION* o *RESTRICT*: deberemos eliminar en primer lugar los registros de las tablas que contienen las claves ajenas y después los de las tablas principales. En este caso tendríamos que eliminar primero los profesores de un departamento para poder eliminar el departamento.
- Si el modo es *SET NULL*: tendremos deberemos borrar los datos de cada tabla en cualquier orden.

4.5 MODIFICACIÓN DE DATOS EN VISTAS

Algunas vistas son actualizables. Esto significa que se pueden emplear en sentencias como *UPDATE*, *DELETE*, o *INSERT* para actualizar el contenido de las tablas subyacentes. Además las vistas reflejan instantáneamente los cambios producidos en las tablas de manera que aunque estos cambien se verán reflejados en la consulta que se haga sobre la vista.

Para que una vista sea actualizable, debe haber una relación uno a uno entre los registros de la vista y los registros de la tabla subyacente.

Existen además ciertas restricciones:

- ✓ Que no incluyan funciones de agregado.
- ✓ Que no incluyan cláusulas *DISTINCT*.
- ✓ Que no incluyan subconsultas.
- ✓ Que no se usen tablas temporales.
- ✓ No usar cláusulas *GROUP BY* ni *HAVING*.
- ✓ No usar uniones ni reuniones externas.
- ✓ No usar consultas correlacionadas.

Para el caso de reuniones internas (tipo *INNER*) podemos actualizar o insertar siempre y cuando los campos afectados sean únicamente los de una de las tablas implicadas en el *JOIN*.

Con respecto a la posibilidad de agregar registros mediante sentencias *INSERT*, es necesario que las columnas de la vista actualizable también cumplan los siguientes requisitos adicionales:

- No debe haber nombres duplicados entre las columnas de la vista.
- La vista debe contemplar todas las columnas de la tabla en la base de datos que no tengan indicado un valor por defecto.
- Las columnas de la vista deben ser *referencias* a columnas simples y no columnas derivadas. Una columna derivada es una que deriva de una expresión.

No podemos insertar registros en una vista conteniendo una combinación de columnas simples y derivadas, pero podemos actualizarla si actualizamos únicamente las columnas no derivadas.



EJEMPLO 4.9

Si tenemos la tabla *t1* de la base *test*, podemos crear la siguiente vista:

```
CREATE VIEW v AS SELECT a, 2*a AS b FROM t1;
```

Consistente en dos campos *a* y un campo derivado *b* cuyo valor es el doble de *a* para cada fila.

Entonces podríamos únicamente modificar el campo no derivado *a* con:

```
UPDATE v SET a=a+1;
```

Pero no modificar el campo derivado *b*:

```
UPDATE b SET b=b+1;
```

Error 1348: Column *b* is not updatable

La cláusula *WITH CHECK OPTION* puede utilizarse en una vista actualizable para evitar inserciones o actualizaciones en registros distintos de los determinados por la cláusula *WHERE* incluida en la definición de la vista.

Las opciones adicionales *LOCAL* y *CASCADE* hacen que la comprobación anterior afecte solo a la vista actual (*LOCAL*) o al resto de vistas de las que deriva (*CASCADE*).

*SHOW VARIABLES
like 'update_views_with_limit';*



EJEMPLO 4.10

Si hacemos las siguientes operaciones:

CREATE TABLE t1 (a INT);

```
CREATE VIEW v1 AS SELECT * FROM t1 WHERE a < 2 WITH CHECK OPTION;
```

```
CREATE VIEW v2 AS SELECT * FROM v1 WHERE a > 0 WITH LOCAL CHECK OPTION;
```

```
CREATE VIEW v3 AS SELECT * FROM v1 WHERE a > 0 WITH CASCDED CHECK OPTION;
```

Obtenemos las vistas *v3* y *v2* basadas ambas en la vista *v1* que, a su vez, se basa en una tabla *t1*.

Sin embargo las sentencias:

```
INSERT INTO v2 VALUES (2);
```

```
INSERT INTO v3 VALUES (2);
```

Producen un error en el segundo caso ya que la condición *WHERE* se comprueba en cascada, es decir, también en la vista *v1* subyacente.

ACTIVIDADES 4.3



- Indique si en las vistas creadas en el capítulo anterior son o no actualizables y por qué.
- Cree una vista con los nombres y equipo de los jugadores del CAI. Crea otra basada en la anterior con los nombres de jugadores del CAI. Modifique los nombres poniéndolos a mayúscula. Compruebe y explique el resultado en la tabla *base*.
- Cree dos vistas, una con los campos *nombre*, *id_jugador*, *equipo* y *altura* y otra con *id_jugador*, *equipo* y *nombre*, ambas basadas en la tabla *jugador*. Inserte un jugador nuevo en ellas y explique lo que ocurre.

4.6 TRANSACCIONES

Una transacción es un conjunto de órdenes (en nuestro caso comandos SQL) que se ejecutan de manera atómica o indivisible, es decir o se ejecutan todas o no se ejecuta ninguna.

Para ser consideradas como tales deben cumplir las cuatro propiedades ACID:

- **Atomicidad:** asegura que se realizan todas las operaciones o ninguna, no puede quedar a medias.
- **Consistencia:** o integridad que asegura que solo se empieza lo que se puede acabar.
- **Aislamiento:** asegura que ninguna operación afecta a otras pudiendo causar errores.
- **Durabilidad:** asegura que una vez realizada la operación, ésta no podrá cambiar y permanecerán los cambios.

El ejemplo clásico de transacción es una transferencia económica en la que debe sustraerse una cantidad de una cuenta, hacer una serie de cálculos relacionados con comisiones, intereses etc. e ingresar en otras cuentas. Todo ello debe ocurrir de manera simultánea como si fuese una sola operación ya que de otro modo se generarían inconsistencias.

Una de las características de los sistemas gestores es si permiten o no el uso de transacciones en sus tablas. En el caso de MySQL esto depende del tipo de tabla o motor utilizado. MySQL soporta distintos tipos de tablas tales como *ISAM*, *MyISAM*, *InnoDB* y *BDB* (*Berkeley Database*).

Las tablas que permiten transacciones son del tipo *InnoDB*. Están estructuradas de forma distinta que *MyISAM*, ya que se almacenan en un solo archivo en lugar de tres y, además de transacciones, permiten definir reglas de integridad referencial.

Las transacciones aportan una fiabilidad superior a las bases de datos. Si disponemos de una serie de operaciones SQL que deben ejecutarse en conjunto, con el uso de transacciones podemos tener la certeza de que nunca nos quedaremos a medio camino de su ejecución. De hecho, podríamos decir que las transacciones aportan una característica de “deshacer” a las aplicaciones de bases de datos.

Para este fin, las tablas que soportan transacciones, como es el caso de *InnoDB*, son mucho más seguras y fáciles de recuperar si se produce algún fallo en el servidor, ya que las instrucciones se ejecutan o no en su totalidad. Por otra parte, las transacciones pueden aumentar el tiempo de proceso de instrucciones.

Volviendo al ejemplo anterior, si una cantidad de dinero es transferida de la cuenta de un cliente (*cc1*) a otro (*cc2*), se requerirán por lo menos dos instrucciones de actualización:

```
UPDATE cuentas SET balance = saldo - cantidad_transferida WHERE cod_cliente=cc1;
UPDATE cuentas SET balance = saldo + cantidad_transferida WHERE cod_cliente = cc2;
```

Estas dos consultas deben trabajar bien pero, ¿qué sucede si ocurre algún imprevisto y se cae el sistema después de que se ejecuta la primera instrucción, pero la segunda aún no se ha completado? El cliente 1 tendrá una cantidad de dinero descontada de su cuenta y creerá que ha realizado su pago, sin embargo, el cliente 2 pensará que no se le ha depositado el dinero que se le debe. En este sencillo ejemplo se ilustra la necesidad de que las consultas, o bien sean ejecutadas de manera conjunta o que no se ejecute ninguna de ellas. Es aquí donde las transacciones toman un papel crucial.

Los pasos para usar transacciones en MySQL son:

- 1 Iniciar una transacción con el uso de la sentencia *START TRANSACTION* o *BEGIN*.
- 2 Actualizar, insertar o eliminar registros en la base de datos.
- 3 Si se quieren los cambios a la base de datos, completar la transacción con el uso de la sentencia *COMMIT*. Únicamente cuando se procesa un *COMMIT* los cambios hechos por las consultas serán permanentes.
- 4 Si sucede algún problema, podemos hacer uso de la sentencia *ROLLBACK* para cancelar los cambios que han sido realizados por las consultas que han sido ejecutadas hasta el momento.

En tablas *InnoDB* toda la actividad del usuario se produce dentro de una transacción. Si el modo de ejecución automática (*autocommit*) está activado, cada sentencia SQL conforma una transacción individual por sí misma. MySQL siempre comienza una nueva conexión con la ejecución automática habilitada.

Si el modo de ejecución automática se deshabilitó con *SET AUTOCOMMIT = 0*, entonces puede considerarse que un usuario siempre tiene una transacción abierta. Una sentencia SQL *COMMIT* o *ROLLBACK* termina la transacción vigente y comienza una nueva. Ambas sentencias liberan todos los bloqueos *InnoDB* que se establecieron durante la transacción vigente. Un *COMMIT* significa que los cambios hechos en la transacción actual se convierten en permanentes y se vuelven visibles para los otros usuarios. Por otra parte, una sentencia *ROLLBACK*, cancela todas las modificaciones producidas en la transacción actual.

Si la conexión tiene la ejecución automática habilitada, el usuario puede igualmente llevar a cabo una transacción con varias sentencias si la comienza explícitamente con *START TRANSACTION* o *BEGIN* y la termina con *COMMIT* o *ROLLBACK*.



EJEMPLO 4.11

Vamos a ejecutar algunas consultas para ver cómo trabajan las transacciones. Lo primero que tenemos que hacer es comprobar el estado de la variable *autocommit*:

```
mysql> SHOW VARIABLES LIKE 'autocommit';
```

Si tiene el valor 1 la desactivamos con *SET*. Otra opción es realizar los ejemplos con *START TRANSACTION*.

A continuación creamos una tabla, en la base de datos *test*, del tipo *InnoDB* e insertar algunos datos.

Para crear una tabla *InnoDB*, procedemos con el código SQL estándar *CREATE TABLE*, pero debemos especificar que se trata de una tabla del tipo *InnoDB* (*TYPE = InnoDB*). La tabla se llamará *trantest* y tendrá un campo numérico. Primero activamos la base *test* con la instrucción *USE* para después crear la tabla e introducir algunos valores:

```
mysql> USE test;
mysql> CREATE TABLE trantest(campo INT NOT NULL
PRIMARY KEY) TYPE = InnoDB; engine = InnoDB;
mysql> INSERT INTO trantest VALUES(1), (2), (3);
```

Una vez cargada la tabla iniciamos una transacción:

```
mysql> BEGIN;
Query OK, 0 rows affected (0.01 sec)
mysql> INSERT INTO trantest VALUES(4);
Query OK, 1 row affected (0.00 sec)
```

Si en este momento ejecutamos un *ROLLBACK*, la transacción no será completada, y los cambios realizados sobre la tabla no tendrán efecto:

```
mysql> ROLLBACK;
```

Si ahora hacemos un *SELECT* para mostrar los datos de *trantest* veremos cómo no se ha llegado a producir ninguna inserción.

Ahora vamos a ver qué sucede si perdemos la conexión al servidor antes de que la transacción sea completada:

```
mysql> BEGIN;
Query OK, 0 rows affected (0.00 sec)
mysql> INSERT INTO trantest VALUES(4);
Query OK, 1 row affected (0.00 sec)
mysql> SELECT * FROM trantest;
+-----+
| campo |
+-----+
| 1 |
| 2 |
| 3 |
| 4 |
+-----+
4 rows in set (0.00 sec)
mysql> EXIT;
Bye
```

Cuando obtengamos de nuevo la conexión, podemos verificar que el registro no se insertó, ya que la transacción no fue completada.

Ahora vamos a repetir la sentencia *INSERT* ejecutada anteriormente, pero haremos un *COMMIT* antes de perder la conexión al servidor al salir del monitor de MySQL:

```
mysql> BEGIN;
Query OK, 0 rows affected (0.00 sec)
mysql> INSERT INTO innertest VALUES(4);
Query OK, 1 row affected (0.00 sec)
mysql> COMMIT;
Query OK, 0 rows affected (0.02 sec)
mysql> EXIT;
Bye
```

Una vez que hacemos un *COMMIT*, la transacción es completada y todas las sentencias SQL que han sido ejecutadas previamente afectan de manera permanente a las tablas de la base de datos.

Hay instrucciones SQL que, dadas sus características, implican confirmación automática. Éstas son:

- Instrucciones del *DDL* que definen o modifican objetos de la base de datos. *CREATE, ALTER, DROP, RENAME* y *TRUNCATE*.
- Instrucciones que modifican tablas de la base de datos administrativa llamada *mysql.GRANT* y *REVOKE*.
- Instrucciones de control de transacciones y bloqueo de tablas. *START TRANSACTION, BEGIN, LOCK* y *UNLOCK*.
- Instrucciones de carga de datos. *LOAD DATA*
- Instrucciones de administración de tablas. *ANALIZE, CHECK, OPTIMIZE* y *REPAIR*.

Existen además tres comandos para la gestión de transacciones, son:

- *SAVEPOINT id*: es una instrucción que permite nombrar cierto paso en un conjunto de instrucciones de una transacción.
- *ROLLBACK TO SAVEPOINT id*: permite deshacer el conjunto de operaciones realizadas a partir del identificador de *savepoint*.
- *RELEASE SAVEPOINT id*: elimina o libera el *savepoint* creado.

Cualquier operación *COMMIT* o *ROLLBACK* sin argumentos eliminara todos los *savepoints* creados.



EJEMPLO 4.12

```
mysql> START TRANSACTION;  
mysql> INSERT INTO test.trantest VALUES(1);  
mysql> SAVEPOINT 1; Comenzar por carácter alfabético "P1"  
mysql> INSERT INTO test.trantest VALUES(2),(3),(4);  
mysql> ROLLBACK TO 1;
```

Este sencillo ejemplo hace que solo se inserten realmente (se confirmen) los datos del primer *INSERT*. Podemos comprobarlo con un *SELECT*.

ACTIVIDADES 4.4



- ▶ ¿Qué diferencia hay entre las instrucciones *START TRANSACTION* y *BEGIN*?
- ▶ Inicie dos terminales *MS-DOS* para conectarse al servidor MySQL. Inicie en una de ellas una transacción sobre la base *ebanca* en la que se van a transferir 100 euros de la cuenta 3 a la 4. Es decir, debe incrementarse el saldo de una y decrementarse el de la otra. Desde la otra terminal intente modificar la tabla de movimientos actualizando el saldo primero de la cuenta 5 y después de la 6. Antes de efectuar *COMMIT* en la primera consola, ¿qué ocurre? Haga ahora *COMMIT* en la primera consola. Explique lo que sucede.
- ▶ Si iniciamos una transacción y de pronto terminamos nuestra sesión, ¿qué operación hace el servidor, un *COMMIT*, un *ROLLBACK* o indefinido?
- ▶ En el ejercicio anterior, después de actualizar la cuenta 3, ¿podrá otro usuario tener acceso de lectura sobre la tabla *cuentas*? Explíquelo.
- ▶ ¿Qué ocurre si en mitad de una transacción sin confirmar modificamos la variable *autocommit* para ponerla a 1? ¿Se deshará o confirmará la transacción o habrá un resultado imprevisto?

4.7

POLÍTICAS DE BLOQUEO DE TABLAS

Bloquear una tabla o vista permite que nadie más pueda hacer uso de la misma de modo que tenemos la exclusividad de lectura y/o escritura sobre ella.

MySQL permite el bloqueo de tablas por parte de clientes con el objeto de cooperar con otras sesiones de clientes o de evitar que estos puedan modificar datos que necesitamos en nuestra sesión.

Los bloqueos permiten simular transacciones así como acelerar operaciones de modificación o de inserción de datos.

4.7.1 COMANDOS DE BLOQUEO DE TABLAS

La instrucción para bloqueo de tablas es *LOCK* sigue la siguiente sintaxis:

```
LOCK TABLES  
    tbl_name [[AS] alias] lock_type  
    [, tbl_name [[AS] alias] lock_type] ...  
  
lock_type:  
    READ [LOCAL]  
    | [LOW_PRIORITY] WRITE
```

Para desbloquear tablas (solo podemos desbloquear todas las tablas a la vez) usamos la instrucción *UNLOCK TABLES*.

Para obtener un bloqueo de todas las tablas de la base de datos usamos:

```
LOCK TABLES WITH READ LOCK
```

Bloquear una tabla implica indicar su nombre (*tbl_name*) o alias y el tipo de bloqueo, lectura o escritura (*READ/WRITE*).

4.7.2 TIPOS DE BLOQUEO

Existen dos tipos de bloqueo:

■ *READ [LOCAL]*

En este caso la sesión o cliente que tiene el bloqueo puede leer pero ni él ni ningún otro cliente podrá escribir en la tabla.

Es un bloqueo que pueden adquirir varios clientes simultáneamente y permite que cualquiera, incluso sin bloqueo, pueda leer las tablas bloqueadas.

El modificador *LOCAL* permite que haya inserciones concurrentes de otros clientes mientras dura el bloqueo.

■ *[LOW_PRIORITY] WRITE*

La sesión o cliente que adquiere este tipo de bloqueo puede leer y escribir en la tabla pero ningún otro cliente podrá acceder a ella o bloquearla.

4.7.3 ADQUISICIÓN-LIBERACIÓN DE UN BLOQUEO

Cuando necesitamos adquirir bloqueos debemos hacerlo en una única sentencia ya que si separamos las instrucciones automáticamente se desbloquean las anteriores quedando activo solamente el último bloqueo.

En ese momento solo tendremos acceso a las tablas bloqueadas.

Los bloqueos de escritura tienen por defecto mayor prioridad ya que implican modificación de datos que deben hacerse lo más rápido posible. Así si una sesión adquiere un bloqueo de lectura y otra sesión pretende uno de *lectura*, éste tendrá prioridad sobre otras solicitudes de bloqueos de lectura subsecuentes. De este modo hasta que la sesión que solicitó el bloqueo de escritura no libere dicho bloqueo no se podrán adquirir nuevos bloqueos de lectura.

Esto cambia si el bloqueo de escritura se adquiere con la opción *LOW_PRIORITY* en cuyo caso si se permite que los bloqueos de lectura se adquieran antes que el de escritura. De hecho el bloqueo de escritura solo se obtendrá cuando no queden bloqueos de lectura pendientes.

La política de bloqueos de MySQL sigue la siguiente secuencia:

- ✓ 1. Ordena internamente las tablas a bloquear.
- ✓ 2. Si una tabla debe bloquearse para lectura y escritura sitúa la solicitud de bloqueo de escritura en primer lugar.
- ✓ 3. Se bloquea cada tabla hasta que la sesión obtiene todos sus bloqueos.

De este modo se evita el conocido *deadlock* o bloqueo mutuo, fenómeno que hace que el acceso a los bloqueos se pueda prolongar indefinidamente al no poder ningún proceso obtenerlos.

Si adquirimos un bloqueo sobre una tabla, todos los bloqueos activos en ese momento se liberan automáticamente.

Si comenzamos una transacción todos los bloqueos se liberan automáticamente.

4.7.4 BLOQUEOS Y TRANSACCIONES

El uso de bloqueos está íntimamente ligado a las transacciones, especialmente para tabla de tipo transaccional como *InnoDB* y *CLUSTER*.

En tablas *InnoDB* los bloqueos se adquieren a nivel de fila permitiéndose que varios usuarios puedan bloquear varias filas simultáneamente.

En tablas *InnoDB* todo es una transacción, de hecho, como ya hemos señalado, si *autocommit* está activado (=1), cada operación SQL es en sí misma una transacción. En este caso podemos iniciar una transacción con *START TRANSACTION* o *BEGIN* y terminarla con *COMMIT* para que los cambios sean permanentes o *ROLLBACK* para deshacer los cambios.

En el caso de *autocommit* inactivo se considera que siempre hay una transacción en curso en cuyo caso las sentencias *COMMIT* y *ROLLBACK* suponen el final de dicha transacción y comienzo de la siguiente.

Tipos de bloqueo en *InnoDB*

En este tipo de tablas diferenciamos dos tipos de bloqueo:

- **Bloqueo compartido (s):** permite a una transacción la lectura de filas. En este caso varias transacciones pueden adquirir bloqueos sobre las mismas filas pero ninguna transacción puede modificar dichas filas hasta que no se liberen los bloqueos.
- **Bloqueo exclusivo (x):** permite a una transacción bloquear filas para actualización o borrado. En este caso las transacciones que deseen adquirir un bloqueo exclusivo deberán esperar a que se libere el bloqueo sobre las filas afectadas.

También se soporta el “bloqueo de múltiple granularidad”, según el cual una transacción puede indicar que va a bloquear algunas filas de una tabla bien para lectura (IS) o para escritura (IX). Es lo que se denomina una **intención de bloqueo**.

De este modo es más fácil para MySQL gestionar posibles conflictos evitando los temidos *deadlocks* ya que permite a varias transacciones compartir la reserva de una tabla puesto que el bloqueo se produce fila a fila en el momento de la modificación. Así, si dos transacciones reservan la misma tabla, solo en el momento en que una de ellas esté modificando una fila, ésta quedará bloqueada.

Ejemplos de este tipo de bloqueo son:

- ✓ Sentencias *SELECT...LOCK IN SHARE MODE*: para bloqueo tipo IS. De este modo se crea un bloqueo de lectura sobre las filas afectadas por la consulta *SELECT*.



EJEMPLO 4.13

Supongamos que queremos agregar un registro en la tabla *jugador* pero asegurándonos de que existe su equipo (suponemos además que no hay integridad referencial).

Para asegurar que existe el equipo haríamos una consulta sobre *equipo* para comprobarlo y después insertaríamos el jugador. Sin embargo, nadie nos asegura que entre medio se elimine dicho equipo. Para evitarlo haríamos la consulta de este modo:

```
SELECT * FROM equipo WHERE nombre='id_equipo' LOCK IN SHARE MODE;
```

Con lo que conseguimos un bloqueo de lectura de modo que mientras no se confirme o deshaga la transacción nadie podrá modificar los datos de *equipo*.

- ✓ Sentencias *SELECT ... FOR UPDATE*: para bloqueo tipo IX. De este modo se bloquean para escritura las filas afectadas por la consulta *SELECT* así como las entradas de índice asociadas.



EJEMPLO 4.14

Si queremos añadir un nuevo jugador con un *id* determinado por el valor de un contador almacenado en la tabla *contador*. Si hacemos un bloqueo de lectura es posible que más de una transacción acceda al mismo valor de contador para dos jugadores distintos de manera que se producirá un error al ser un campo clave y no poder repetirse. Para evitarlo usaríamos un bloqueo de escritura o exclusivo para después incrementar el contador con la siguiente orden:

```
SELECT num_jugadores FROM contador FOR UPDATE;  
UPDATE num_jugadores SET num_jugadores= num_jugadores+1;
```

Ahora cualquier transacción que quiera leer el campo *num_jugadores* deberá esperar a que se libere el bloqueo exclusivo y, por tanto, obtendrá el valor correcto.

Select SQL_NO_CACHE * From tabla

Niveles de aislamiento

Los niveles de aislamiento hacen referencia al grado de actualización de los datos que leemos de una base de datos. Así distinguimos 4 niveles:

■ READ UNCOMMITTED

Las lecturas se realizan sin bloqueo de manera que podemos estar leyendo un dato que ya ha sido modificado por otra transacción. Es lo que se denomina lectura inconsistente o sucia.

■ READ COMMITED

En este tipo de lectura en cada instrucción se usan los valores más recientes y no los de comienzo del bloqueo. Es decir, si en alguna instrucción se modifica un valor que después se usará en otra, el valor será el modificado y no el original cuando se adquirió el bloqueo.

■ REPEATABLE READ

A begin update nombre = autorre 11 commit.

Es el valor por defecto. En este caso se obtiene el valor establecido al comienzo de la transacción. Es decir, aunque durante la transacción los valores cambien, se tendrán en cuenta siempre los del comienzo.

■ SERIALIZABLE

Es como la anterior con la diferencia de que ahora de manera interna se convierten los *SELECT* en *SELECT ... LOCK IN SHARE MODE* si *autocommit* está desactivado.

Con el comando *SET TRANSACTION* podemos establecer el nivel deseado de aislamiento de nuestras transacciones indicando si es a nivel global o solo para nuestra sesión actual.

La sintaxis es:

Select @@tx_isolation; en la sesión actual
SET [GLOBAL | SESSION] TRANSACTION ISOLATION LEVEL
{
 | READ UNCOMMITTED
 | READ COMMITTED
 | REPEATABLE READ
 | SERIALIZABLE
}

Select @@global.tx_isolation; en todas las sesiones

4.7.5 INSERCIÓNES CONCURRENTES

Son inserciones que permiten la lectura simultánea de datos de una tabla. Es decir, mientras en una sesión se está modificando una tabla, otras sesiones pueden leer de dicha tabla.

El motor de MySQL *MyISAM* soporta inserciones concurrentes para facilitar la competencia de lectura y escritura sobre este tipo de tablas. Esto se da normalmente cuando la tabla no contiene huecos en sus ficheros de datos, es decir, no hay filas borradas entre el resto de filas.

Este comportamiento está controlado por una variable de sistema llamada *concurrent_insert* cuyos valores posibles son los siguientes:

- AUTO ó 1: se activan las inserciones concurrentes.
- 0: se impiden las inserciones concurrentes.
- 2: se permiten las inserciones incluso aunque haya filas borradas entre los datos de la tabla.



Estudiaremos las variables de sistema en el segundo curso de este ciclo en administración de bases de datos. De momento basta con saber que existen y tienen ciertos valores que se pueden consultar y modificar con los comandos *SHOW VARIABLES* y *SET*, respectivamente.

Para el caso e que estén habilitadas este tipo de inserciones no se hace necesario el uso de la cláusula *DELAYED* del comando *INSERT* por motivos obvios.

Para el caso de la sentencia *LOAD DATA* si usamos la cláusula *CONCURRENT* permitimos el uso de inserciones concurrentes siempre que se cumpla que la tabla no contiene huecos.

La cláusula *HIGH PRIORITY* también desactiva el uso de inserciones concurrentes.

Para bloqueos de tablas para lectura la cláusula *LOCAL* permite el uso de inserciones concurrentes que se ejecutan mientras el bloqueo está activo.

ACTIVIDADES 4.5



- ¿Qué tipo de tablas soportan bloqueo?
- ¿Es lo mismo usar *BEGIN* que *LOCK* en tablas *InnoDB*?
- Si adquiero un bloqueo *FOR UPDATE*, ¿podrá otro usuario modificar la tabla bloqueada?
- Indique el comando necesario para comprobar el nivel de aislamiento de su servidor.
- Modifíquelo a *READ COMMITTED* e inicie una transacción para mover 100 euros de la cuenta 1 a la 2. No ejecute todavía la orden *COMMIT*.
- Lea ahora los valores del saldo de ambas cuentas desde otro cliente antes y después de hacer *COMMIT* en la transacción anterior.
- Repita lo anterior (la transacción y lectura de saldos) usando el nivel de aislamiento *REPEATABLE READ* y explique la diferencia.



RESUMEN DEL CAPÍTULO

Este capítulo se ha dedicado a la manipulación de tablas y datos en bases de datos relacionales en lo que respecta a la inserción, importación, exportación y modificación de los mismos.

Hemos visto cómo insertar datos individuales y de manera masiva usando distintas utilidades de importación.

También hemos estudiado cómo hacer copias de seguridad y exportar datos a ficheros y a otras tablas.

Hemos estudiado la modificación y borrado de datos en tablas y vistas.

Finalmente, hemos visto el concepto de transacción, las políticas de bloqueo de tablas y cómo se implementan en MySQL.



EJERCICIOS PROPUESTOS

- 1. Si hay dos inserciones simultáneas sobre un mismo registro de datos, una con la opción *DELAYED* y la otra con *HIGH PRIORITY*, ¿cuál se ejecutará antes? ¿Y si hay una tercera inserción, también simultánea y sin opciones?
- 2. Actualice el valor de puesto para el jugador con *id = 4* cambiándolo a *ala-pivot* de tres formas distintas, usando una cláusula *INSERT*, *REPLACE* y *UPDATE*.
- 3. Cree un fichero con datos de equipos de la liga ACB para insertarlos en la tabla *equipos* (separa los campos por comas y los registros por un guión). Incluya al principio una descripción con el contenido del fichero y la fecha de creación. Haga lo necesario para insertarlos usando *LOAD DATA*.
- 4. Tenemos un fichero de noticias con registros de muchas noticias, pero algunas están repetidas, ¿cómo podemos insertarlas todas de golpe sin preocuparnos de los errores debidos a la repetición de claves?
- 5. Cree un fichero a partir de un fichero *rss* (en formato XML) de uno de tus *blogs* o periódicos preferidos y adáptelo para insertarlo en la tabla *noticias* de la base *motorblog* usando *LOAD XML*.
- 6. Borre el equipo y sus jugadores para el equipo con menos puntos.
- 7. Cree un vista con los datos de autores y noticias de la base *motorblog*. ¿Es actualizable?, ¿en qué casos? Compruébelo con un ejemplo.
- 8. ¿Tiene sentido el uso de transacciones en la base de datos de un *blog*? Explíquelo.

- 9. Cree una transacción con las siguientes instrucciones:
1. Active la base *ebanca*.
 2. Aumentar el saldo la cuenta 3 un 10%.
 3. Descontar el importe correspondiente en la cuenta 4.
 4. Generar los registros correspondientes en la tabla movimientos.
 5. Terminar.

- 10. Durante la transacción anterior (antes de su confirmación), ¿qué información ven los usuarios que consulten las cuentas afectadas?, ¿está actualizada? Explíquelo.



TEST DE CONOCIMIENTOS

1 ¿Qué es cierto respecto al comando *INSERT*?

- Permite insertar registros de datos en tablas.
- Permite actualizar datos en tablas.
- Permite volcar información de tablas en un fichero.
- Todo lo anterior.

2 Una inserción con alta prioridad:

- Se ejecuta siempre antes que cualquier consulta.
- Se ejecuta con mayor prioridad que las consultas.
- Se ejecuta solo si no hay otra sentencia pendiente.
- Se ejecuta antes que los *SELECT*, *UPDATE* y *DELETE* que haya en ese momento.

3 Un borrado con baja prioridad:

- Se ejecuta solo cuando nadie ha bloqueado las tablas.
- Se ejecuta solo cuando no hay clientes leyendo la tabla.
- Se ejecuta en cuanto hay pocas consultas sobre la tabla.

4 Las inserciones concurrentes:

- Permiten insertar varios registros de datos simultáneamente.
- Permiten que dos clientes escriban simultáneamente en una tabla.
- Permiten inserciones y lecturas concurrentes.
- Todo lo anterior.

5 XML:

- Es lo mismo que *xhtml*.
- Es una especificación para codificar documentos.
- Solo sirve para publicar noticias.
- Es un tipo de dato en MySQL.

6 De los siguientes programas y comandos, ¿cuáles me sirven para hacer una copia de seguridad de mis datos?

- SELECT*.
- INSERT*.
- mysqldump*.
- Todos los anteriores.

7

La variable *autocommit*:

- a) Hace un *ROLLBACK* automático si hay algún problema en una transacción.
- b) Hace un *ROLLBACK* automático si hay algún problema en una transacción.
- c) Hace *COMMIT* tras la ejecución cada instrucción SQL.
- d) Con valor cero inhabilita el uso de transacciones.

8

Iniciar una transacción:

- a) Equivale a bloquear todas las tablas.
- b) Equivale a desbloquear todas las tablas bloqueadas previamente para esa sesión con *LOCK TABLES*.
- c) Bloquea todas las tablas para escritura.
- d) Bloquea para escritura las tablas propiedad del usuario.

9

Obtener un bloqueo de lectura sobre una tabla:

- a) Implica que nadie más va a poder leer la tabla.
- b) Me asegura que no será actualizada mientras la leo.
- c) Me permite modificar los datos mientras mantiene los datos originales para acceso de lectura del resto de usuarios.

10

¿Para qué sirven los bloqueos de tablas?

- a) Para acelerar la actualización de datos.
- b) Para emular transacciones en tablas no transaccionales.
- c) Para todo lo anterior.